ELSEVIER

Theoretical Computer Science 173 (1997) 3–48

Theoretical Computer Science

Helios: A modeling language for global optimization and its implementation in Newton

Laurent Michel, Pascal Van Hentenryck *

Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, USA

Abstract

Helios is the first (to our knowledge) modeling language for global optimization using interval analysis. Helios makes it possible to state global optimization problems almost as in scientific papers and textbooks and is guaranteed to find all isolated solutions in constraint-solving problems and all global optima in optimization problems. Helios statements are compiled to Newton, a constraint logic programming language using constraint satisfaction and interval analysis techniques and their efficiency is comparable to direct programming in Newton.

This paper presents the design of Helios, describes its theoretical foundation and semantic properties, sketches its implementation, reports some experimental results, and compares Helios to other modeling languages and direct programming in Newton.

1. Introduction

Many applications in science and engineering (e.g., chemistry, robotics, economics, mechanics) require to solve global optimization problems: i.e., 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 ensure termination and to guarantee correctness (i.e., finding all solutions or the global optimum). Note also that many challenging and important applications in this area involve problems with less than 20 variables.

There are a variety of global optimization methods and a recent survey may be found in [13]. Two techniques which are generally recommended for difficult, highly nonlinear, problems are continuation (e.g. [27, 35]) and interval methods [7–10, 12, 14–16, 17, 22, 28, 31]. Continuation methods work in the domain of complex numbers and are effective for system of constraints not involving more than 20 variables, since they follow as many computation paths as the degree of the constraint systems. Interval methods work directly with real numbers and recent results [33] have shown that they

* Corresponding author. E-mail: {ldm,pvh}@cs.brown.edu

PII S0304-3975(96)00190-9

are comparable in efficiency to continuation methods on their benchmarks. In addition, they have been applied to larger problems, since they are not inherently limited by the degree of the system.

Interval techniques are now in use in a variety of constraint logic programming (CLP) languages. The integration of interval methods in CLP originates from the pioneering work of Cleary [4] and BNR-Prolog [29]. It was further investigated in CLP(BNR) [2], where real, integer, and Boolean constraints were considered. More recently, the constraint programming language Newton [1, 34] showed that techniques from numerical analysis and artificial intelligence [19, 21] can be combined in a CLP language to support state-of-the-art algorithms [33].

CLP languages based on intervals offer a number of attractive features. The development time of interesting applications is generally small due to the availability of the constraint solver, the nondeterminism of the language which makes it easy to implement branch and prune algorithms, and the symbolic nature of the language which makes it easy to construct and preprocess the constraints. However, CLP languages can still be far from the original statements used by scientists and engineers to describe these applications. For instance, recursive predicates must be defined to generate the problem constraints and data structures must be built to represent sets and arrays of variables and constants. These mundane activities prevent CLP languages from being accessible to users who are not computer scientists or not familiar with logic programming.

This limitation can be addressed by building a modeling language on top of CLP languages. Modeling languages (e.g. GAMS [3], AMPL [6], LINDO) have been the topic of much investigation in the mathematical programming community in the last 15 years, since they are convenient front-ends for linear programming and sometimes nonlinear programming; see [5] for an early overview of these languages and their advantages. Modeling languages are attractive tools because they make it possible to write problem statements that are almost identical to mathematical descriptions of the applications. These high-level declarative statements are then converted into a description of a set of constraints that is used by some linear programming or nonlinear programming solver.

We take this approach in this paper and we describe the design and implementation of Helios, a modeling language for global optimization which serves as a front-end for the CLP language Newton. As a consequence, Helios enables users to state global optimization problems using traditional notations from textbooks and scientific papers. These statements are then translated into Newton programs which, when executed, solve the original Helios statements.

From a design standpoint, Helios was inspired by the modeling language AMPL [6] and it contains features such as ranges, sets, constants, functions, and aggregation operators which are also present in [6]. However, it differs from AMPL, and other modeling languages we are aware of, on a number of important points.

Perhaps the most important novelty in Helios is its sound semantic foundation. Traditionally, the semantic of modeling languages is presented informally by means of examples without reference to a constraint solver. Although this is probably appropriate for linear programming, it is not fully satisfactory for global optimization. If the

specification of the underlying solver is not given, the nature of application domain and the fact that nonlinear solvers have fundamentally different functionalities make it impossible for users to interpret the results. Indeed, some solvers may give incorrect results on constraint-solving examples or they may converge to a local minimum for a minimization problem or to a point that is not even optimal in any sense. In Helios, this limitation is addressed by providing a set of minimal requirements that any implementation should address. These requirements give a precise meaning to the results and some useful information on the modeling issues. In addition, we show that any implementation satisfying these requirements has some nice soundness and completeness properties. As a consequence, Helios becomes independent from its underlying solver, not only at the syntactic level, but also at the semantic level.

A second novelty, which is a consequence of our sound foundation, is the output of an Helios statement. Contrary to other modeling languages that we are aware of, the output of Helios is a set of solution-boxes which associates a small interval with each variable of the statement. These solution-boxes are guaranteed to contain all solutions (resp. global optima) for constraint-solving (resp. optimization) problems and, in many cases, Helios is also capable of proving the existence of solutions in these boxes. In contrast, AMPL returns values for the variables and does not provide any guarantee on nonlinear programs. Helios also contains new modeling concepts such as the notions of soft constraints and new environmental tools for monitoring the computation.

From an implementation standpoint, our current implementation takes the somewhat unusual step of generating Newton programs which, when executed, solve the original statements. Informally speaking, the resulting Newton program queries the user for some input values, builds data structures for constants, variables, functions, ranges, and sets, produces a set of primitive constraints, and solve these constraints using a branch and prune algorithm. This approach has the advantage of simplifying the implementation of Helios, while inducing only a negligible overhead over Newton. The reduction in development time comes from the support for memory management, nondeterminism, symbolic manipulation, and constraint solving in Helios. The small overhead comes from the fact that the compilation process is linear in the size of the statement, while the constraint generation step is linear in the size of the constraint system generated for reasonable statements (i.e., statements using all objects they declare).

The contributions of this paper can thus be summarized as follows:

1. It presents Helios, the first (to our knowledge) modeling language for global optimization using interval analysis. Altough Helios contains many features from existing modeling languages, it differs from them by its sound semantic foundation and its guarantees on the soundness and completeness of its results.

2. It indicates that Helios can be compiled into Newton programs which, when executed, exhibit a performance comparable to direct programming in Newton and thus to state-of-the-art interval and continuation methods.

The rest of this paper is organized as follows. Section 2 is a gentle and informal presentation of Helios. Section 3 describes the semantics of Helios formally and prove various soundness and completeness properties. Section 4 describes the

implementation of Helios and justifies formally why Helios induces a marginal overhead over Newton in general. Section 5 reports some experimental results of Helios. Section 6 discusses related work, while Section 7 concludes the paper.

2. A tour of Helios

This section presents a gentle introduction to Helios through a number of examples. It contains an informal description of the syntax, the functionality, and the performance of Helios on a number of well-known problems.

2.1. Getting started

As mentioned previously, Helios is a modeling language which enables nonlinear statements to be stated almost as in textbooks and scientific papers. Let us start by solving some conceptually easy nonlinear problems in Helios. Consider the problem of finding all roots of the function

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

in the interval $[0, 10^8]$. This problem can be modeled in Helios as follows:

```
Variable:
   x in [0..1e8];
Body:
   solve system all
      EQ: x^4 - 12 * x^3 + 47 * x^2 - 60 * x + 24 = 0;
```

The variable section declares a variable x whose range is $[0, 10^8]$. The body section contains the keywords solve system to specify that a system of constraints must be solved. The keyword all indicates that all solutions must be found. If only one solution is desired, the keyword one must be used instead. The next line specifies the constraint of the problem. The constraint is preceded by its name (in this case EQ) for reasons that will become clear later on.

The output of Helios for a nonlinear problem is a solution-box, i.e., the association of a small interval with each of the problem variables. Helios guarantees (modulo implementation errors) that all solutions are located in the solution-boxes. For instance, the execution of Helios on the above problem produces the following solution-boxes:

```
Solution: 1 [SAFE]
------------
x = [0.88830577907174 , 0.88830577907176]
```

```
Solution: 2 [SAFE]
------------
x = [0.99999999999999 , 1.00000000000001]
```

Helios thus guarantees that all roots of the function

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

are in the above intervals. The annotation SAFE means that there exists a solution in the given solution-box. Helios proves existence of solutions numerically. For some problems, this guarantee cannot be obtained, in which case the annotation POSSIBLE is returned. The compilation time for this example is 0.07 s and the execution time is 0.29 s.[1] When asked to find the roots of

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

Helios does not return any (SAFE or POSSIBLE) solution-boxes, which means that the function has no root in the given interval.

Consider now a multivariate problem wich consists of finding the intersection of a circle and a parabola as specified by the equations

$$x^2 + y^2 = 1,$$
$$x^2 = y.$$

The problem can be stated in Helios as follows:

```
Variable:
   x in [-1e8..1e8];
   y in [-1e8..1e8];
Body:
   solve system all
      Circle:    x^2 + y^2 = 1;
      Parabola:  x^2 = y;
```

The variable section declares variables x and y. The body section defines the two constraints. Helios returns the following solution-boxes for this problem:

```
Solution: 1 [SAFE]
------------
x = [-0.78615137775743 , -0.78615137775742]
y = [0.61803398874989 , 0.61803398874990]
```

[1] All computation times given in this paper are on the SUN-SPARC-10 running Solaris.

```
Solution: 2 [SAFE]
------------
x = [0.78615137775742 , 0.78615137775743]
y = [0.61803398874989 , 0.61803398874990]
```

The compilation time for this example is 0.07 s and the execution time is 0.32 s.

2.2. Array of variables

The same problem can be defined in terms of arrays. Assume that the problem consists of finding the solution to the two equations

$$x_1^2 + x_2^2 = 1,$$
$$x_1^2 = x_2.$$

The closest Helios statement is as follows:

```
Variable:
   x : array[1..2] in [-1e8..1e8];
Body:
   solve system all
      Circle:    x[1]^2 + x[2]^2 = 1;
      Parabola:  x[1]^2 = x[2];
```

The variable section declares an array of two variables which is then used in the body section. The output of Helios on this problem is as follows:

```
Solution: 1 [SAFE]
------------
x[1] = [-0.78615137775743 , -0.78615137775742]
x[2] = [0.61803398874989 , 0.61803398874990]

Solution: 2 [SAFE]
------------
x[1] = [0.78615137775742 , 0.78615137775743]
x[2] = [0.61803398874989 , 0.61803398874990]
```

The compilation time for this example is 0.7 s and the execution time is 0.32 s. The above statement uses a 1-dimensional array of variables. As we will see later on, Helios also supports multi-dimensional arrays.

