Metalevel Interval Arithmetic and Verifiable Constraint Solving

Timothy J. Hickey
Michtom School of Computer Science
Volen Center for Complex Systems
Brandeis University, USA
tim@cs.brandeis.edu

October 19, 2001

Abstract

CLIP is an implementation of CLP(Intervals) which has been designed to be verifiably correct in the sense that the answers it returns are mathematically correct solutions to the underlying arithmetic constraints. This fundamental design criteria affects many aspects of the implementation from the input and output of decimal constants to the design of the interval arithmetic libraries and the constraint solving algorithms. In particular, to enhance verifiability, CLIP employs the simplest model of constraint solving in which constraints are decomposed into sets of primitive constraints which are then solved using a library of primitive constraint contractors. This approach results in a simple constraint solver whose correctness is relatively straightforward to verify, but the solver is only able to solve relatively simple constraints. In this paper, we present the syntax, semantics, and implementation of CLIP, and we show how to use metalevel techniques to enhance the power of the CLIP constraint solver while preserving the simple structure of the system. In particular, we demonstrate that several of the box-narrowing algorithms from the Newton and Numerica systems can be easily implemented in CLIP. The principal advantages of this approach are (1) the resulting solvers are relatively easy to prove correct, (2) new solvers can be rapidly prototyped since the code is more concise and declarative than for imperative languages, and (3) contractors can be implemented directly from mathematical formulae without having to first prove results about interval arithmetic operators. Finally, the source code for the system is publicly available, which is a clear prerequisite for public, independent verifiability.

1 Introduction

Interval arithmetic is an approach to solving numerical problems by performing computations on sets of reals rather than on floating point approximations to reals. There are many versions of interval arithmetic that differ mainly in the way they represent sets of reals. The classical approach [30] considers only closed, bounded, floating point intervals

$$[a,b] = \{x \in \mathcal{R} : a \leq x \leq b\}$$

where a, b are finite floating point numbers. One then builds libraries of routines for computing an interval containing the range of various classes of mathematical functions. For example, the interval multiplication procedure, $mult(X,Y)$, must satisfy

$$\{x * y : \ x \in X, \ y \in Y\} \subseteq mult(X,Y)$$

and this can be achieved by defining

$$mult([a,b],[c,d]) = [min(a *_{lo} c, a *_{lo} d, b *_{lo} c, b *_{lo} d), max(a *_{hi} c, a *_{hi} d, b *_{hi} c, b *_{hi} d)]$$

where $x *_{lo} y$ and $x *_{hi} y$ are the optimal floating point approximations to $x * y$ which satisfy

$$x *_{lo} y \leq x * y \leq x *_{hi} y$$

for all floating point numbers x, y. On the other extreme, the most general approach used in interval arithmetic systems today defines intervals to be finite unions of connected sets of reals. Such an interval can be represented as a union of connected subsets of reals

$$\bigcup_i [a_i, b_i]_{\alpha_i,\beta_i}$$

where each connected subset is represented by an interval $[a, b]_{\alpha,\beta}$

$$[a, b]_{\alpha,\beta} = \{x \in \mathcal{R} : a \leq x \leq b \ \wedge \ (\alpha \vee (a < x)) \ \wedge \ (\beta \vee (x < b))\}$$

with endpoints a, b in the extended reals. The endpoints themselves may or may not be in the set, depending on the value of the boolean variables α and β. The arithmetic operations on such sets are more complex, but are still efficiently computable on modern hardware [23].

There are two principal advantages of interval arithmetic over classical numerical analysis which both rely on the fact that interval evaluations return a superset of the mathematically correct result. The first is that all roundoff error from the underlying hardware arithmetic operations is automatically incorporated into the result interval. Thus, interval evaluation can be viewed as automatically performing both a calculation and an error analysis of the calculation. The second is that interval arithmetic allows one to compute provably correct upper and lower bounds on the range of a function over an interval. This is useful in solving global optimization problems.

An important application of interval arithmetic is the construction of verifiable constraint solvers, which solve constraints by returning intervals which are guaranteed to contain all real solutions. Interval arithmetic constraint solvers can be written in any language provided one has access to a library of interval routines. Writing solvers in imperative languages introduces an extra level of complexity as compared to standard numerical approaches, as one must handle not only real and integer variables, but interval variables as well.

A more declarative approach to integrating interval arithmetic into a computer language is followed in the CLP(Intervals) family of Constraint Logic Programming languages [25] which extend Prolog by allowing constraints over the domain of reals. The constraint solver for a CLP(Intervals) language associates an interval to each variable (initially $(-\infty, \infty)$) and uses the constraints to contract these intervals without removing any solutions to the constraints. The answer to a CLP(Intervals) query specifies an interval for each real variable in the query, and this tuple of variables is guaranteed to contain all solutions to the query. This provides a highly declarative approach to handling interval constraints. For example, to find the unique fixed point of the cosine function, one executes the following query and gets an interval solution guaranteed to contain the answer.

```
?- {cos(X)=X}.
X = [0.73908513321515, 0.73908513321517]
```

Historically, there have been two approaches to implementing the constraint solvers inside CLP(Intervals) languages: the RISC approach of CLP(BNR) [7, 33, 32, 6, 2], and the CISC approach of Newton and similar systems [3, 39, 38, 4, 15, 27]. In the RISC approach each constraint is decomposed into primitive constraints (similar to compiling to 3-address code), and then a general constraint solving engine is invoked to repeatedly select a primitive constraint and use it to contract the intervals it constrains, until some termination condition is satisfied. In the CISC approach, the constraint solver uses a more powerful constraint solver to handle each constraint.

The RISC approach yields a constraint solver which is relatively easy to implement and to prove correct, but which is much weaker than the CISC solver. For example, the constraint

```
?- {X+Y=2,X-Y=0}.
```

has the unique solution $X = Y = 1$, but a RISC style solver will not be able to solve this constraint because neither of the primitive constraints provide enough information to contract the intervals for X

and Y. Indeed, for any real number X there is a real number Y such that $X + Y = 2$, thus the constraint $X + Y = 2$ can not, by itself, eliminate any possible values for X. A CISC-style solver on the other hand could analyze both constraints together and apply Gaussian elimination or some other method to solve the problem.

In this paper, we propose a hybrid approach where one first implements a fast RISC-style CLP system and then uses meta-level techniques to build a CISC-style constraint solver on top of this RISC system.

The main benefits of this hybrid approach are:

- it is relatively straightforward to provide a convincing argument for the correctness of the combined system, whereas proving the correctness of a CISC-style solver implemented in traditional languages such as C or Java would be more difficult.
- new solvers can be rapidly prototyped since the necessary code is simpler and more concise in CLP(Intervals) than in extensions of imperative or functional languages that have been augmented by interval arithmetic libraries (e.g. C [26], Java[20], Fortran [37]).
- contractors can be implemented directly from the corresponding mathematical formulae without having to detour through the language of interval arithmetic. In particular, one does not need to consider interval extensions of functions.