2.3. Generic constraints

In nonlinear applications, it is frequent to encounter the same constraints applied to different sets of variables. For instance, kinematics applications often contain constraints

of the form

$$s_i^2 + c_i^2 = 1 \quad (1 \leqslant i \leqslant 6).$$

These constraints can be expressed in Helios in the following way:

```
Variable:
   s : array[1..6] in [-1e8..1e8];
   c : array[1..6] in [-1e8..1e8];
Body:
   solve system all
      trigo(i in [1..6]) : s[i]^2 + c[i]^2 = 1;
```

The variable section defines two arrays of variables s and c. The body section contains a single generic constraint which defines 6 constraints, one for each value of i in the integer range [1..6]. The first constraint is obtained by replacing i by 1, while the sixth constraint is obtained by replacing i by 6. Of course, a real kinematics application will contain additional constraints. For instance, Fig. 1 describes a kinematics application for a six-joint robot which has 16 solutions. The compilation time is about 0.7 s and the execution time to find all solutions is about 30 s.

2.4. Ranges

Ranges, i.e. finite sets of successive integers, are used in various places in an Helios statement to define arrays of constants and variables, to specify constraints, and to control aggregation operators. Explicit range declarations reduce the risk of errors and make it easier to modify and upgrade the statements. Ranges are declared in Helios in the range section. With this additional feature, the partial statement of the kinematics

```
Variable:
   s :  array[1..6] in [-1e8..1e8];
   c :  array[1..6] in [-1e8..1e8];

Body:  solve system all
   trigo(i in [1..6]) :  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;
```

Fig. 1. A robot kinematics application in Helios.

application becomes

```
Range:
   idx = [1..6];
Variable:
   s : array[idx] in [-1e8..1e8];
   c : array[idx] in [-1e8..1e8];
Body:
   solve system all
      trigo(i in idx) : s[i]^2 + c[i]^2 = 1;
```

The range section defines a range idx which stands for [1..6]. This range is used subsequently to define the two arrays of variables and the generic constraint in a concise way.

2.5. Input parameters

Some kinematics applications are based on models which are parametrized by the number of joints in the robots. The availability of arrays and generic constraints opens the possibility to define statements that are generic as well. To support these generic statements, Helios includes the concept of input parameters. Consider the Helios statement

```
Input:
   int N : "Number of Joints: ";
Range:
   idx = [1..N];
Variable:
   s : array[idx] in [-1..1];
   c : array[idx] in [-1..1];
Body:
   solve system all
      trigo(i in idx) : s[i]^2 + c[i]^2 = 1;
```

The execution of this statement queries users with the message *Number of Joints:* to obtain the value N which is then used to define the ranges, variables, and constraints of the problem.

2.6. Constants

Helios statements often contain integer and real numbers. It is good practice in general to isolate them in some specific part of the statement. This reduces the risk of errors and makes it easier to modify the statement subsequently. Helios supports

this practice through its constant section. Consider, for instance, the following problem from neurophysiology:

$$
\begin{aligned}
&x_1^2 + x_3^2 = 1,\\
&x_2^2 + x_4^2 = 1,\\
&x_5 x_3^3 + x_6 x_4^3 = 5.1,\\
&x_5 x_1^3 + x_6 x_2^3 = 4.3,\\
&x_5 x_1 x_3^2 + x_6 x_4^2 x_2 = 3.1,\\
&x_5 x_1^2 * x_3 + x_6 x_2^2 x_4 = 2.3.
\end{aligned}
$$

This is a possible Helios statement for this problem.

```
Constant:
   real C1 = 5.1;
   real C2 = 4.3;
   real C3 = 3.1;
   real C4 = 2.3;
Variable:
   x : array[1..6] in [-1e1..1e1];
Body:
   solve system all
      Cons1 : x[1]^2 + x[3]^2 = 1;
      Cons2 : x[2]^2 + x[4]^2 = 1;
      Cons3 : x[5] * x[3]^3 + x[6] * x[4]^3 = C1;
      Cons4 : x[5] * x[1]^3 + x[6] * x[2]^3 = C2;
      Cons5 : x[5] * x[1] * x[3]^2 + x[6] * x[4]^2 * x[2] = C3;
      Cons6 : x[5] * x[1]^2 * x[3] + x[6] * x[2]^2 * x[4] = C4;
```

The constant section describes four constants that are then used in the body section. The compilation time for this example is about 0.4 s and the execution time is about 0.4 s. Another way to express the same problem in Helios consists of using an array of constants.

```
Constant:
   real C = array[1..4] = [5.1,4.3,3.1,2.3] ;
Variable:
   x : array[1..6] in [-1e1..1e1];
Body:
   solve system all
      Cons1 : x[1]^2 + x[3]^2 = 1;
      Cons2 : x[2]^2 + x[4]^2 = 1;
      Cons3 : x[5] * x[3]^3 + x[6] * x[4]^3 = C[1];
```

```
Cons4 : x[5] * x[1]^3 + x[6] * x[2]^3 = C[2];
Cons5 : x[5] * x[1] * x[3]^2 + x[6] * x[4]^2 * x[2] = C[3];
Cons6 : x[5] * x[1]^2 * x[3] + x[6] * x[2]^2 * x[4] = C[4];
```

Here the constant section defines an array of four constants with values 5.1, 4.3, 3.1, and 2.3. Arrays of constants are accessed in the traditional way. Helios supports other ways to initialize arrays of constants which will be illustrated in the next example. Note also that Helios supports a number of predefined constants such as π and e. These will also be illustrated later in the paper.

2.7. Aggregation operators

Consider now a traditional problem from numerical analysis: the discretization of a nonlinear integral equation [24]. The objective is to find the zeros of the functions $f_k(x_1,\ldots,x_m)(1 \leqslant k \leqslant m)$ defined as follows:

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

with $t_j = jh$ and $h = 1/(m+1)$. To express this problem concisely, Helios has a notation to express sums and products. In particular, expressions of the form

```
sum(i in [1..n]) a[i]
```

are used in Helios as equivalents of the mathematical expressions

$$\sum_{i=1}^{n} a_i.$$

The Moré–Cosnard problem can then be expressed in Helios as depicted in Fig. 2. There are several novel features in this statement. The main novelty is the presence of the sum operators. These operators use a sum variable j which ranges over a set (e.g., [1..k]). Operators such as sum and product can be nested arbitrarily (although parenthesis are sometimes necessary to overrule the priority rules). The second novelty is the presence of the generic constant

```
Constant:
   real t[j in idx] = j * h;
```

which defines an array of constants t[1],...,t[m] and initializes it to h, 2*h,..., m*h. The performance of Helios on this example is given in Fig. 3. We separate the compilation time, the time to generate constraints, the solving time. Note that the time for generating the constraints is essentially linear in the size of the constraint system.

```
Input:
   int m :  "Number of variables:  ";
Range:
   idx = [1..m];
Constant:
   real h = 1/(m+1);
   real t[j in idx] = j * h;
Variable:
   x :  array[idx] in [-1e8..0];
Body:
   solve system all
      f(k in idx):
         0 =  x[k] + 0.5 * h * [
                 (1 - t[k]) * (Sum(j in [1..k]) t[j]*(x[j] + t[j] + 1)^3) +
                 t[k] * (Sum(j in [k+1..m]) (1-t[j])*(x[j] + t[j] + 1)^3)];
```

Fig. 2. The Moré–Cosnard Problem in Helios.

n	n^2	Compilation Time (ms)	Generation Time	Solving Time (ms)	Growth Factor
5	25	100	210	1310	
7	49	100	250	2700	2.06
10	100	100	250	7970	2.95
14	196	100	350	22180	2.78
20	400	100	530	73270	3.30
28	784	100	790	233180	3.18
40	1600	100	1450	864780	3.70

Fig. 3. Performance results of Helios on Moré–Cosnard Problem.

2.8. Functions

2.9. Unconstrained optimization

So far, we have only considered the solution of nonlinear systems of constraints. Helios also supports unconstrained optimization, i.e. the minimization or maximization of nonlinear functions. In Helios, an unconstrained optimization problem is transformed into a constraint-solving problem by enforcing some necessary conditions of the optima. In particular, the derivatives of the function with respect to all variables must be equal to zero and the cost of any candidate solution must not be worse than any previously found value.

The result returned by Helios for unconstrained optimization problems is a set of solution-boxes that contain all *global* optima in the initial range of the variables.[2] Helios also returns an interval enclosing the value of the global optima.

It is important to stress that Helios actually returns and bounds the global optima. It is a global search method, not a local search method. As a consequence, Helios

[2] Helios assumes that there exists at least one global solution in the initial range of the variables. The result is undefined otherwise.

guarantees that global optima cannot lie outside the solution-boxes returned. Consider the problem of minimizing the function

$$(1.5 - x_1 * (1 - x_2))^2 + (2.25 - x_1 * (1 - x_2^2))^2 + (2.625 - x_1 * (1 - x_2^3))^2$$

in the box $[-10, 10] \times [-10, 10]$. This problem is known as the `Beale` problem and it can be stated in `Helios` in the following way:

```
Range:
   idx = [1..2];
Variable:
   x : array[idx] in [-10..10];
Body:
   minimize
      (1.5-x[1]*( 1-x[2]))^2+(2.25-x[1]*( 1-x[2]^2 )) ^2+
      (2.625-x[1]*(1-x[2]^3))^2;
```

The output of `Helios` for this problem (which has a single global minimum) is as follows:

```
Global Bound
-------------
[-0.00000000000000 , 0.00000000000001]

Solution: 1 [SAFE]
------------
x[1] = [2.99999999985663 , 3.00000000014339]
x[2] = [0.49999999997889 , 0.50000000002111]
```

The interval depicted below `Global Bound` encloses the value of the global minimum. The rest of the display shows the solution-box. `Helios` also displays upper bounds as they are found in the implementation, although this is not shown in the above example.

Another interesting application from [18] is the 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 minimum. Fig. 4 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 Fig. 5.

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

Fig. 4. Unconstrained optimization in Helios: Problem Levy 5.

n	Compilation Time (s)	Generation Time (s)	Running Time (s)	Growth Factor
5	0.07	0.14	1.60	
10	0.07	0.16	4.43	2.76
20	0.07	0.22	14.81	3.34
40	0.07	0.21	54.67	3.69
80	0.07	0.33	216.49	3.95

Fig. 5. Performance results of Helios on Problem Levy 5.

2.10. Constrained optimization

Helios also supports the solving of constrained optimization problems, i.e., the minimization or maximization of a (nonlinear) function subject to a set of (nonlinear) constraints. Once again, the basic idea is to transform the optimization problem into a constraint-solving problem by imposing necessary conditions satisfied by all global optima. In particular, Helios applies the "so-called" Fritz–John conditions (a generalization of the Kuhn–Tucker necessary conditions for optimality) and the requirement that any candidate solution should not be worse than any previously found solution.

Helios is guaranteed to return all global optima and an interval enclosing the value of the objective function for these optima. It is thus a global search method as was already the case for unconstrained optimization:

$$\begin{aligned} &\text{minimize} && -x_1 \\ &\text{subject to} \\ &&& x_1^3 - x_2 + x_3^2 \geqslant 0 \\ &&& x_1^2 - x_2 - x_4^2 \geqslant 0 \end{aligned}$$

$$x_1^3 - x_2 + x_3^2 \leqslant 0.1$$
$$x_1^2 - x_2 - x_4^2 \leqslant 0.1$$
$$-4 \leqslant x_i \leqslant 4 \quad (1 \leqslant i \leqslant 4)$$

can be stated in Helios as

```
Variable:
   x : array[1..4] in [-4..4];
Body:
   minimize
      - x[1]
   subject to
      eq1: x[1]^3 - x[2] + x[3]^2 >= 0;
      eq2: x[1]^2 - x[2] - x[4]^2 >= 0;
      eq1: x[1]^3 - x[2] + x[3]^2 <= 0.1;
      eq2: x[1]^2 - x[2] - x[4]^2 <= 0.1;
```

Helios returns the solution

```
Solution: 1
------------
x[1] = [1.08495290355817 , 1.08495290359180]
x[2] = [1.17712280289355 , 1.17712280301227]
x[3] = [-0.00000000000001 , 0.00000000000621]
x[4] = [-0.00000000000001 , 0.00000000000001]
```

on this problem, isolating the global minimum. The compilation time is about 0.4 s, while the execution time is about 0.4 s.