In this paper, we describe CLIP, a RISC-style verifiable CLP(Intervals) system we have developed [17] and we show that some of the Newton/Numerica style contractors can be easily and efficiently implemented at the metalevel in CLIP. The CLIP system has been designed to be verifiable, and has been hand-verified, but we do not claim to have formally verified the correctness of its implementation. Nevertheless, the mere fact that formal verifiability is an eventual goal has influenced the design and has contributed to the stability of the system. The problem of formally verifying the correctness of a hardware/software system is an important one, but one which we will not investigate here.

We have successfully used the metalevel approach with CLIP to build constraint solvers for a large class of Ordinary Differential Equation (ODE) constraints [16, 18], but have yet to incorporate the higher level ODE contractors of Deville, Janssen, and van Hentenryck[10] or the multidimensional Taylor approximations of Berz and Hoffstätter [5]. Our metalevel approach is similar to Puget and Leconte's [36], except that we manipulate constraints at the metalevel while they advocate giving constraints first class status in the underlying constraint solver. Another approach to mixing RISC/CISC style constraint solving is the DECLIC system[13] which differs from ours in that it provides both RISC and CISC capabilities in the underlying constraint solver itself. CIAL [27] is another RISC/CISC style system which handles non-linear constraints in a CISC-manner in the underlying constraint solver.

In the remainder of this paper, we first give an overview of the underlying CLP(Intervals) system CLIP with special attention paid to the goal of developing a verifiably correct system. Next we show how to extend this design using metalevel techniques to build higher level Numerica-like contractors based on Taylor's theorem. We then give some performance benchmarks for several concrete examples. Finally, we describe the implementation and semantics of CLIP and discuss directions for future research. An early version [19] of this paper appeared in the Second International Workshop on Practical Aspects of Declarative Languages (PADL'00).

2 Overview of the CLP(Intervals) language: CLIP

Although there are several RISC-style CLP(Intervals) systems that are currently available (e.g. BNR-PROLOG [33], clp(BNR) [32], Prolog IV [35], ECLiPSe/RIA[11], DECLIC [13]) our interest was in a system whose underlying constraint solver was very fast and publicly accessible (so we could analyze it), and which used algorithms that had been proved correct. None of the systems we had seen at the time had this property especially in regard to the correctness of the implementations of the constraint contractors for the elementary functions $\exp, \log, \sin, \cos$ etc. This was the primary motivation for developing the

```
Relations:  X=Y, X<Y, X=<Y, X>Y, X>=Y, X\==Y, integer(X), boolean(X)
Constants:  0, -1, 1.23, 1.0E100, '1.10000*', '1.123...', pi,
Functions: X+Y, X-Y, X*Y, X/Y, -X, abs(X),
           exp(X), log(X), X^n, X^(p/q), X**Y,
           sin(X), cos(X), tan(X), asin(X), acos(X), atan(X),
Variables: (capitalized identifiers)
```

Figure 1: CLIP constraint language

CLIP system [17]. The metalevel techniques used in this paper, could also be applied to any other RISC-style system, but verifiability would depend on the correctness of their underlying constraint contraction algorithms.

We also wanted a system whose correctness could be verified independently of the correctness of the Prolog engine. For this reason, the CLIP system was built as a foreign-file extension to Prolog which could easily be grafted on to other existing Prolog systems. Currently it can be used with GnuProlog[14, 8] but earlier versions were easily ported to other systems (Sicstus Prolog and ALS Prolog). The CLIP constraint solver is a C program which maintains all intervals and constraints in its own internal arrays. These intervals and arrays are referenced from the Prolog engine by their indices in these arrays. The implementation is discussed in more detail in Section 6.

2.1 Syntax and Semantics of CLIP

Syntactically, CLIP is an extension of Prolog in which arithmetic constraints are enclosed in curly brackets and are separated by commas. Each constraint is an atom in a particular first order theory of the reals. This theory contains some specified set of relation, function, constant, and variable symbols. In CLIP the currently allowable symbols include those in Fig. 1.

CLIP uses an interval arithmetic [23] in which intervals are defined to be topologically closed and connected subsets of the reals. Thus, initially all intervals have bounds $[-\infty, \infty]$. This is at odds with the established view of intervals in numerical analysis which requires both endpoints to be finite. Some CLP(Intervals) implementations have more sophisticated interval systems, e.g. Prolog IV allows unions of open and closed intervals.

The primitive constraint solvers in CLIP have all been written from scratch using methods described in Section 6. In particular, the solvers for the arithmetic constraints ($X + Y = Z$, $X - Y = Z$, $X * Y = Z$, $X/Y = Z$) have been implemented using algorithms which have been proved correct in [23], and the constraint solvers for the elementary functions ($\exp, \log, \sin, \ldots$) were implemented directly in terms of the arithmetic constraint solvers using the Taylor series with remainder and various range reduction methods which we have proved correct. In particular, we have used none of the standard math libraries to compute $\exp, \log$ etc. because there were no formal proofs for the correctness of these implementations with respect to any fixed error bounds. There have recently appeared several methods for building math libraries with established error bounds that have been proved correct [34, 41]. Although such libraries could be used to build verifiable interval arithmetic libraries which would be potentially faster than ours, their verification is also substantially more difficult since they must perform a very careful bit-by-bit error analysis to rigorously bound the error. In our implementation, the interval arithmetic routines do this analysis for us.

Semantically, CLIP constraints are interpreted as relations among real numbers. If a CLIP constraint $C(X_1, \ldots, X_n)$ fails, then there is implicitly a proof that there are no real numbers X_i which make the constraint true.

2.2 Constraint Propagation in CLIP

CLIP is designed so that the solver is powerful enough to be able to always solve constraints that require only simple propagation. More precisely, if the constraints can be rearranged in some order so that they have the form $X_i = E_i$ where E_i is an expression only involving variables $X_1, \ldots, X_{i-1}$, then CLIP will correctly evaluate all X_i, e.g.,

```
| ?- {Z = exp(5/2)-1, Y=(cos(Z)/Z)^(1/3), X=1+log((Y+3/Z)/Z)}.
   X = -2.06163426224723...
   Y = 0.2551887203100...
   Z = 11.18249396070347 ... ?
```

Note that the answers use the "..." notation. As we will see in the next subsection, this indicates that all decimal digits shown are mathematically correct.

CLIP solves a system of constraints by introducing temporary variables and transforming the system into a conjunction of primitive constraints which is solved by an iterative narrowing algorithm described in Section 6. For example, the constraint above would be transformed to:

```
{T1=5/2, T2=exp(T1), Z = T2-1,
 T3=cos(Z), T4=T3/Z, Y=T4^(1/3),
 T5=3/Z, T6=Y+T5, T7=T6/Z, T8=log(T7), X=1+T8}.
```

Once this translation is made, the solver iteratively picks a constraint $C(X_1, \ldots, X_n)$ and uses it to contract the intervals X_i it relates. For example, the contraction procedure $K_{exp}(X, Y)$ for the constraint $Y = \exp(X)$ contracts $X = [a, b]$ and $Y = [c, d]$ simultaneously via

$$\begin{aligned} Y &\leftarrow Y \cap [\exp_{lo}(a), \exp_{hi}(b)] \\ X &\leftarrow X \cap [\log_{lo}(max(0, c)), \log_{hi}(max(0, d))] \end{aligned}$$

where $\exp_{lo}, \exp_{hi}, \log_{lo}, \log_{hi}$ are procedures returning floating point numbers giving verifiable lower and upper bounds of exp and log respectively:

$$\exp_{lo}(x) \leq \exp(x) \leq \exp_{hi}(x) \quad \log_{lo}(x) \leq \log(x) \leq \log_{hi}(x)$$

and $\log_{lo}(0) = -\infty$, $\log_{hi}(\infty) = \infty$, $\exp_{lo}(-\infty) = 0$, $\exp_{hi}(\infty) = \infty$.

Note that $Y = \log(X)$ if and only if $X = \exp(Y)$ and so the contraction procedure for log can be defined simply using the contraction operator for exp: $K_{log}(X, Y) = K_{exp}(Y, X)$. Note also that most publicly available math libraries (e.g. fdlibm [12]) do not come with verified estimates of their error and hence can not be used to implement verifiable interval functions. In Section 6 we discuss our approach to implementing these interval functions directly.

One immediate benefit of the primitive decomposition approach is that CLIP is able to solve seemingly more complex constraints like the following

```
| ?- {2*log(Z+1) = 5, Z*exp(X-1)=Y+3/Z, Z*Y^3=cos(Z)}.
   X = -2.06163426224723...
   Y = 0.2551887203100...
   Z = 11.18249396070347 ... ?
```

which generates the same primitive constraint set as the previous one except for primitive equivalences, such as $Y = \exp(X)$ being replaced by $\log(Y) = X$, which is handled automatically by the solver, as mentioned above.

The CLIP solver will also make a limited attempt to solve more complex constraints which require iterative constraint propagation, e.g.,

```
| ?- {X*Y=1, Y=sin(X), pi/2 >= X, X >= 0}.
   X = 1.1141571408719...
   Y = 0.89753946128048...
```

(See [1], p. 77 for a classical approach to this problem, or [21] for a comparable constraint contraction approach.)

The user is allowed to set parameters which put a limit on how much work the CLIP solver will invest each time it is called. This allows one to write CLIP programs knowing that each time the constraint solver is invoked (by writing a constraint in curly braces), control will always return to the Prolog engine in time at most T, where T is implicitly determined by setting various constraint engine parameters. On the other hand, such limits imply that the constraint solver will have stopped before a fixed point was reached and hence the computed intervals may not be as narrow as would otherwise have been the case. To counteract this loss, CLIP provides a method, `narrow_all`, which reactivates all constraints and reinvokes the constraint solver. This guarantees that constraint narrowing which was prematurely halted by the constraint engine can be resumed at a later time, typically after all constraints have been added to the system. These parameters and their implementation are discussed in more detail in Section 6.2 below.

2.3 Syntax and Semantics of Real Constant Symbols

One of the more subtle aspects of building a verifiable constraint solver is the problem of maintaining a rigorous semantics for real constants. The difference between base 2 and base 10 representations of numbers requires most decimal constants c to be represented by intervals $[L_c, U_c]$, where L_c and U_c are the nearest floating point numbers to c. If this is not done carefully, logically incorrect results can follow. Consider, for example, the following query:

```
?- {X = 1.1, (X-1)*10=1+Z, Z=0}.
```

The simplest implementation of real constants would simply replace 1.1 by the nearest double precision floating point number a to 1.1. In this case, we have

```
a = 1 + 450359962737050*2^{-52} = 1.1 + 0.4*2^{-52} =
1.100000000000000088817841970012523233890533447265625
```

but this would cause the query to fail because mathematically we have $Z = (X-1)*10-1$, and substituting in $a = 1.1 + 0.4 * 2^{-52}$ for X we get $Z = (a-1) * 10 - 1 = 2^{-50} \neq 0$, when Z should have the value $(1.1-1) * 10 - 1 = 0.0$.

To avoid these problems, we follow CLP(BNR)[2, 7] and let the floating point constants denote the smallest floating point intervals which contain them. Since this introduces a small loss of precision which the user may wish to avoid, CLIP also allows the use of the quoted constants listed below to give the user more flexibility in deciding how to interpret constants. Thus,

- {`X = 1.1`} will bind `X` to a small interval with floating point bounds which contains `X` (but possibly not the smallest interval, since it is first processed by the Prolog engine not the CLP engine).
- {`X = '1.1'`} will bind `X` to the smallest interval with floating point bounds which contains `X`. Note that the quotes are needed so that the conversion from a string to a double will be done by the constraint engine and not by the prolog engine. Thus a quoted fifty decimal digit number might be exactly represented by a double precision floating point number and hence would be represented by a point interval.
- {`X = '1.1#'`} or {`X = '#1.1'`} will bind `X` to the floating point number which is closest to 1.1. This is useful when the user really does want a point interval close to a decimal number. In case of a decimal number which is exactly halfway between two floating point numbers, the floating point number farthest from zero is selected.
- {`X = '1.100*'`} will specify that `X` is only known to four digits of accuracy, i.e., the asterisk indicates that the number `X` is in `[1.0995,1.1005]`.
- {`X='1.1000...'`} where the "..." notation indicates that all of the decimal digits listed so far are correct, so, in this case, `X` represents a number in `[1.1000,1.1001]`.

The same conventions are used for printing intervals, guaranteeing a rigorous semantics for the answer constraints.

Finally, CLIP allows the notation `[L,H]` where `L, H` are real constants, for expressing a real variable T with the constraints $(L \leq T) \wedge (T \leq H)$. Thus, the constraint $X^2 = \sin([1,2])$ is equivalent to $(X^2 = \sin(T)) \ \wedge \ (1 \leq T) \ \wedge \ (T \leq 2)$ where T is a new interval variable.

3 Metalevel Construction of Higher level Contractors

In this section we present two higher level contractors, both of which are derived from the Taylor formula. We will present the implementation and a correctness proof, assuming that the underlying CLIP interpreter has been proved correct.

The advantage of using CLIP as the implementation language for the solver is that the correctness of the contractors is equivalent to a corresponding mathematical property of the function being contracted. Making this correspondence explicit provides a methodology for proving the correctness as part of the implementation rather than as a separate analysis.