2.11. *Soft constraints*

In many applications, some particular properties of the problem or of its solutions can be stated in terms of constraints. A typical example is the use of constraints to remove symmetries by imposing an ordering on some variables. These constraints could be added directly to the Helios statement. Unfortunately, they may sometimes interfere with the ability of Helios to prove existence of solutions. To remove this limitation, Helios supports the concept of *soft constraints*. Soft constraints behave in essentially the same way as standard constraints, except that they are ignored when proving existence of solutions. This distinction captures the special nature of these "redundant" constraints. Consider, for instance, the statement from neurophysiology that we have seen before.

```
Variable:
   x: array[1..6] in [-10..10];
Body:
   solve system all
      C1 : x[1]^2 + x[3]^2 = 1;
      C2 : x[2]^2 + x[4]^2 = 1;
      C3 : x[5] * x[3]^3 + x[6] * x[4]^3 = 5;
      C4 : x[5] * x[1]^3 + x[6] * x[2]^3 = 4;
      C5 : x[5] * x[1] * x[3]^2 + x[6] * x[4]^2 * x[2] = 3;
      C6: x[5] * x[1]^2 * x[3] + x[6] * x[2]^2 * x[4] = 2;
```

It is easy to see that, if $(v_1, \ldots, v_6)$ is a solution, so is $(v_2, v_1, v_4, v_3, v_6, v_5)$. As a consequence, adding the constraint `x[1] <= x[2]` will remove some symmetries in the problem, while not affecting our ability to find the solutions. The `Helios` statement implementing this idea is as follows:

```
Variable:
   x: array[1..6] in [-10..10];
Body:
   solve system all
      C1 : x[1]^2 + x[3]^2 = 1;
      C2 : x[2]^2 + x[4]^2 = 1;
      C3 : x[5] * x[3]^3 + x[6] * x[4]^3 = 5;
      C4 : x[5] * x[1]^3 + x[6] * x[2]^3 = 4;
      C5 : x[5] * x[1] * x[3]^2 + x[6] * x[4]^2 * x[2] = 3;
      C6: x[5] * x[1]^2 * x[3] + x[6] * x[2]^2 * x[4] = 2;
   with soft constraint
      R1 : x[1] <= x[2];
```

Symmetries also appear in optimization problems. Consider the following statement:

```
Range:
   idx = [1..2];
Variable:
   x : array[idx] in [-10..10];
Body:
   minimize
      Prod(k in idx) (Sum(i in [1..5]) i * cos((i+1)*x[k] + i));
```

Variables `x[1]` and `x[2]` play a symmetric role in this problem. As a consequence, it is possible to add the constraint `x[1] <= x[2]` without affecting our ability to find a

global minimum. The statement simply becomes:

```
Range:
   idx = [1..2];
Variable:
   x : array[idx] in [-10..10];
Body:
   minimize
      Prod(k in idx) (Sum(i in [1..5]) i * cos((i+1)*x[k] + i))
   with soft constraint
      R : x[1] <= x[2];
```

2.12. Pragmas

Helios also contains a number of pragmas which can be used to control the execution of the underlying constraint-solving algorithm. These pragmas are not fundamental to the efficiency of Helios in general and are almost never used in our benchmarks. The only two of them used in our benchmarks are the precision and the split pragmas.

The pragma precision controls the width of the solution-box. For instance, the statement

```
Pragma:
   precision = 1e-6;
Range:
   idx = [1..2];
Variable:
   x : array[idx] in [-10..10];
Body:
   minimize
      Prod(k in idx) (Sum(i in [1..5]) i * cos((i+1)*x[k] + i))
   with soft constraint
      R : x[1] <= x[2];
```

specifies that the solution-box should be smaller in width than 1e-6. The default value of this pragma is 1e-8. It is used when a certain width is requested by the problem statement or to overwrite the default to improve our ability to obtain proofs of existence.

The pragma split specifies the heuristics used to split a box in the branch and prune algorithm used in the implementation. The default is *round robin* which is almost always the best strategy for our set of benchmarks. The default can be overwritten by specifying *largest first* to select the largest box as the next box to split. In general,

this strategy is outperformed by *round robin*. Note that both strategies are standard in interval analysis.

2.13. Display

The display section describes the output of Helios. By default, Helios returns the list of solution-boxes and each solution-box associates an interval with each problem variable. In addition, in optimization problems, Helios displays an interval bounding the global optima. The display section enables to overwrite this default. In particular, it make it possible to specify

- which variables to display;
- which constraint to display;
- whether to display the value of the objective function on each solution-box in optimization problems.

By specifying a subset of the variables, it is possible to visualize only those variables of interest. This is valuable whenever the additional variables have been introduced to factorize some expressions in the hope of improving efficiency. For instance, the kinematics example presented previously could be written as

```
Range:
   idx = [1..12];
Variable:
   x : array[idx] in [-1e8..1e8];
   a : array[1..4] in [-1e8..1e8];
Body:
   solve system all
      trigo(i in [1..6]): x[2*i-1]^2 + x[2 * i]^2 = 1;
      t1: a[1] = x[4] + x[6] + x[8];
      t2: a[2] = x[3] + x[5] + x[7];
      t3: a[3] = x[4] + x[6] + x[8];
      t4: a[4] = 3 * x[4] + 2 * x[6] + x[8];
      c1: x[12] * a[1] - (x[10] * x[11] * a[2]) = 0.4077;
      c2: x[2] * x[9] * a[3] + x[1] * x[10] = 1.9115;
      c3: x[9] * a[2] = 1.9791;
      c4: x[2] * a[4] = 4.0616;
      c5: x[1] * a[4] = 1.7172;
      c6: 3 * x[3] + 2 * x[5] + x[7] = 3.9701;
```

This statement contains four new variables a[1],...,a[4] which are used in factorizing some expressions. Obviously, the intervals associated with them is only marginally relevant and the display section may indicate that only the variables x are of interest.

The display section expressing this information

```
Display:
   Variable: x;
```

can be inserted just after the body section.

Helios also supports the display of constraints for each solution. For a given solution-box, displaying a constraint consists of displaying the evaluation of its left- and right-sides over the solution-box and of computing the difference between these two intervals. The ability to display constraints is valuable for a number of reasons. On the one hand, the display may expose some numerical problems of the statement, e.g., the intervals resulting of the evaluation of each side may be large. This suggests that the problem is not numerically stable and that another modeling should be proposed. On the other hand, displaying constraints indicates which constraints are tight, which is of special interest in optimization problems. Consider the constrained optimization problem from [11] depicted in Fig. 6.

The output of Helios for this problem is as follows:

```
Global Bound
------------
[0.015619525242393,0.015619525242662]
```

```
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:
   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;
Display:
   constraint:  C1,C2,C3,C4;
```

Fig. 6. Constrained optimization in Helios.

```
Solution: 1
-----------
x1 = [0.00000000000000 , 0.00000000000001]
x2 = [0.00000000000000 , 0.00000000000001]
x3 = [0.00000000000000 , 0.00000000000001]
x4 = [0.00000000000000 , 0.00000000000001]
x5 = [0.00000000000000 , 0.00000000000001]
x6 = [0.00332330324306 , 0.00332330324318]

Constraints
-----------
c1 : Slack ->
   LHS = [4.96999999999998 , 4.97000000017102]
   RHS = [4.96999999999999 , 4.97000000000001]
   EPS = [-0.00000000000002 , 0.00000000017102]
c2 : Slack ->
   LHS = [4.60343965229019 , 4.60343965244861]
   RHS = [-1.88000000000001 , -1.87999999999999]
   EPS = [6.48343965229019 , 6.48343965244861]
c3 : Slack ->
   LHS = [-0.00000000000001 , 0.00000000000001]
   RHS = [-29.08000000000001 , -29.07999999999999]
   EPS = [29.07999999999999 , 29.08000000000001]
c4 : Slack ->
   LHS = [7.30462052825141 , 7.30462052850278]
   RHS = [-78.02000000000002 , -78.01999999999999]
   EPS = [85.32462052825140 , 85.32462052850280]
```

The display indicates that the first constraint is most probably tight while the remaining ones are not. Note also that there is no evidence that the problem is unstable numerically.

Finally, it may be useful in optimization problems to evaluate the objective function on each of the solution-boxes. Once again, this gives some indication of the precision of the solution and its numerical reliability. For instance, the unconstrained optimization `Helios` statement

---

```
Pragma:
   precision = 1e-6;
Range:
   idx = [1..2];
Variable:
   x : array[idx] in [-10..10];
```

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

produces the following display:

```
Global Bound:
-------------
[-186.730920708520956, -186.730896662254964]

Solution: 1
------------
x[1] = [0.00000000000000 , 0.00000000000000]
x[2] = [-1.42512843108909 , -1.42512842677609]
Objective Value: [-186.73092070852096 , -186.73089666225493]
```

Note also that Helios contains some vocabulary for describing how to monitor the execution. The facilities enable users to debug the correctness and the performance of their statements.

2.14. Scoping rules

To complete the description of Helios, it is necessary to specify the scoping rules of the language. These rules specify when an identifier is defined and which object it refers to. For instance, in a statement such as

```
Constant:
   int n = 3;
Variable:
   x : array[1..2] in [-10,10];
Body:
   solve system
      c: Sum(n in [1..2]) n + x[n] = 3;
      c: Prod(n in [1..2]) (n - x[n]) = 3;
```

it is important to specify which object the identifier n refers to in the expression n + x[n].

The scoping rules are given in three steps. First, the various scopes of Helios are described. Second, the visibility of each identifier is specified. Finally, the scope-resolution algorithm is described. Note that these scoping rules implement automatically our convention that an identifier must be defined before being used.

The various scopes appearing in an Helios statement are as follows:

- The outermost scope, called the global scope, which will eventually contains all variables, constants, ranges, sets, functions, and constraints opens at the beginning of the statement and terminates at the end of the statement.
- A constant, function, or set declaration opens a new scope which is closed at the next semi-colon.
- A set definition opens a new scope that is closed at the definition.
- An aggregation operator opens a new scope that is closed at the end of the expression it applies to.

The following rules specify when an identifier becomes visible in a given scope.

- Constant, range, set, variable, function, and constraint identifiers become visible in the global scope at the end of their declarations (i.e., the first occurrence of a semi-colon).
- Index variables in generic constants, generic sets, or functions become visible in the innermost scope at the next equality symbol.
- Index variables in an aggregation operator or in a constraint becomes visible in the innermost scope at the end of the signature containing them.

The scope-resolution algorithm, which associates an object (if any) with an identifier, implements the traditional hiding rule. The identifier is first looked up in the innermost scope. If it is not found, the algorithm is applied recursively in the parent scope. The algorithm terminates unsuccessfully if it is applied in the global scope and the identifier is not declared in this outermost scope.