3.1 The Taylor Formula

The first contraction we implement is the simple Taylor contraction. It is based on the Taylor formula, which states that any function f, of one real variable x, which is continuously differentiable on an interval I must satisfy the following property:

$$\forall a, x \in I, \exists t \in [0,1], \xi \in \mathcal{R} \qquad f(x) = f(a) + f'(\xi)(x-a) \ \wedge \ \xi = a + t * (x-a)$$

Observe that the second equation states that ξ is a point which lies between a and x. The idea behind the Taylor contraction is to solve a constraint of the form

$$f(x_1, \ldots, x_n) = 0$$

by adding, for each i, a redundant constraint of the form

$$0 = f_i(a_i) + (x_i - a_i) f_i'(\xi_i), \ \ \xi_i = a_i + t_i * (x_i - a_i)$$

where t_i and ξ_i are new interval variables and

$$f_i(z) = f(x_1, \ldots, x_{i-1}, z, x_{i+1}, \ldots, x_n)$$

is f viewed as a function of x_i, and where $(a_1, \ldots, a_n)$ can be any point in the domain of f. One usually gets the best convergence however by selecting a_i to be the midpoint of the current value of x_i.

Observe that the Taylor constraint differs from the standard interval Newton operator in that the latter evaluates f' on x_i not on $a_i + t_i * (x_i - a_i)$. Indeed, the standard Interval Newton contraction is

$$x_i \quad \leftarrow \quad x_i \cap (a_i - (f_i(a_i)/f_i'(x_i)))$$

In the constraint context, the extra complexity of adding t_i is necessary, because the constraint variables represent real numbers not real sets. Note also that the Taylor constraint has the advantage that as x_i is contracted toward a_i this is reflected in the smaller active domain for f_i'. Thus, one Taylor constraint contraction can perform more narrowing than one Interval Newton contraction.

3.1.1 An example

Consider the case of the Broyden Banded Function example (the first example cited in [39]) which is to solve the following system of n equations in $\{x_1, \ldots, x_n\}$

$$x_i(2 + 5x_i^2) + 1 = \sum_{i-5 \leq j \leq i+1, j \neq i, j \in [1,n]} x_j(x_j + 1)$$

For example, when $n = 2$, the Broyden problem is to solve

$$\begin{aligned} 2x_1 + 5x_1^3 + 1 &= x_2 + x_2^2 \\ 2x_2 + 5x_2^3 + 1 &= x_1 + x_1^2 \end{aligned}$$

The simple Taylor contraction for this system consists of the following constraint, which is true assuming that (x_1, x_2) is a solution to the two Broyden equations above

$\forall a_1, a_2\ \exists t_1, t_2, t_3, t_4, z_1, z_2, z_3, z_4$ s.t.

$$\begin{aligned} 0 &= 2a_1 + 5a_1^3 + 1 - (x_2 + x_2^2) + (2 + 15z_1^2)(x_1 - a_1), \\ 0 &= 2x_1 + 5x_1^3 + 1 - (a_2 + a_2^2) + (-1 - 2z_2)(x_2 - a_2), \\ 0 &= 2x_2 + 5x_2^3 + 1 - (a_1 + a_1^2) + (-1 - 2z_3)(x_1 - a_1), \\ 0 &= 2a_2 + 5a_2^3 + 1 - (x_1 + x_1^2) + (2 + 15z_4^2)(x_2 - a_2), \\ & z_1 = a_1 + t_1(x_1 - a_1), 0 \le t_1 \le 1 \\ & z_2 = a_2 + t_2(x_2 - a_2), 0 \le t_2 \le 1 \\ & z_3 = a_1 + t_4(x_1 - a_1), 0 \le t_4 \le 1 \\ & z_4 = a_2 + t_3(x_2 - a_2), 0 \le t_3 \le 1 \end{aligned}$$

To get good convergence, we select a_1 and a_2 to be the midpoints of x_1 and x_2 and the standard CLIP solver will then narrow the bounds of the x_i.

There are two strategies for applying such a contractor. One is to create new redundant constraints with different a_i each time the contraction is called. This tends to add many constraints to the constraint store and those corresponding to older values of a_i usually contribute very little to the contractions. Our approach is to use these constraints to compute contracted bounds for the x_i using a forward checking operator (implemented with `findall`) as described in Section 3.1.3. This has the advantage of only storing the constraints in the constraint store temporarily while the bounds are being contracted. The soundness of the contraction follows from the soundness of the CLIP semantics, but the forward checking approach allows us to keep the set of active constraints bounded even with a large number of Taylor contractions.

3.1.2 Invoking Contractions

The CLIP implementation of the simple Taylor contraction described above is already powerful enough to solve the Broyden Banded Function examples. Syntactically we define two predicates one to create a contractor and another to apply the contractor. Thus, to solve the 2 variable instance of the Broyden problem we would give the following query:

```
 ?- broyden(2,Vs,Es),make_contractor(taylor,Es,Vs,T),
    iterate_until(T,10,allsmall(Vs,1.0e-6)), Vs=[X,Y].

  X = -4.2730462355816...e-01
  Y = -4.27304623558166...e-01 ?

(60 ms) yes
| ?-
```

where the predicate `broyden(N,Vs,Es)` constructs (symbolically) the Broyden system `Es` on the list `Vs` of `N` variables, the `iterate_until(T,N,C)` operator calls `T`, the contractor, at most `N` times or until `C` is true, and the `make_contractor` predicate creates a contractor as we describe in the next section.

3.1.3 Implementing Contractions

In this section we describe the implementation of the general predicates

```
make_contractor(taylor,Es,Vs,taylor(Vs,As,T))

taylor(Vs,As,T)
```

which will create and apply the Taylor contraction to a list `Es` of constraints on a list `Vs` of variables. The contractor consists in applying the Taylor constraint

$$0 = f(a) + (x-a)f'(b), \;\; b = a + t*(x-a)), \;\; t \in [0,1]$$

for each equation $f = 0$ in `Es`, and each variable x in `Vs`, where the (partial) derivatives f' are computed using a combination of symbolic differentiation and interval contraction.

To illustrate the techniques needed to implement this contractor in CLIP we show in Fig. 2 the predicates to create and apply the Taylor contractor for a single variable `X` and a single expression `Exp1=Exp2`. Observe that the contractor is defined directly in terms of the Taylor formula and so its correctness depends only on the correctness of the underlying constraint solver. The auxiliary procedures used in the figure are not shown but are discussed in the next paragraphs.

The procedure `copyExpr(E,X,A,F)` is a meta-level predicate which makes a copy `F` of the expression `E`, but replaces every occurrence of `X` by `A`. The `diff(F,D,DF,DEs)` predicate produces both a variable `DF` representing the value of the derivative of `F` at `X`, and a set `DEs` of equations which relate `DF` to the other variables and constants in `F`, using rules of the form:

```
diff(X*Y,D,DF,(DF=X1*Y+X*Y1,E1,E2)) :-
   diff(X,D,X1,E1), diff(Y,D,Y1,E2).
```

This provides a very simple approach to automatic numerical differentiation. For example, differentiating $a*\sin(b*\cos(a))$ with respect to a gives a set of numeric constraints `Es` on `A,B,Dif` and the new variables `C,D,E` whose solution is the derivative `Dif`:

```
?- diff(A*sin(B*cos(A)),A,Dif,Es).

Es = (Dif=C+sin(B*cos(A)),C=A*D,D=cos(B*cos(A))*E, E=B*-sin(A)).
```

Thus, in the call `diff(FB,B,DFB,DEs)` in Figure 2, the derivative of the expression `FB` with respect to the variable `B`, is computed as a variable `DFB` whose value is the solution of a set `DEs` of symbolic equations. These equations are then added to the term `C` representing the Taylor constraint.

Observe that the list, `Es`, of equations may contain repeated expressions (e.g. `cos(A)` in the example above) which must be evaluated twice. This can be remedied, using common subexpression elimination, by introducing temporary variables for each repeated subexpression. For the example above this would produce:

```
Es = (Dif=C+sin(T),T=B*cos(A),C=A*D,D=cos(T)*E, E=B*-sin(A)).
```

When the taylor contraction is later applied by invoking the procedure `taylor([X],[A],C)`, this procedure computes the midpoint M of the interval X and invokes the constraint {`A=M,C`}. This constraint causes CLIP to solve the Taylor constraint centered at the midpoint of the interval, and, as a result, it causes the derivative `DFB` over the interval `X` to be computed as the solution to the constraints `DEs`. All of this constraint solving is done in the context of a `forward_check` predicate so that the constraints can then be removed from the constraint store. This has the effect of "unbinding" the variable `A` and allowing the taylor contraction to be applied again later with a different value of `A`. It is interesting that the derivative is

```
% generate a set of contraints for one equation using Taylor
make_contractor(taylor,Exp1=Exp2,[X], taylor([X],[A],C)) :-
  C= (Exp1=Exp2, 0 = FA + DFB*(X-A),
      B = A + T*(X-A), 1 >= T, T >= 0,
      DEs),
  Expr = Exp1 - Exp2,
  copyExpr(Expr,X,A,FA),  copyExpr(Expr,X,B,FB),
  diff(FB,B,DFB,DEs).

% apply a taylor contraction for one variable
taylor([X],[A],C) :-
  forward_check([X],(midpoint(X,M),{A=M,C})).


% generate a set of contraints for each variable
make_contractor(taylor,Es,[X1,X2|Xs],taylor([X],[A1,A2|As],(C1,C2))) :-
  make_contractor(taylor,Es,[X1],taylor([X1],[A1],C1)),
  make_contractor(taylor,Es,[X2|Xs],taylor([X2|Xs],[A2|As],C2)).

% generate a set of contraints for each equation
make_contractor(taylor,(E1,E2),[X],taylor([X],[A1],(C1,C2,A1=A2))) :-
  make_contractor(taylor,E1,[X],taylor([X],[A1],C1)),
  make_contractor(taylor,E2,[X],taylor([X],[A2],C2)).

% apply a taylor contraction for several variables
taylor([X1,X2|Xs],[A1,A2|As],C) :-
  forward_check([X1,X2|Xs],
    (bind_to_midpoints([X1,X2|Xs],[A1,A2|As]),{C})).
```

Figure 2: Implementation of the Taylor Contraction

computed both symbolically (as a solution to a symbolic set of equations) and numerically (as an interval resulting from the contraction of those equations).

Finally, the `forward_check(X,G)` predicate finds upper and lower bounds on `X` for each solution to the query `G`, and then forms the union `U` and then contracts `X` by intersecting it with `U`. Here `X` can be a list of variables and the union finds the smallest box `U` which contains all of the boxes in `Bs`.

```
forward_check(X,G) :-
  findall(B,(G,get_bounds(X,B)),Bs), union(Bs,U), bind(X,U).
```

Note that forward checking in this manner temporarily adds constraints to the constraint store to compute the bounds `B` of each solution to `G`. The temporary constraints are then removed from the constraint store (by backtracking through a `findall`) and the remembered bounds are used to contract `X`.

The last three clauses in Figure 2 extend the program to handle sets of constraints in several variables. The approach is to generate a set of constraints for each variable and each constraint. These constraints are applied simultaneously using the forward checking procedure.

3.2 Multivariate Taylor contractions.

The mathematical formula for multivariate contractions is very similar to the univariate formula. Assuming that f is a continuously differentiable map from a convex subset U of $\mathcal{R}^m$ to $\mathcal{R}$, the multivariable Taylor formula states that:

$\forall a, x \in U, \exists t \in [0,1], \exists \xi \in U$:

$$\begin{aligned} f(x) &= f(a) + Df(\xi) \cdot (x-a) \\ \xi &= a + t*(x-a), \;\; t \in [0,1] \end{aligned}$$

where $Df(z) = \left(\frac{\partial f}{\partial x_j}(z)\right)$ is the multivariate derivative of f at z. This formula can be deduced from the univariate formula applied to

$$g(s) = f(a + s*(x-a))$$

Straightforward translation of this formula into CLIP yields the "centered form" contraction of the equation $f(x) = 0$. For example, the centered form contraction constraint for the 2 variable Broyden problem is as follows:

$\forall a_1, a_2 \; \exists t_1, t_2, z_{11}, z_{12}, z_{21}, z_{22}$ s.t.

$$\begin{aligned} 0 &= 2a_1 + 5a_1^3 + 1 - (a_2 + a_2^2) + (2 + 15z_{11}^2)(x_1 - a_1) - (1 + 2z_{12})(x_2 - a_2), \\ 0 &= 2a_2 + 5a_2^3 + 1 - (a_1 + a_1^2) + (2 + 15z_{22}^2)(x_2 - a_2) - (1 + 2z_{21})(x_1 - a_1), \\ z_{11} &= a_1 + t_1(x_1 - a_1), \; z_{12} = a_2 + t_1(x_2 - a_2), \\ z_{21} &= a_1 + t_2(x_1 - a_1), \; z_{22} = a_2 + t_2(x_2 - a_2), \\ t_1, t_2 &\in [0,1] \end{aligned}$$

Multivariate Taylor contractions are implemented as in the previous section. One adds clauses to `make_contractor` of the form

```
make_contractor(cftaylor,Es,Vs,cftaylor(Vs,As,T))
```

to handle the `cftaylor(Vs,As,T)` contractor by creating a set of constraints for each equation `E` in `Es` viewing the lists `Vs` and `As` as vectors of variables. The multivariate contractor can then be invoked as follows:

```
?- broyden(2,Vs,B),make_contractor(cftaylor,B,Vs,T),
   iterate_until(T,10,allsmall(Vs,1.0e-6)),Vs=[X,Y].
```