3. The semantics of Helios

Traditionally, the semantics of modeling language is presented informally in terms of examples as in the previous section. This approach is probably appropriate for modeling languages which serves as front-ends for linear programming given the simplicity of the translation process. However, this approach is much less satisfactory for global optimization because of the nature of global optimization (e.g., a solution may not be representable exactly on a computer) and the fundamentally different functionalities of nonlinear solvers. Indeed, some solvers may produce incorrect results on some systems of constraints, may converge to a local minimum in a minimization problem or even to a point that is not optimal in any sense. As a consequence, the semantics of the modeling language strongly depends on the underlying constraint solver. Without a specification of the solver, users have no reasonable way to interpret the results or to use the system properly. As a consequence, these modeling languages achieve independence of the language and the solver only at a syntactic level.

In Helios, we recognize this potential problem and we describe the semantics of the language in a precise way. The semantics imposes certain requirements on the implementation, it provides users with a number of guarantees on the results of an Helios statement, and it helps them in formulating their statements. As a consequence, the semantics is a step towards independence both at the syntactic and semantic levels.

The rest of this section is organized as follows. Section 3.1 briefly discussed how to transform the Helios statement into a set of constraints. Section 3.2 describes some background on interval analysis. Sections 3.3–3.5 describe the semantics of constraint solving, unconstrained minimization, and constrained minimization, respectively.

3.1. Translation

The semantics of an Helios statement is given in two steps. In a first step, the Helios statement is transformed into a set of constraints. In a second step, the Helios statement is given a meaning in terms of these constraints. The translation of the Helios statement into a set of constraints is discussed in some detail in the implementation section and we will not formalize it in detail. It mainly consists of applying a number of rewriting rules of the form

- replace a constant by its defining body;
- replace a range by the set of elements it denotes;
- replace a generic object by a set of individual objects;
- unfold aggregate operators.

The output of the translation is a set of constraints (possibly with an objective function to minimize) written in a language whose abstract syntax is specified by the following grammar:

$$c \in \mathit{Constraint}$$

$$f \in \mathit{Function}$$

$$x_i \in \mathit{RealVariables} = \{x_1, \ldots, x_n\}$$

$$q \in Q \cup \{pi\}$$

$$n \in \mathcal{N}$$

$$c ::= f = 0 \mid f \geqslant 0$$

$$f ::= q \mid x_i \mid f + f \mid f - f \mid f \times f \mid f^n \mid (f) \mid \sin(f) \mid \cos(f) \mid \log(f) \mid \exp(f) \mid \mathrm{sqrt}(f)$$

Note that we assume for simplicity that all constraints are written using a finite (but arbitrary large) set of variables $\{x_1, \ldots, x_n\}$. The semantics of the language is given by the following semantic functions whose signatures are

$$\mathcal{S}_c : \mathit{Constraint} \to \Re^n \to \mathit{Bool}$$

$$\mathcal{S}_f : \mathit{Function} \to \Re^n \to \Re$$

and whose semantic equations are specified as follows:

$$\mathcal{S}_c[\![f = 0]\!] = \lambda t.\ \mathcal{S}_f[\![f]\!]t = 0$$

$$\mathcal{S}_c[\![f \geqslant 0]\!] = \lambda t.\ \mathcal{S}_f[\![f]\!]t \geqslant 0$$

$$\mathcal{S}_f[\![q]\!] = \lambda t.\ q$$

$$
\begin{aligned}
\mathcal{S}_f[\![pi]\!] &= \lambda t.\ \pi \\
\mathcal{S}_f[\![x_i]\!] &= \lambda t.\ t_{|i} \\
\mathcal{S}_f[\![f_1 + f_2]\!] &= \lambda t.\ \mathcal{S}_f[\![f_1]\!]t + \mathcal{S}_f[\![f_2]\!]t \\
\mathcal{S}_f[\![f_1 - f_2]\!] &= \lambda t.\ \mathcal{S}_f[\![f_1]\!]t - \mathcal{S}_f[\![f_2]\!]t \\
\mathcal{S}_f[\![f_1 \times f_2]\!] &= \lambda t.\ \mathcal{S}_f[\![f_1]\!]t \times \mathcal{S}_f[\![f_2]\!]t \\
\mathcal{S}_f[\![f^n]\!] &= \lambda t.\ (\mathcal{S}_f[\![f]\!]t)^n \\
\mathcal{S}_f[\![(f)]\!] &= \lambda t.\ \mathcal{S}_f[\![f]\!]t \\
\mathcal{S}_f[\![g(f)]\!] &= \lambda t.\ g(\mathcal{S}_f[\![f]\!]t) \text{ with } g \in \{\sin, \cos, \exp, \log, \text{sqrt}\}
\end{aligned}
$$

where, given a tuple $t = (r_1, \ldots, r_n)$, $t_{|i}$ denotes element r_i $(1 \leqslant i \leqslant n)$. In the following, we often abuse notation and use f (resp. c) to denote $\mathcal{S}_f[\![f]\!]$ (resp. $\mathcal{S}_c[\![c]\!]$) and vice versa. However, the meaning should be clear from the context.

It is useful at this point to emphasize that the way constraints are written may have a significant impact on the efficiency and accuracy of `Helios`. This is the main reason for formalizing the language in which constraints are written.

3.2. Interval arithmetic

We now turn to some basic notions of interval arithmetic that are necessary to define the semantics of `Helios`. We consider $\Re^\infty = \Re \cup \{-\infty, \infty\}$ the set of real numbers extended with the two infinity symbols and the natural extension of the relation $<$ to this set. We also consider a finite subset $\mathcal{F}$ of $\Re^\infty$ containing $-\infty, \infty, 0$. In practice, $\mathcal{F}$ corresponds to the floating-point numbers used in the implementation.

Definition 1 (*Interval*). An interval $[a, b]$ with $a, b \in \mathcal{F}$ is the set of real numbers

$$\{r \in \Re \mid a \leqslant r \leqslant b\}.$$

The set of intervals is denoted by $\mathcal{I}$ and is ordered by set inclusion.

Definition 2 (*Approximation*). Let S be a subset of $\Re$. The approximation of S, denoted by $\overline{S}$ or $\Box S$, is the smallest interval I such that $S \subseteq I$. We often write $\overline{r}$ instead of $\overline{\{r\}}$ for $r \in \Re$.

In the following, we denote real numbers by the letters r, v, $\mathcal{F}$-numbers by the letters a, b, l, m, u, intervals by the letter I, real functions by the letters f, g interval functions the letters F, G, relations over the reals by the letter c, and interval relations by the letter C, all possibly subscripted. We use a^+ (resp. a^-) to denote the smallest (resp. largest) $\mathcal{F}$-number strictly greater (resp. smaller) than the $\mathcal{F}$-number a. To capture outward rounding, we use $\lceil r \rceil$ (resp. $\lfloor r \rfloor$) to return the smallest (resp. largest) $\mathcal{F}$-number greater (resp. smaller) or equal to the real number r. We also use $\vec{I}$ to denote a box $\langle I_1, \ldots, I_n \rangle$ and $\vec{r}$ to denote a tuple $\langle r_1, \ldots, r_n \rangle$. Note that a tuple of n

intervals denotes a set of n-dimensional points. A canonical interval is an interval of the form $[l, l]$ or of the form $[l, l^+]$. A canonical box is a tuple of canonical intervals and we denote by $K(\vec{I})$ all the canonical boxes in $\vec{I}$. Finally, we use the following notations.

$$\mathit{left}([l,u]) = l$$

$$\mathit{right}([l,u]) = u$$

$$\mathit{center}([l,u]) = \lfloor (l+u)/2 \rfloor \text{ when } l \neq -\infty \text{ and } u \neq \infty.$$

The fundamental concept of interval arithmetic is the notion of interval extension.

Definition 3 (*Interval extension*). $F : \mathcal{I}^n \rightarrow \mathcal{I}$ is an interval extension of $f : \Re^n \rightarrow \Re$ if

$$\forall \vec{I} \in \mathcal{I}^n : \vec{r} \in \vec{I} \Rightarrow f(\vec{r}) \in F(\vec{I}).$$

An interval relation $C : \mathcal{I}^n \rightarrow \mathit{Bool}$ is an interval extension of a relation $c : \Re^n \rightarrow \mathit{Bool}$ if

$$\forall \vec{I} \in \mathcal{I}^n : [\exists \vec{r} \in \vec{I}\ c(\vec{r})] \ \Rightarrow\ C(\vec{I}).$$

Example 1. The interval function $\oplus$ defined as

$$[a_1, b_1] \oplus [a_2, b_2] = [\lfloor a_1 + a_2 \rfloor, \lceil b_1 + b_2 \rceil]$$

is an interval extension of addition of real numbers. The interval relation $\approx$ defined as

$$I_1 \approx I_2 \ \Leftrightarrow\ (I_1 \cap I_2 \neq \emptyset)$$

is an interval extension of the equality relation on real numbers.

It is important to stress that a real function (resp.) can be extended in many ways. For instance, the interval function $\oplus$ is the most precise interval extension of addition (i.e., it returns the smallest possible interval containing all real results) while a function always returning $[-\infty, \infty]$ would be the least accurate. It is useful to formalize this concept of optimality precisely.

Definition 4 (*Optimal interval extensions*). The optimal interval extension $\diamond f : \mathcal{I}^n \rightarrow \mathcal{I}$ of a function extension of $f : \Re^n \rightarrow \Re$ is defined as follows:

$$\diamond f(\vec{I}) = \square\{f(\vec{r}) \mid \vec{r} \in \vec{I}\}.$$

The optimal interval relation $\diamond c : \mathcal{I}^n \rightarrow \mathit{Bool}$ of a relation $c : \Re^n \rightarrow \mathit{Bool}$ is defined as follows:

$$\diamond c \ = \ \{\vec{I} \ \mid \ \vec{I} \cap c \neq \emptyset\}.$$

The above definitions can be generalized to partial functions (e.g. [28]).

Definition 5 (*Interval extension of partial functions*). $F : \mathcal{I}^n \to \mathcal{I}$ is an interval extension of $f : D \subseteq \Re^n \to \Re$ if

$$(1)\ \forall \vec{I} \in D: \quad \vec{r} \in \vec{I} \Rightarrow f(\vec{r}) \in F(\vec{I}),$$

$$(2)\ \forall \vec{I} \notin D: \quad F(\vec{I}) = [-\infty, \infty].$$

An interval relation $C : \mathcal{I}^n \to \mathit{Bool}$ is an interval extension of a relation $c : D \subseteq \Re^n \to \mathit{Bool}$ if

$$\forall \vec{I} \in D: \quad [\exists \vec{r} \in \vec{I}\ c(\vec{r})] \ \Rightarrow\ C(\vec{I}),$$

$$\forall \vec{I} \notin D: \quad C(\vec{I}).$$

In the following, we assume fixed interval extensions for the basic real operators (for instance, the interval extension of $+$ is defined by $\oplus$) and the basic real relations $=, \geqslant$. In addition, we overload the real symbols and use them for their interval extensions. We also assume that interval functions and interval relations are defined using a similar abstract syntax and semantics except that real variables (i.e., x_i) are replaced by interval variables (i.e., X_i) and rational numbers by intervals. We use *CONSTRAINT* and *FUNCTION* to denote the set of interval constraints and functions.