4 Composing Contraction Operators

In the previous section we showed how to implement two higher level contractors in CLIP and proved their correctness assuming the correctness of the underlying CLIP interpreter. There are certainly many other higher level contraction operators that could be implemented, and some of these will be discussed in the final remarks. Once a collection of primitive higher order contractors become available however, one must then choose which contractors to apply at what times to solve a given problem. One approach, used so successfully by Numerica, is to hardwire in a finely tuned strategy. Another approach, presented in this section is to define compound contraction operators where the user can explicitly specify parameters that tune the application of the operator. An advantage of this approach is that the compound contractors are guaranteed to be sound, because each one is equivalent to some sequence of compound contractors. Thus the user is free to experiment with a wide variety of solving strategies without having to worry about correctness. In the remainder of this section we describe several of these contraction composition operators.

```
% absolve(Vars,Choose,Constraints,
%         Percent,NumIter,Termination)
  absolve(_Vs,_Ch,_Co,_P,_N, Terminate) :-    Terminate,!.
  absolve(_Vs,_Ch,_Co,_P, N,_Te) :-    N=<0,!.
  absolve( Vs, Ch, Contract, P, N, Te) :-    N>0, N1 = N-1,
    choosevar(Ch,V,Vs,NewVs),
    forward_check(Vs,(multisplit(V,P,Contract))),
    Contract,
    absolve(NewVs,Ch,Contract,P,N1,Te).

multisplit(V,P,C) :- get_bounds_clip(V,L,H),
   D is (H-L)*P,   ms(V,C,L,D,H).

ms(V,C,L,D,_) :- {V=<L+D},C.
ms(V,C,_,D,H) :- {V>=H-D},C.
ms(V,_,L,D,H) :- {L+D =<V, V =< H-D}.
```

Figure 3: Implementing Absolve

4.1 Absolving

The first operator we implement is what we call the **absolve** operator. The `absolve(Vs,M,Cs,P,N,T)` operator contracts each of the variables `V` in the list `Vs` by removing from `V` small subsets near the endpoints in which it can prove there are no solutions. The relative size of the subset it tries to remove is given by the parameter `P`. A limit on the number of times the absolve procedure can iterate is given by `N`. Another termination procedure is given by `T`. The variable `M` encodes the method used to select which variable `V` to consider next. This algorithm can be easily expressed in CLIP as shown in Fig. 3. The absolve operator plays a key role in the box-consistency algorithm of Newton/Numerica, and has been called the squash algorithm in the RIA solver of the ECLiPSe CLP system[11]. It was also called **absolving** in early implementations of CLP(BNR) [32].

For example, the following query will solve the 20 variable Broyden problem using the absolve algorithm where the simple Taylor contractor is applied at each step in an attempt to trim 12.5% off of each end of the interval. Termination occurs when all variables have relative or absolute width at most 10^{-6} or when a limit of 10 iterations has been exceeded:

```
?-  broyden(20,Vars,Eqns),
    make_contractor(taylor,Eqns,Vars,T),
    absolve(Vars,roundrobin,T,
            0.125,10,allsmall(Vars,1.0e-6)).
```

This query finds a solution in 5 seconds after performing about 1.2 million primitive constraint contractions on a constraint set of about 7K primitive constraints.

4.2 Depth First Split Solving

Sometimes a constraint may have multiple solutions and a solver must use some sort of domain splitting to return each answer. The are several strategies that one might employ. The simplest is the `splitsolve` method [7], which repeatedly selects a variable, splits it into two or more parts and uses backtracking to look for solutions in each partition.

```
  splitsolve(Vars,Choose,Contract,Terminate) :-    Terminate,!.
```

```
splitsolve(Vars,Choose,Contract,Terminate) :-
  choosevar(Choose,V,Vars,NewVars),split(V),
  Contract,
  splitsolve(NewVars,Choose,Contract,Terminate).
split(V) :- midpoint(V,M), {V>=M}.
split(V) :- midpoint(V,M), {V<M}.
```

For example, to apply the Taylor contractor for the 20 variable Broyden example using a splitsolve search for possible solutions with 6 decimal digits of precision, we can pose the following query:

```
?- broyden(20,Vars,Eqns), make_contractor(taylor,Eqns,Vars,Taylor),
   forward_check(Vars,
      splitsolve(Vars,roundrobin,Taylor,allsmall(Vars,1.0e-6))).
```

This finds a solution box in 6 seconds after performing 1.7 million primitive narrowings on a constraint set containing about 7K primitive constraints on 6K variables.

4.3 Specifying compound contractions

The philosophy underlying our meta-level approach to constraint solving is that the particular constraint contraction operators (e.g. taylor, or absolve) can be written and verified by the system designer. The system user can then use these to construct arbitrarily complex compound contractors. Since each contractor is provably sound, the various compound contractors the user can build will only differ in how tightly they are able to enclose the solutions to the constraints. In this view, the user will always have a correct interval answer to the constraint, but the intervals may be too large to be useful. The problem solving task is then seen as combining contractors in such a way as to narrow the intervals to a useful width.

A disadvantage of the formalism presented so far is that it requires one to keep track of the lists of relevant variables. This detail can be hidden by having an interpreter perform that bookkeeping. For example, one can write a simple interpreter

```
solve(Solver) :-
  make_con(Solver,_,C),
  C.
make_con(taylor(E),V,T) :-
  get_vars_nodup(E,V),
  make_taylor_contractor(E,V,T).
...
make_con(iterate_until(T,N,U),V,iterate_until(C,N,U)) :-
  make_con(T,V,C).
...
```

where `get_vars_nodup(E,V)` is a metalevel predicate which returns a list `V` (without duplication) of the variables in the expression `E`. Given this framework, we can succinctly and declaratively specify a wide range of contraction strategies. For example, the following query

```
?- broyden(20,Vs,B),
   solve(absolve(cftaylor(B),0.10,20,1.0e-10)).
```

applies an approximation to the `NARROW_BOX_NE` operator of [39] to the 20 variable Broyden problem. It first constructs the centered form taylor contraction for the 20 variable Broyden problem, and then repeatedly applies an "absolve" contraction which attempts to trim 10% off of each end of the interval. This continues at most 20 times or until the interval widths are below 10^{-10}. More complex queries such as the following can also be made

```
?- broyden(20,Vs,B),
   solve(iterate( (absolve(cftaylor(B),0.1,5,1.0e-10),
                   taylor(B),2,1.0e-10),
                  10,1.0e-10)).
```

This iterates a compound contraction up to 10 times. The compound contraction first applies up to 5 iterations of the absolve strategy with the centered form contraction, and then applies a simple taylor contraction.

5 Performance Benchmarks

We evaluated the performance of CLIP's metalevel contractors on two standard benchmarks: the Broyden-banded example described above, and the Moré-Cosnard problem.

5.1 The Broyden-Banded problem

The Table in Fig. 4 shows several performance statistics for solving the Broyden-banded problem using the following CLIP query for various values of `N`.

```
?- broyden(N,Vs,Es),make_contractor(taylor,Es,Vs,T),
   iterate_until(T,10,allsmall(Vs,1.0e-6)).
```

Fig. 5 shows the performance data for several variants of the centered form contraction. For example, the third column of the row labelled "CFTaylor Iterate" data represents the timing for the following query:

```
?- broyden(20,Vs,Es),make_contractor(cftaylor,Es,Vs,T),
   iterate_until(T,10,allsmall(Vs,1.0e-6)).
```

The statistics gathered for the tables in Figures 4 and 5 are

- The `prim constraints` row lists the total number of primitive constraints generated during the execution of the query.
- The `variables` row lists the total number of interval variables created.
- The `prim contractions` row lists the total number of times that primitive contraction procedures were called during the execution of the query.
- The `CLIP CPU time` lists the total execution time of the query running on a 750MHz Athlon with 64MB memory running RedHat Linux 6.1.

5.2 The Moré-Cosnard problem

The Moré-Cosnard problem is to find the zeroes of the following set of n non-linear equations, where we let $m = n + 1$ and $k = 1, \ldots, m - 1$:

$$0 = x_k + \frac{1}{2m}\left((1 - \frac{k}{m})\sum_{j=1}^{k}\frac{j}{m}(x_j + 1 + \frac{j}{m})^3 \; + \frac{k}{m}\sum_{j=k+1}^{m-1}(1 - \frac{j}{m})(x_j + 1 + \frac{j}{m})\right)$$

In Figure 6 we show the performance of the four different CLIP queries applied to the Moré-Cosnard problem:

```
?- more(N,Xs,Es),solve(CONTRACTOR)
```

N	10	20	40	80
prim constraints	2720	6965	16955	43295
variables	2405	6032	14179	34314
prim contractions	0.6M	1.4M	2.6M	5.6M
CLIP CPU time	1.1s	2.7s	6.2s	16.3s

Figure 4: Taylor Contractions for the Broyden Example. CLIP CPU time is on a 750MHz Athlon with 64Mb of memory running RedHat Linux 6.1.

N	10	20	40	80
prim constraints	2053	6468	21538	77118
variables	1834	5769	19296	69340
prim contractions	0.4M	1.0M	1.8M	4.3M
CLIP CPU time	0.7s	1.8s	4.3s	14.7s

Figure 5: Centered Taylor Contractions for the Broyden Example CLIP CPU time is on a 750MHz Athlon with 64MB of memory running RedHat Linux 6.2.

N		5	10	20	40
CLIP Taylor	Absolve	5.50s	12.20s	62.30s	****
CLIP Taylor	Iterate	2.20s	14.0s	46.60s	****
CLIP CFTaylor	Absolve	1.40s	2.8s	7.30s	36.8s
CLIP CFTaylor	Iterate	0.65s	1.4s	4.67s	24.7s

Figure 6: Various Contractions for the Moré-Cosnard Example. CLIP used initial bounds $[-1, 0]$ on all variables. Numerica had initial bounds $[-10^8, 0]$. CLIP CPU time is on a 750MHz Athlon with 65MB of memory running RedHat Linux 6.2. The two Taylor contractor examples ran out of space for the N=40 case.

where CONTRACTOR is one of

```
  taylor absolve:  absolve(  taylor(Es),100,Eps)
  taylor iterate:  iterate(  taylor(Es),100,Eps)
cftaylor absolve:  absolve(cftaylor(Es),100,Eps)
cftaylor iterate:  iterate(cftaylor(Es),100,Eps)
```

for $N = 5, 10, 20, 40$, where `more(N,Xs,Es)` constructs the equations for the Moré-Cosnard problem of size N where all variables are initially $[-1, 0]$. For the $N = 40$ case, the `taylor` contractors ran out of memory while trying to solve the problem and hence the time entry is listed as "****". These memory failures are mostly likely due to the fact that the Prolog on which CLIP is implemented (GnuProlog) does not currently have a garbage collector. It is also interesting to note that the `absolve` contractor performed worse than just straight iteration on this example.

We have compared the performance of CLIP with published timings for Numerica [38] on the Broyden and Moré-Cosnard problems. The relative timings are shown in Figure 7 and demonstrate that CLIP performance is within a constant factor of Numerica performance over a wide range of problem sizes for these particular problems. The Numerica data is already several years old however, and so this data overestimates the relative speed of CLIP, nevertheless this data does suggest that CLIP and Numerica have the same asymptotic performance on these examples.

Numerica/CLIP CPU time

N	10	20	40	80
Broyden	0.71	0.88	0.97	0.67
More-Cosnard	0.29	0.28	0.28	0.30

Figure 7: Comparison of CLIP cftaylor and Numerica for the Broyden and Moré-Cosnard examples . CLIP CPU time is on a 750MHz Athlon with 64Mb of memory running RedHat Linux 6.1. Numerica CPU time is taken from [38] where the machine is unspecified.

---

6 The Implementation of CLIP

CLIP is implemented as a foreign file extension to Prolog. It consists of two parts:

- a Prolog program `clip.pl` which defines the constraint predicate `{}/1` and translates compound constraints to conjunctions of primitive constraints, and
- a C program `clip.c` which implements a simple interval arithmetic constraint solver. The Prolog interface code is about 1500 lines of Prolog (700 to translate constraints to primitives and 800 to implement various split solvers). The constraint engine is about 10000 lines of C code (6000 for the primitive narrowing and interval arithmetic and 4000 for the constraint solver).

In this section we discuss the design of these two programs with particular attention paid to simplifying the problem of verifying their correctness.

6.1 Prolog Interface code

CLIP currently interprets constraints by translating them, at runtime, into primitive constraints and invoking the constraint engine's solver. The constraint engine maintains its own choice stack and the interface code is responsible for sending commands to push and pop that stack appropriately. The toplevel predicates of `clip.pl` are shown in Fig. 8. The main idea is that a constraint {`C1,...,Cn`} is handled by pushing a choice point in the constraint stack (and popping it on backtracking), converting the constraints into primitive constraints which are stored in the constraint engine, and invoking the constraint engine narrowing loop. The loop succeeds (with `A=1`) or fails (with `A=0`), or overflows a resource bound (with `A=2`), in the latter case the execution is aborted.

The complete list of interface predicates between `clip.pl` and `clip.c` is in Fig. 9. The `clip_make_*` predicates are used for decomposing complex constraints into primitive constraints, while `clip_add_...` is used to add toplevel constraints to the constraint store. The `clip_set_*` and `clip_get_*` are used to set and get constraint engine parameters. The predicates `clip_narrow*` invoke the constraint solver loop. The remaining predicates are used to support split solving, and to query the engine about other parameters. For example, the constraint $X * \sin(Y) = Z$ would be translated into the following primitive constraint calls:

```
X='$INT'(A),Y='$INT'(B),Z='$INT'(C),
clip_make_interval(A),
clip_make_interval(B),
clip_make_interval(C),
clip_get_prim_constraint(SIN_opcode,B,0,D)
clip_add_prim_constraint(MULT_opcode,A,D,C).
```

Note that intervals are represented in the Prolog engine by terms of the form `$INT(N)`. The main point to observe here is that the Prolog front end `clip.pl` is a simple and straightforward translator to primitive

```
{C} :- set_cp,add_constraints(C),narrow.

set_cp :- clip_push_cp(C),reset_cp(C).
reset_cp(_).
reset_cp(C) :- clip_pop_cp(C),fail.
narrow:-clip_narrow(A),(A=1;(A=2,abort)).
```

Figure 8: Toplevel of `clip.pl`

constraints and the interface between Prolog and the constraint solver is also exceedingly simple and hence relatively easy to verify.