Given the interval extensions for the basic operators, it is easy to obtain interval extensions for the constraints of the language. The simplest extension of a function (resp. of a constraint) is its natural interval extension. Informally speaking, it consists of replacing each number by the smallest interval enclosing it, each real variable by an interval variable, each real operation by its fixed interval extension and each real relation by its fixed interval extension. In the following, if f (resp. c) is a real function (resp. constraint), we denote by $\widehat{f}$ (resp. $\widehat{c}$) its natural extension.

Example 2 (*Natural interval extension*). The natural interval extension of the function $x_1(x_2 + x_3)$ is the interval function $X_1(X_2 + X_3)$. The natural interval extension of the constraint $x_1(x_2 + x_3) = 0$ is the interval constraint $X_1(X_2 + X_3) \approx \overline{0}$.

The advantage of this extension is that it preserves the way constraints are written and hence users of the system can choose constraint representations particularly appropriate for the problem at hand. Formally, it can be defined as follows.

Definition 6 (*Natural interval extension*). The natural interval extension of a constraint c, denoted by $\widehat{c}$ and of signature $\widehat{\bullet} : \mathit{Constraint} \to \mathit{CONSTRAINT}$ is defined inductively as follows:

$$\widehat{[\![f = 0]\!]} = \widehat{[\![f]\!]} = \overline{0}$$

$$\widehat{[\![f \geqslant 0]\!]} = \widehat{[\![f]\!]} \geqslant \overline{0}$$

$$\widehat{[\![q]\!]} = \overline{q}$$

$$\begin{aligned}
\widehat{[\![x_i]\!]} &= X_i \\
[\![f_1 + f_2]\!] &= \widehat{[\![f_1]\!]} + \widehat{[\![f_2]\!]} \\
[\![f_1 - f_2]\!] &= \widehat{[\![f_1]\!]} - \widehat{[\![f_2]\!]} \\
[\![f_1 \times f_2]\!] &= \widehat{[\![f_1]\!]} \times \widehat{[\![f_2]\!]} \\
\widehat{[\![f^n]\!]} &= (\widehat{[\![f]\!]})^n \\
[\![g(f)]\!] &= g(\widehat{[\![f]\!]}) \\
[\![(f)]\!] &= \widehat{[\![f]\!]}
\end{aligned}$$

Proposition 1. *The natural extension of a function f (resp. a constraint c) defined in our abstract language is an interval extension of f (resp. c).*

Proof. The proof is by induction on the structure of the function. The result holds for the basic cases (i.e., variables and numbers). Consider now an expression of the form $f_1 + f_2$. By induction,

$$\forall \vec{I}: \quad \vec{r} \in \vec{I} \Rightarrow f_1(\vec{r}) \in \widehat{f_1}(\vec{I}),$$

$$\forall \vec{I}: \quad \vec{r} \in \vec{I} \Rightarrow f_2(\vec{r}) \in \widehat{f_2}(\vec{I})$$

Since interval addition is an interval extension of real addition, we have that

$$f_1(\vec{r}) + f_2(\vec{r}) \in \widehat{f_1}(\vec{I}) + \widehat{f_2}(\vec{I})$$

and the result follows. The other cases are similar. □

Other interval extensions (e.g., centered forms) exist and have been studied extensively (e.g. [28]).

3.3. Semantics of constraint solving

In this section, we define the semantics of constraint solving in `Helios`. The semantics imposes a requirement on the output of `Helios` to be satisfied by any implementation but it leaves considerable latitude on how to enforce this requirement. The semantics is defined in terms of two notions. Recall that we assume that all constraints are defined over variables $x_1, \ldots, x_n$.

Definition 7 (*Interval solution and approximation*). A canonical box $\vec{I}$ is an interval solution to a system of constraint $\mathscr{S}$ if $\forall c \in \mathscr{S} : \diamond c(\vec{I})$. A canonical box $\vec{I}$ is an interval approximation to a system of constraint $\mathscr{S}$ if $\forall c \in \mathscr{S} : \widehat{c}(\vec{I})$. In addition, we define

$$S(\mathscr{S}, \vec{I_0}) = \{\vec{I} \subseteq \vec{I_0} \mid \vec{I} \text{ is an interval solution of } \mathscr{S}\},$$

$$A(\mathscr{S}, \vec{I_0}) = \{\vec{I} \subseteq \vec{I_0} \mid \vec{I} \text{ is an interval approximation of } \mathscr{S}\}.$$

Proposition 2. *Let $\mathscr{S}$ be a system of constraint and $\vec{I}_0$ be a range. If $\vec{r}$ is a solution to $\mathscr{S}$ in $\vec{I}_0$, then there exists a box $\vec{I}$ in $S(\mathscr{S},\vec{I}_0)$ (resp. in $A(\mathscr{S},\vec{I}_0)$) such that $\vec{r} \in \vec{I}$.*

Proof. If $\vec{r}$ is a solution to $\mathscr{S}$ in $\vec{I}_0$, then there exists a canonical box $\vec{I}$ such that $\vec{r} \in \vec{I} \subseteq \vec{I}_0$. Since $\vec{r}$ is a solution, $c(\vec{r})$ holds for all $c \in \mathscr{S}$ and hence $\diamond c(\vec{I})$ and $\widehat{c}(\vec{I})$ hold by definition of interval extensions. □

Interval solutions are the best boxes that we can hope for but it may be impossible to obtain them in general due to numerical errors. Specifying an upper bound is also important, since otherwise an Helios implementation could simply return all canonical boxes. Interval approximations are particularly appropriate as a basis for the upper bound, since they are easy to obtain and since they respect the way constraints are written. In a sense, interval approximations provides a quality assurance that is controlled by users. We are now in position to define the semantics of constraint solving in Helios. The key idea is that Helios returns a set of canonical boxes which contains at least all interval solutions and at most all interval approximations.

Specification 1 (*Semantics of* Helios *for constraint solving*). Given a set $\mathscr{S}$ of constraints and an initial range $\vec{I}_0$ for the variables, Helios returns a set $H(\mathscr{S},\vec{I}_0)$ of canonical boxes satisfying

$$S(\mathscr{S},\vec{I}_0) \subseteq H(\mathscr{S},\vec{I}_0) \subseteq A(\mathscr{S},\vec{I}_0).$$

Note that it is not difficult to design a naive implementation of Helios: simply enumerate all the canonical intervals and select those which are interval approximations. The algorithm is guaranteed to terminate whenever the floating-point system is finite. Reasonable implementations may be based on implicit enumeration and may use a variety of pruning techniques. For instance, traditional interval methods uses the interval Newton method as a pruning technique during implicit enumeration, while Newton uses a consistency condition called box-consistency. The following theorem indicates that Helios cannot miss any solution.

Theorem 1 (Completeness of Helios). *Let $\mathscr{S}$ be a set of constraints and let $\vec{I}_0$ be the initial range for the variables. If $\vec{r}$ is a solution to $\mathscr{S}$ in $\vec{I}_0$, then there exists a box $\vec{I}$ in $H(\mathscr{S},\vec{I}_0)$ such that $\vec{r} \in \vec{I}$.*

Proof. Direct consequence of Proposition 2. □

3.4. Semantics of unconstrained optimization

We now turn to the semantics of unconstrained minimization in Helios. Unconstrained maximization is defined in a similar way. The key notion in this context is the concept of interval minimum.

Definition 8 (*Interval minimum*). Let f be a function, F be an interval extension of f, and $\vec{I}_0$ be an initial range for the variables. The interval minimum value of f for F in $\vec{I}_0$, denoted by $min(f,F,\vec{I}_0)$, is an interval $[l,u]$ satisfying

$$l = \min_{\vec{I} \in K(\vec{I}_0)} left(F(\vec{I})),$$
$$u = \min_{\vec{I} \in K(\vec{I}_0)} right(F(\vec{I})).$$

An interval minimum of f for F in $\vec{I}_0$ is a canonical box $\vec{I} \subseteq \vec{I}_0$ such that

$$F(\vec{I}) \cap min(f,F,\vec{I}_0) \neq \emptyset.$$

The set of interval minima of f for F in $\vec{I}_0$ is denoted by $Smin(f,F,\vec{I}_0)$.

The following proposition is the basis of the soundness and completeness result for `Helios`.

Proposition 3. *Let f be a function, F be an interval extension of f, and $\vec{I}_0$ be an initial range*:

1. *The value $f^* = min\{f(\vec{r}) \mid \vec{r} \in \vec{I}_0\}$ belongs to $min(f,F,\vec{I}_0)$.*
2. *If $\vec{r}$ is a global minimum, there exists an interval minimum $\vec{I}$ such that $\vec{r} \in \vec{I}$.*

Proof. (1) Let $\vec{r}$ be a global minimum in $\vec{I}_0$. There exists a canonical box $\vec{I}$ such that $\vec{r} \in \vec{I} \subseteq \vec{I}_0$. By definition of interval extensions, $left(F(\vec{I})) \leqslant f^*$. In addition, for each canonical box $\vec{I} \subseteq \vec{I}_0$, either there exists a tuple $\vec{v}$ such that $f^* \leqslant f(\vec{v})$ or f is undefined on $\vec{I}$. In both cases, $f^* \leqslant right(F(\vec{I}))$ by definition of interval extensions. Hence $f^* \in min(f,F,\vec{I}_0)$ by definition of *min*.

(2) Let $\vec{r}$ be a global minimum in $\vec{I}_0$. $f(\vec{r}) = f^*$ and there exists a canonical box $\vec{I}$ such that $\vec{r} \in \vec{I} \subseteq \vec{I}_0$. $f^* \in F(\vec{I})$ by definition of interval extensions and $f^* \in min(f,F,\vec{I}_0)$ by part (1) of this proof. It follows that $\vec{I}$ is an interval minimum. □

We are now ready to specify the semantics of `Helios` and its soundness and completeness results.

Specification 2. Let f be a function and $\vec{I}_0$ be an initial range that contains at least one global minimum of f. `Helios` bounds the value of the global minima with an interval $min(f,\vec{I}_0)$ satisfying

$$min(f,\diamond f,\vec{I}_0) \subseteq min(f,\vec{I}_0) \subseteq min(f,\widehat{f},\vec{I}_0)$$

and encloses the global minima with a set of canonical boxes $Smin(f,\vec{I}_0)$ satisfying

$$Smin(f,\diamond f,\vec{I}_0) \subseteq Smin(f,\vec{I}_0) \subseteq Smin(f,\widehat{f},\vec{I}_0).$$

Theorem 2 (Soundness and completeness of `Helios`). *Let f be a function and $\vec{I}_0$ be an initial range that contains at least one global minimum of f*:

1. *The value* $f^* = min\{f(\vec{r}) \mid \vec{r} \in \vec{I_0}\}$ *belongs to* $min(f, \vec{I_0})$.
2. *If* $\vec{r}$ *is a global minimum, there exists a box* $\vec{I} \in Smin(f, \vec{I_0})$ *such that* $\vec{r} \in \vec{I}$.

Proof. Direct consequence of Proposition 3. □

Note that the requirement that the initial range contains at least one global minima of the function (without range constraints) is never used. It is however useful to make this assumption to obtain faster implementations of `Helios`.

3.5. Semantics of constrained optimization

We now turn to the semantics of constrained minimization in `Helios`. This semantics is a combination of the semantics for constraint solving and unconstrained optimization. In addition, it requires the following new notion.

Definition 9 (*Safe box*). Let $\mathcal{S}$ be a set of constraints. A safe box for $\mathcal{S}$ is a box containing at least one real solution of $\mathcal{S}$.

There are a variety of techniques for finding safe boxes numerically (see, for instance, [28] for an overview of some of them). In the following, we assume that a set of safe boxes is given to the system.[3]

Definition 10 (*Interval minimum*). Let f be a function, $\mathcal{S}$ be a system of constraints, $\vec{I_0}$ be a range, and B be a set of safe boxes for $\mathcal{S}$ in $\vec{I_0}$ and let P be the set of interval solutions (resp. approximations) of $\mathcal{S}$ in $\vec{I_0}$. The interval minimum value (resp. approximation) of f subject to $\mathcal{S}$ in $\vec{I_0}$ for B is an interval $[l,u]$, denoted by $\diamond min(f, \mathcal{S}, \vec{I_0}, B)$ (resp. $\widehat{min}(f, \mathcal{S}, \vec{I_0}, B)$) such that

$$l = \min_{\vec{I} \in P} left(\diamond f(\vec{I})) \quad (\text{resp. } l = \min_{\vec{I} \in P} left(\widehat{f}(\vec{I})))$$
$$u = \min_{\vec{I} \in S} right(\diamond f(\vec{I})) \quad (\text{resp. } u = \min_{\vec{I} \in S} right(\widehat{f}(\vec{I})))$$

An interval minimum (resp. min-approximation) of f subject to $\mathcal{S}$ in $\vec{I_0}$ for B is a canonical box $\vec{I} \in \vec{P}$ such that

$$\diamond f(\vec{I}) \cap \diamond min(f, \mathcal{S}, \vec{I_0}, B) \neq \emptyset. \quad (\text{resp. } \widehat{f}(\vec{I}) \cap \widehat{min}(f, \mathcal{S}, \vec{I_0}, B) \neq \emptyset).$$

We also denote by $\diamond Smin(f, F, \vec{I_0})$ (resp. $\widehat{Smin}(f, F, \vec{I_0})$) the set of interval minima (resp. min-approximations).

Proposition 4. *Let* f *be a function,* $\mathcal{S}$ *be a system of constraints,* $\vec{I_0}$ *be a range,* B *be a set of safe boxes for* $\mathcal{S}$ *in* $\vec{I_0}$, *and let*

$$f^* = min\{f(\vec{r}) \mid \vec{r} \in \vec{I_0} \ \& \ \vec{r} \text{ is a solution to } \mathcal{S}\}.$$

[3] In practice, these boxes are discovered during execution of the algorithm.

We have:

1. $f^* \in \diamond min(f, \mathcal{S}, \vec{I}_0, B)$;
2. *If* $\vec{r}$ *is a global minimum in* $\vec{I}_0$, *there exists a box* $\vec{I} \in \diamond Smin(f, \mathcal{S}, \vec{I}_0, B)$ *such that* $\vec{r} \in \vec{I}$.
3. $f^* \in \widehat{min}(f, \mathcal{S}, \vec{I}_0, B)$;
4. *If* $\vec{r}$ *is a global minimum in* $\vec{I}_0$, *there exists a box* $\vec{I} \in \widehat{Smin}(f, \mathcal{S}, \vec{I}_0, B)$ *such that* $\vec{r} \in \vec{I}$.

Proof. We only prove points 1 and 2. The proofs for points 3 and 4 are similar.

1. Let $\vec{r}$ be a global minimum in $\vec{I}_0$. By Proposition 2, there exists an interval solution $\vec{I}$ of $\mathcal{S}$ such that $\vec{r} \in \vec{I} \subseteq \vec{I}_0$. By definition of interval extensions, $left(\diamond f(\vec{I})) \leqslant f^*$. In addition, for each safe box $\vec{I} \in B$, there exists a solution $\vec{v}$ of $\mathcal{S}$ by definition of safe boxes and $f^* \leqslant f(\vec{v})$ since f^* is the global minimum value. By definition of interval extensions, $f^* \leqslant right(\diamond f(\vec{I}))$ by definition of interval extensions. Hence $f^* \in \diamond min(f, \mathcal{S}, \vec{I}_0, B)$.

2. Let $\vec{r}$ be a global minimum in $\vec{I}_0$. $f(\vec{r}) = f^*$ and there exists an interval solution $\vec{I}$ of $\mathcal{S}$ such that $\vec{r} \in \vec{I} \subseteq \vec{I}_0$. $f^* \in \diamond f(\vec{I})$ by definition of interval extensions and $f^* \in \diamond min(f, \mathcal{S}, \vec{I}_0, B)$ by part 1 of this proof. It follows that $\vec{I}$ is an interval minimum. □

We are in a position to give the specification of `Helios` for constrained minimization and to prove its soundness and completeness.

Specification 3. Let f be a function, $\mathcal{S}$ be a system of constraints, $\vec{I}_0$ be a range, and B be a set of safe boxes for $\mathcal{S}$ in $\vec{I}_0$. `Helios` bounds the value of the global minima with an interval $min(f, \mathcal{S}, \vec{I}_0, B)$ satisfying

$$\diamond min(f, \mathcal{S}, \vec{I}_0, B) \subseteq min(f, \mathcal{S}, \vec{I}_0, B) \subseteq \widehat{min}(f, \hat{f}, \vec{I}_0)$$

and encloses the global minima with a set of canonical boxes $Smin(f, \mathcal{S}, \vec{I}_0, B)$ satisfying

$$\diamond Smin(f, \mathcal{S}, \vec{I}_0, B) \subseteq Smin(f, \mathcal{S}, \vec{I}_0, B) \subseteq \widehat{Smin}(f, \hat{f}, \vec{I}_0, B).$$

Theorem 3 (Soundness and completeness of `Helios`). *Let* f *be a function,* $\mathcal{S}$ *be a system of constraints,* $\vec{I}_0$ *be a range,* B *be a set of safe boxes for* $\mathcal{S}$ *in* $\vec{I}_0$, *and let*

$$f^* = min\{f(\vec{r}) \mid \vec{r} \in \vec{I}_0 \ \& \ \vec{r} \text{ is a solution to } \mathcal{S}\}.$$

1. $f^* \in min(f, \mathcal{S}, \vec{I}_0, B)$.
2. *If* $\vec{r}$ *is a global minimum, there exists a box* $\vec{I} \in Smin(f, \mathcal{S}, \vec{I}_0, B)$ *such that* $\vec{r} \in \vec{I}$.

Proof. Direct consequence of Proposition 4. □

3.6. Pragmatics

The above description assumes that results must be of maximal precision. In practice, it may not be necessary to reach this accuracy and, as mentioned previously, Helios allows users to specify the size of the solution-boxes. In this case, the semantics remains the same, except that canonical boxes becomes boxes of the required precision. Another useful convention is also to allow Helios to return safe boxes whenever they are found, even if the required precision is not obtained. It is easy to generalize the semantics to include this functionality.

4. The implementation of Helios

The purpose of this section is to sketch the implementation of Helios using Newton, a constraint logic programming language over nonlinear constraints. The key idea behind the implementation is to generate a Newton program which, when executed, returns the results of Helios statement.

A complete description of Newton is available in [34]. 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)` and `minimize(Function,Constraints)`.[4] The implementation of these predicates is based on a branch and prune algorithm described in [33] which is an implementation of the semantics of Helios using the concept of box-consistency to prune the search space.

Because of the functionalities of Newton, the main task of the implementation is the translation described in Section 3.1. This translation is performed in two steps:

1. The Helios statement is parsed and a Newton program is generated.
2. The Newton program is executed, it generates a set of constraints, and solves them using one of the predefined predicates.

This approach to the implementation of Newton has a number of benefits. First, the symbolic nature of constraint logic programming makes the generation of constraints reasonably easy, since memory management issues are abstracted away. Second, the support of Newton for constraint solving and nondeterminism simplifies the implementation of the solver substantially. Finally, the overall implementation is efficient since the generation of constraints is proportional to the size of the constraint system generated (for reasonable Helios statements).

The rest of this section focuses some of the code generation aspects, since it is somewhat non-standard to generate code in such a high-level language. More details on the implementation can be found in the technical report version of this paper.

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

4.1. Overview of code generation

Fig. 7 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) correspond to the code generated for the various Helios sections: the input, constant, variable, function, and body sections. The rest of this section studies some of these goals in some 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 Fig. 7 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`.

4.2. 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.

```
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).
```

Fig. 7. Code generation: The top-level predicate.

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) :-
        input0(Global).
input0(Global) :-
        Global = environment(Constant,Function,Variable,Scope),
        heliosReadInteger('Number of rows:',Local),
        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.

4.3. The constant section

The compilation of the constant section follows the same basic idea as the code for the input section. 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 generate

`Constant`$_j$ for individual constant is as follows:

```
generateConstantj(Global,Result) :-
        generateExpressionk(Global,Local),
        evalReal(Local,Result).
```

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`. The use of a symbolic language simplifies this code generation, since the `Newton` implementation handles all the memory management issues.

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. Fig. 8 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]];
```

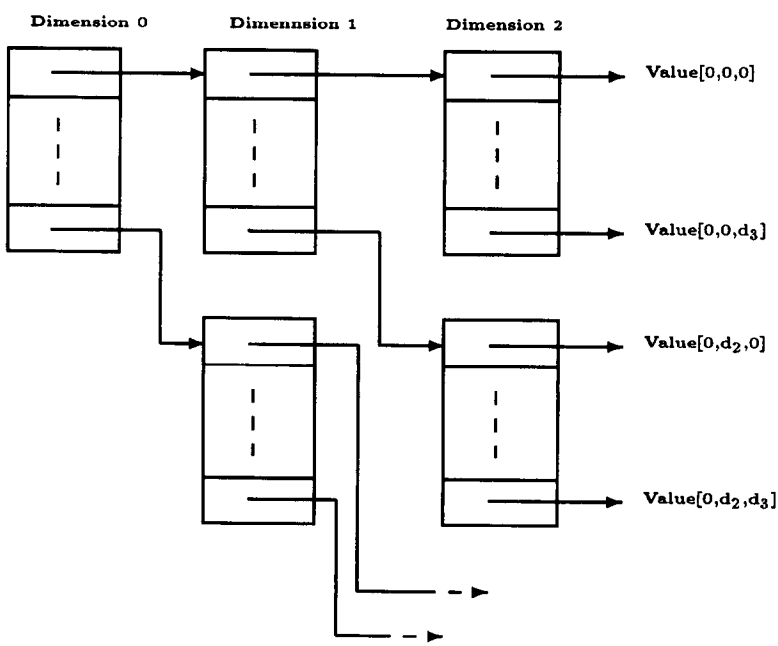

Fig. 8. Constant array: An example of a 3-dimensional array.

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:

```
generateConstanti(Global,Result) :-
        collectj(Global,Dim),
        libBuildArray(Result,Dim),
        fillArrayk(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. 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.

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 $\texttt{fillArray}_2$.

4.4. Variable section

The code generation for variables 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).

4.5. 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 Fig. 9.

4.6. 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:

```
generateConstraints(Global,Result) :-
        collect_i(Global,Result).
collect_i(Global,Result) :-
        Global = environment(Constant,Function,Variable,Scope),
```

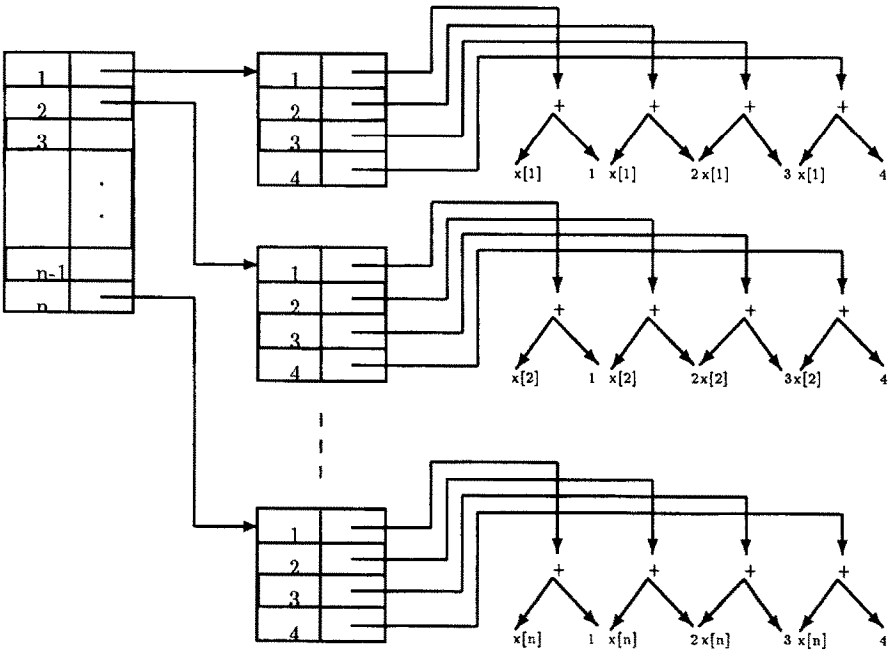

Fig. 9. Function: The entry in array `function` for function `foo`.

```
        genThisConstraint_k(Global,Result,Tail),
        collect_{i+1}(Global,Tail).
        ⋮
collect_{i+n}(Global,[]).
```

It remains to describe 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. The code generated obviously depends on the kind of objects encountered.

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.

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`.

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.

4.7. Complexity results

We now justify more formally why the implementation of `Helios` is efficient and, in fact, comparable in efficiency with a `Newton` program with the same functionality. The first result indicates that the generation of the `Newton` program is linear in the size of the `Helios` statement.

Theorem 4. *The running time of the* `Helios` *compiler is linear in the size of the* `Helios` *statement.*

Proof. Direct consequence of the fact that the compiler produces a constant number of instructions per node of the syntax tree. □

The second result concerns the generation of the constraints. An `Helios` statement denotes a set of constraints in our abstract language (and possibly an evaluation function). Informally speaking, the constraint generation part of the `Newton` program runs in time linear in the size of the generated constraints. However, the formal result should account for the fact that the statement can define generic objects that are never used

in the constraints, sets which filter all their elements, and other singularities of this nature.

Theorem 5. *Let $\mathscr{H}$ be an* Helios *statement without input constants, I be the size of $\mathscr{H}$, R be the (integer) range of maximal size in $\mathscr{H}$, and N be the maximum number of range nestings in $\mathscr{H}$. The running time of constraint generation in the* Newton *program is* $O(IR^N)$.

Proof. The proof is by inspection of the various code patterns. □

These two results indicate that the Helios implementation is essentially comparable in efficiency with a program directly written in Newton, since this last program should generate the same set of constraints. This explains the small overhead of Helios compared to Newton.

5. Experimental results

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

5.1. 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 [24], interval analysis [10, 12, 23], and continuation methods [35, 27, 26, 20]. Complete details on the benchmarks can be found in [33, 34]. 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.[5] Finally, we compare Helios with a state-of-the-art continuation method [35], 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 (v), the total degree of the system (d),[6] the initial range for the variables, the times in seconds to compile the Helios statement (Helios (C)), to generate the constraints (Helios (G)), and to solve them in Helios (Helios (S)), as well as

[5] Some interval methods such as [9] 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.

[6] The degree of the system is the product of the degree of each equation which is the highest degree of product terms.

Table 1
Summary of the experimental results on equation solving

Benchmarks	v	d	Range	Helios (C)	Helios (G)	Helios(S)	HRB	CONT
Broyden	10	3^{10}	$[-1,1]$	0.08	0.26	1.75	18.23	
Broyden	20	3^{20}	$[-1,1]$	0.08	0.32	5.20	?	
Broyden	160	3^{160}	$[-1,1]$	0.11	1.34	93.70	?	
Broyden	160	3^{160}	$[-10^8,10^8]$	0.11	1.34	103.53	?	
Moré-Cosnard	20	3^{20}	$[-4,5]$	0.11	0.34	11.47	968.25	
Moré-Cosnard	40	3^{40}	$[-4,5]$	0.11	0.50	71.53	?	
Moré-Cosnard	40	3^{40}	$[-10^8,0]$	0.11	0.44	71.15	?	
i1	10	3^{10}	$[-2,2]$	0.14	0.08	0.22	14.28	
i2	20	3^{20}	$[-1,2]$	0.20	0.06	0.49	1821.23	
i5	10	11^{10}	$[-1,1]$	0.17	0.07	0.25	33.58	
kin1	12	4608	$[-10^8,10^8]$	0.11	0.12	24.11	1630.08	
kin2	8	256	$[-10^8,10^8]$	0.15	0.12	808.98	4730.34	35.61
eco	4	18	$[-10^8,10^8]$	0.09	0.13	0.40	2.44	1.13
eco	5	54	$[-10^8,10^8]$	0.07	0.31	2.14	29.88	5.87
eco	6	162	$[-10^8,10^8]$	0.08	0.17	10.50	?	50.18
eco	7	486	$[-10^8,10^8]$	0.08	0.25	85.32	?	991.45
eco	8	1458	$[-10^8,10^8]$	0.08	0.21	697.64	?	
eco	9	4374	$[-10^8,10^8]$	0.08	0.30	6309.49	?	
combustion	10	96	$[-10^8,10^8]$	0.10	0.09	12.83	?	57.40
chemistry	5	108	$[0,10^8]$	0.12	0.04	5.71	?	56.55
neuro	6	1024	$[-10,10]$	0.09	0.25	0.66	28.84	5.02
neuro	6	1024	$[-1000,1000]$	0.10	0.13	13.92	?	5.02

the solving times for the other two methods. 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 h).

It is interesting to see that the compilation and generation times are almost negligible, showing that the cost of Helios over direct programming is really minimal on these benchmarks. Note that Helios solves Broyden, Moré-Cosnard, and interval benchmarks i1, i2, i5 without backtracking (contrary to most interval methods we know of).

5.2. *Unconstrained optimization*

Table 2 describes the results of Helios on unconstrained optimization. The benchmarks were taken mainly from [18, 25, 30, 32] and, for each of them, we give the number of variables, the range of the variables, the time for compilation, generation, and solving as well as the number of splits. Full details on the benchmarks can be found in [34]. The experimental results once again exhibit a number of interesting facts. First, compilation and generation times are in general negligible. Second, Helios is able to solve problems such as Levy5 and Levy6 in essentially linear time in the number of variables. Helios solves the problems Ratz25, Ratz27, and

Table 2
Summary of the experimental results on unconstrained optimization

Benchmarks	v	Range	Compilation	Generation	Solving	Splits
Hump	2	$[-10^7, 10^8]$	0.07	0.04	0.65	3
Levy1	1	$[-10^7, 10^7]$	0.06	0.07	0.68	2
Levy2	1	$[-10, 10]$	0.06	0.05	1.62	4
Levy3	2	$[-10, 10]$	0.08	0.06	34.93	30
Levy4	2	$[-10, 10]$	0.06	0.10	4.14	7
Levy5	3	$[-10, 10]$	0.08	0.21	1.02	1
Levy5	5	$[-10, 10]$	0.08	0.19	1.62	1
Levy5	10	$[-10, 10]$	0.08	0.19	4.34	1
Levy5	20	$[-10, 10]$	0.08	0.29	14.48	1
Levy5	40	$[-10, 10]$	0.08	0.31	54.42	1
Levy5	80	$[-10, 10]$	0.08	0.33	216.49	1
Levy8	3	$[-10, 10]$	0.08	0.15	3.03	1
Levy8	5	$[-10, 10]$	0.08	0.17	5.83	1
Levy8	10	$[-10, 10]$	0.08	0.17	4.30	1
Levy8	20	$[-10, 10]$	0.08	0.20	13.48	1
Levy8	40	$[-10, 10]$	0.08	0.29	45.82	1
Levy8	80	$[-10, 10]$	0.08	0.30	294.18	1
Beale	2	$[-4.5, 4.5]$	0.06	0.15	2.76	3
Beale	2	$[-10^2, 10^2]$	0.06	0.15	3.71	12
Beale	2	$[-10^4, 10^4]$	0.07	0.15	5.81	31
Beale	2	$[-10^7, 10^7]$	0.07	0.15	22.98	61
Schwefel1	3	$[-10^7, 10^7]$	0.07	0.07	0.61	0
Booth	2	$[-10^7, 10^7]$	0.06	0.05	0.40	0
Powell	4	$[-10, 20]$	0.05	0.07	5.20	57
Schwefel3	2	$[-10^7, 10^7]$	0.06	0.02	0.62	0
Rosenbrock	2	$[-10^7, 10^7]$	0.06	0.07	1.42	10
Ratz1	5	$[-500, 600]$	0.07	0.09	1.53	0
Ratz25	4	$[0, 10]$	0.09	0.18	2.42	0
Ratz27	4	$[0, 10]$	0.09	0.21	3.99	0
Ratz210	4	$[0, 10]$	0.09	0.21	5.72	0
Ratz3	6	$[0, 1]$	0.09	0.11	17.48	2
More1	3	$[-4, 4]$	0.07	0.09	12.95	5
More2	4	$[-25, 25]$	0.08	0.16	271.45	65

Ratz210 without splitting. These problems were used in [30] to study splitting strategies.

5.3. 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 [11]. We give the number of variables in the initial statement (v), the number of constraints (c), the times for compilation, generation and solving, 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.

Table 3
Summary of the experimental results on constrained optimization

Benchmarks	v	c	Compilation	Generation	Solving	Splits
h95	6	16	0.13	0.06	6.74	6
h96	6	16	0.13	0.06	6.47	8
h97	6	16	0.13	0.06	294.14	258
h98	6	16	0.13	0.06	244.56	144
h100	7	18	0.11	0.07	108.04	143
h106	8	22	0.10	0.08	926.72	149
h113	10	28	0.19	0.05	4410.26	7296

6. Related work

In this section, we relate Helios to the systems it is most closely related to: the modeling language AMPL and the programming language Newton.

6.1. Comparison with AMPL

We first compare Helios to AMPL, which is one of the most advanced modeling languages. Ref. [5] contains a comparison of many modeling languages including GAMS and LINDO. From a syntactic standpoint, most of these languages are, in fact, very closely related.

Since Helios was primarily inspired by AMPL, it is not surprising that many of its concepts such as ranges, constants, sets, aggregation operators, and generic objects have direct counterparts in AMPL. These concepts are in fact closely related to the traditional mathematical notations and the differences between Helios and AMPL are essentially due to personal styles of the designers. However, Helios makes a number of contributions to the field of modeling languages.

As far as the application domain is concerned, Helios is the first (to our knowledge) modeling language for global optimization based on interval analysis. As a consequence, and contrary to other modeling languages, Helios provides soundness and completeness guarantees on its results. In particular, it is guaranteed to find all isolated solutions in constraint solving problems and all global optima in optimization problems.

As far as the syntax is concerned, the main novelty of Helios is the output of the statements: Helios returns a set of solution-boxes while AMPL and the other modeling languages we are aware of returns values. The ability to return intervals is critical to guarantee the soundness and completeness properties of the results. Another novelty is the concept of soft constraints which is also motivated by our application area and our desire to produce guarantees on the results.

As far as semantic issues are concerned, Helios differs significantly from other modeling languages. Traditional modeling languages abstract the syntax of constraint solvers but their semantics strongly depend on the nonlinear solvers used in the implementation. Although this is probably not a major issue for modeling languages which act as front-ends to linear programming solvers, it becomes a significant problem for

global optimization for reasons that we mentioned several times already. The semantics of Helios as presented in this paper makes it possible to abstract not only the syntax of a given solver but also its operational behavior provided that it respects our basic requirements. *As a consequence, the results of* Helios *can be interpreted independently of the implementation.*

6.2. *Comparison with* Newton

It is also useful to compare the use of Helios with direct programming in Newton to understand better the practical contributions of Helios.

Helios was motivated by the fact that most applications of Newton come from scientists who would prefer to avoid learning a specific programming language to

```
broyden(N,L) :-
        precision(8),
        generateConstraints(N,L), split(L).

generateConstraint(N,L) :- createDomain(1,N,L), generateConstraints(1,N,L).

generateConstraints(L,U,V) :-  L > U.
generateConstraints(L,U,V) :-  L <= U,
        generateConstraint(L,U,V),
        L1 is L + 1,
        generateConstraints(L1,U,V).

generateConstraint(I,N,V) :-
        L2 is I - 5, maximum(1,L2,L), U2 is I + 1, minimum(N,U2,U),
        gterm(1,L,U,I,V,Res),
        constraint Res.

gterm(C,L,U,I,V,1) :-  C > U.
gterm(C,L,U,I,[V|Vs],Res) :- C < L, C1 is C + 1,
        gterm(C1,L,U,I,Vs,Res).
gterm(C,L,U,I,[F|T],F * ( 2 + 5 * F^2) + Res) :- C = I, C1 is C + 1,
        gterm(C1,L,U,I,T,Res).
gterm(C,L,U,I,[F|T],Res - F * (1 + F)) :-
        C >= U, C <= U, C <> I, C1 is C + 1,
        gterm(C1,L,U,I,T,Res).

domain(L,U,[]) :- L > U.
domain(L,U,[F|T]) :-  L <= U, range(F,1.0,1.0), L1 is L + 1, domain(L1,U,T).

maximum(X,Y,X) :- X >= Y,
maximum(X,Y,Y) :- Y > X.

minimum(X,Y,X) :- X <= Y.
minimum(X,Y,Y) :- Y < X.
```

Fig. 10. A Newton program for the Broyden-banded function.

```
Input:
    int n :  "Number of variables:  ";
Range:
    idx = [1..n];
Set:
    J[i in idx] = { j in [max(1,i-5)..min(n,i+1)] | j <> i };
Variable:
    x :  array[idx] in [-10e8..10e8];
Body:
    solve system all
        f(i in idx):
            0 = x[i] * (2 + 5 * x[i]^2) + 1 -
                Sum(j in J[i])
                    x[j] * (1 +x[j]);
```

Fig. 11. An Helios statement for the Broyden-banded function.

solve their problems. Although Newton is high-level and declarative, there is still a considerable gap between a mathematical statement of a nonlinear application and the corresponding Newton program. To illustrate this point, Fig. 10 depicts the Newton program to solve the Broyden-banded function, while Fig. 11 describes an Helios statement for the same problem. For a noncomputer scientist, there is a substantial difference between the Helios statement and the corresponding Newton program. Note in particular the predicate gterm which is essentially implementing the generation of the summation operator. The Helios statement is clearer, more compact, higher-level, and much closer to the original mathematical description. Helios makes Newton technology accessible to a much wider audience.

Our results in the previous sections have also indicated that the additional functionality comes at a reasonable price: Helios statements are comparable in performance to Newton programs. The main conceptual reason explaining this result comes from our complexity results for the compilation and constraint generation phases and from the fact that most Newton programs follow a pattern of contraint generation and choices, which seems relatively uniform across our applications. Of course, Helios is more limited than Newton: it is not a full programming language and it does not let users write their own strategies for solving a problem (e.g., decomposition techiques, choice heuristics, ...). Some of these limitations could be removed at the expense of a more complex language. However, many problems seem in fact to require the standard search strategies only.

7. Conclusion

In this paper, we have introduced Helios, the first (to our knowledge) modeling language for global optimization using interval analysis. Helios makes it possible to

state global optimization problems almost as in scientific papers and textbooks and is guaranteed to find all isolated solutions in constraint-solving problems and all global optima in optimization problems. `Helios` statements are compiled to `Newton`, a constraint logic programming language using constraint satisfaction and interval analysis techniques.

From a user standpoint, `Helios` simplifies significantly the solving of these applications, while introducing a negligible overhead compared to direct programming in `Newton`.

From a modeling language standpoint, `Helios` differs from traditional modeling languages by its sound semantic foundation. We defined its semantics through a set of minimal requirements. Any implementation satisfying these requirements enjoys some nice soundness and completeness properties.

From an implementation standpoint, `Helios` indicates that constraint logic programming is an appropriate tool for implementing modeling languages. We showed that `Helios` statements can be compiled in linear time into `Newton` programs and that these `Newton` programs generate the set of constraints denoted by the statement in time linear in the size of the set (for reasonable statements).

Acknowledgements

Special thanks to Baudouin Le Charlier, Ugo Montanari, and the reviewers for many constructive suggestions. This work was supported in part by the Office of Naval Research under grant ONR Grant No. 0014-94-1-1153 and a NSF National Young Investigator Award with matching funds of Hewlett-Packard.

References

[1] F. Benhamou, D. McAllester and P. Van Hentenryck, CLP (intervals) revisited, in: *Proc. Internat. Symp. on Logic Programming (ILPS-94)*, Ithaca, NY (1994) 124–138.

[2] F. Benhamou and W. Older, Applying interval arithmetic to real, integer and Boolean constraints, *J. Logic Programm.* (1997), to appear.

[3] J. Bisschop and A. Meeraus, On the development of a general algebraic modeling system in a strategic planning environment, *Math. Programm. Study* **20** (1982) 1–29.

[4] J.G. Cleary, Logical arithmetic, *Future Generation Comput. Systems* **2**(2) (1987) 125–149.

[5] R. Fourer, Modeling languages versus matrix generators for linear programming, *ACM Trans. Math. Software* **9** (1983) 143–183.

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

[7] 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, Heidelberg, 1993).

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

[9] E.R. Hansen and R.I. Greenberg, An interval Newton method, *Appl. Math. Comput.* **12** (1983) 89–98.

[10] E.R. Hansen and S. Sengupta, Bounding solutions of systems of equations using interval analysis, *BIT* **21** (1981) 203–211.

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

[12] H. Hong and V. Stahl, Safe starting regions by fixed points and tightening, *Computing* **53** (1994) 323–335.
[13] R. Horst and P.M. Pardalos, *Handbook of Global Optimization* (Kluwer Academic Publishers, Dordrecht, 1995).
[14] R.B. Kearfott, Preconditioners for the interval Gauss-Seidel method, *SIAM J. Numer. Anal* **27** (1990).
[15] R.B. Kearfott, A review of preconditioners for the interval Gauss–Seidel method, *Interval Comput.* **1** (1991) 59–85.
[16] R.B. Kearfott, A review of techniques in the verified solution of constrained global optimization problems, to appear, 1994.
[17] R. Krawczyk, Newton-algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken, *Computing* **4** (1969) 187–201.
[18] A.V. Levy and A. Montalvo, The tunnelling algorithm for the global minimization of functions, *SIAM J. Sci. Statist. Comput.* **6** (1985) 15–29.
[19] A.K. Mackworth, Consistency in networks of relations, *Artificial Intelligence* **8** (1977) 99–118.
[20] K. Meintjes and A.P. Morgan, Chemical equilibrium systems as numerical test problems, *ACM Trans. Math. Software* **16** (1990) 143–151.
[21] U. Montanari, Networks of constraints: fundamental properties and applications to picture processing, *Inform. Sci.* **7**(2) (1974) 95–132.
[22] R.E. Moore, *Interval Analysis* (Prentice-Hall, Englewood Cliffs, NJ, 1966).
[23] R.E. Moore and S.T. Jones, Safe starting regions for iterative methods, *SIAM J. Numer. Anal.* **14** (1977) 1051–1065.
[24] J.J. More and M.Y. Cosnard, Numerical solution of nonlinear equations. *ACM Trans. Math. Software* **5** (1979) 64–85.
[25] J. Moré, B. Garbow and K. Hillstrom, Testing unconstrained optimization software, *ACM Trans. Math. Software* **7**(1) (1981) 17–41.
[26] A.P. Morgan, Computing all solutions to polynomial systems using homotopy continuation, *Appl. Math. Comput.* **24** (1987) 115–138.
[27] A.P. Morgan, *Solving Polynomial System Using Continuation for Scientific and Engineering Problems* (Prentice-Hall, Englewood Cliffs, NJ, 1987).
[28] A. Neumaier, *Interval Methods for Systems of Equations*, PHI Series in Computer Science (Cambridge Univ. Press, Cambridge, 1990).
[29] W. Older and A. Vellino, Extending prolog with constraint arithmetics on real intervals. in: *Canadian Conf. on Computer & Electrical Engineering*, Ottawa, 1990.
[30] D. Ratz, Box-splitting strategies for the interval Gauss–Seidel step in a global optimization method, *Computing* **53** (1994) 337–353.
[31] S.M. Rump, Verification methods for dense and sparse systems of equations, in: J. Herzberger, ed., *Topics in Validated Computations* (Elsevier, Amsterdam, 1988) 217–231.
[32] H. Schwefel, *Numerical Optimization of Computer Models* (Wiley, New York, 1981).
[33] P. Van Hentenryck, D. McAllister and D. Kapur, Solving polynomial systems using a branch and prune approach, *SIAM J. Numer. Anal.* **34** (1997) to appear.
[34] P. Van Hentenryck, L. Michel and F. Benhamou, Newton: constraint programming over nonlinear constraints, *Sci. Comput. Programm.* 1997, to appear.
[35] J. Verschelde, P. Verlinden and R. Cools, Homotopies exploiting newton polytopes for solving sparse polynomial systems, *SIAM J. Numer. Anal.* **31** (1994) 915–930.