6.2 Constraint Engine in C

The core of the constraint engine is the `narrow_loop` which repeatedly

- takes a constraint off of the queue of active constraints,
- contracts the constraint and determines which of the variables have changed values significantly
- adds constraints to the active queue if they contain variables which have changed significantly

This loop is repeated until the queue is empty or unsatisfiability is detected.

The system also keeps track of how many primitive constraint narrowings have been computed since the last constraint was sent to the solver. When that number exceeds a user selected bound, `max_narrow`, the solver rapidly stops contracting without having reached a fixed point; typically this parameter is set at 1000. More precisely, when `max_narrow` is exceeded, the solver then continues contracting the constraints on the queue, but only adds new constraints to the queue if they contain variables whose ranges have been drastically reduced by the most recent contraction (e.g., if the size of the interval, measured by the number of floating point numbers it contains, is decreased by 50% or more). This has the effect of rapidly clearing out the queue of constraints, but at the same time guaranteeing that all simple propogation as described in Section 2.2 will continue to completion.

The engine can also be told to add all constraints to the active queue and run the narrow loop with a specified value N of `max_narrow` by using the `narrow_all(N)` command. Thus one useful strategy is to set the `max_narrow` parameter to a low value while the constraint set is being built up and then just before printing the answer, add all the constraints back into the queue and set `max_narrow` to a large value.

Again, this part of the system has been designed to be simple and straightforward so that it could be easily verified correct.

6.3 Primitive Constraint Contraction

The primitive constraint contractors (shown in Fig 10) are based on an interval arithmetic library which has been designed with careful attention to soundness. The underlying interval arithmetic algorithms have been proved correct [23, 22], and we are preparing a proof of the correctness of the contractors for the elementary functions (exp, log, sin, etc.).

Note that since our goal is to have a mathematically verifiable constraint solver, we were not able to use the standard math libraries that are distributed with standard C compilers as the numerical error in these libraries has not been proved to lie within any useful bounds. Thus, we had no choice but to rewrite the libraries from scratch. One approach would be to use some of the well-known published methods to build a math library which returns results that are correctly rounded to the last bit [41] or the last few bits [34].

```
void init(int n);
Bool clip_make_interval(int *a);
Bool clip_make_constant(double b,int *c);
Bool clip_make_integer_constant(int b,int *c);
Bool clip_make_quoted_constant(char* b,int *c);
Bool clip_get_prim_constraint(int I,
          int *op,int *a,int *b,int *c,int *N);
Bool clip_add_prim_constraint(int op,int a,int b,int c);
Bool clip_make_term(int op,int a,int b,int *c);
Bool clip_push_cp(int *a);
Bool clip_pop_cp(int a);
Bool clip_reset(int a);
Bool clip_get_bounds(int a,string *b,string *c);
Bool clip_set_bounds(int a,string b,string c);
Bool clip_get_bounds(int a,double *b,double *c);
Bool clip_set_bounds(int a,double b,double c);
Bool clip_set_float(int a,double b);
Bool clip_get_float(int a,double *b);
Bool clip_set_int(int a,int b);
Bool clip_get_int(int a,int *b);
Bool clip_narrow_all(int *succeeded);
Bool clip_narrow(int *succeeded);
Bool clip_midpoint(int a,char * *m);
Bool clip_split_hi(int a, char * m, int *c);
Bool clip_split_lo(int a, char * m, int *c);
Bool clip_small_interval(int a, double eps, int *c);
Bool clip_dump(int a);
Bool clip_print_var_dep(int a);
Bool clip_do_cut(int a);
```

Figure 9: C/Prolog Interface predicates for Constraint Engine

```
nar_add(X,Y,Z)   nar_mul(X,Y,Z)   nar_u_minus2(X,Y)
nar_exp2(X,Y)    nar_square2(X,Y)
nar_eq2(X,Y)     nar_lt2(X,Y)     nar_le2(X,Y)     nar_ne2(X,Y)
nar_integer1(X)  nar_bool1(X)     nar_abs2(X,Y)    nar_sgn2(X,Y)
nar_max(X,Y,Z)   nar_min(X,Y,Z)   nar_flr2(X,Y)    nar_ceil2(X,Y)
nar_or(X,Y,Z)    nar_and(X,Y,Z)   nar_xor(X,Y,Z)   nar_not2(X,Y)
nar_imp(X,Y,Z)   nar_if(X,Y,Z)
nar_sin2(X,Y)    nar_cos2(X,Y)    nar_tan2(X,Y)
nar_sin2pi2(X,Y) nar_cos2pi2(X,Y) nar_tan2pi2(X,Y)
nar_lessfn(X,Y,Z)nar_leqfn(X,Y,Z) nar_eqfn(X,Y,Z)  nar_subset2(X,Y)
nar_pow_even(X,Y,Z) nar_pow_odd(X,Y,Z)
```

Figure 10: Primitive narrowing routines used by the constraint engine

We opted instead to rewrite the standard numerical algorithms but using interval arithmetic techniques to get a provably correct result. Our main goal was to implement a numerical interval library which was sound, i.e. returned intervals which were guaranteed to contain the correct answer. Minimizing the width of the computed intervals was a secondary goal and doesn't affect the correctness of the system, thus we don't have to worry about the table maker's dilemma [28] which arises when one must compute the result rounded to the nearest floating point.

To give an example of how our approach works, we consider the simplest example, narrowing the exponential function. Since exp is a monotone increasing function, the contraction of the constraint $\exp(X) = Y$ (or equivalently $X = \log(Y)$) for intervals X and Y can be optimally computed using

$$X \leftarrow X \cap \log(Y) \quad \wedge \quad Y \leftarrow Y \cap \exp(X)$$

where

$$\exp([a,b]) \subseteq [\exp_{lo}(a), \exp_{hi}(b)] \ \wedge \ \log([a,b]) \subseteq [\log_{lo}(a), \log_{hi}(b)]$$

as was noted in the Introduction. Thus, it suffices to be able to compute sound upper and lower bounds on the values of exp and log on floating point numbers.

Our approach to computing a small interval containing $\exp(x)$ is based on the following Taylor formula with remainder which is valid for all $d > 0$ if $a \in [-\ln(2), \ln(2)]$:

$$\exp(a + n\ln(2)) \in 2^n * \left(\sum_{i=1}^{d-1} \frac{a^i}{i!} \ + \ [0.5, 2] * \frac{a^d}{d!} \right)$$

Thus, to compute $\exp(x)$ we must first compute an integer n and an interval a such that $x \in a + n * \ln(2)$ and $a \in [-\ln(2), \ln(2)]$. Maintaining accuracy in the reduced interval a requires using an extended precision representation of $\ln(2)$

$$\ln(2) \in b + B$$

where b is a floating point number and B is an interval, then we use interval arithmetic to compute

$$a = (x - (n * b)) - n * B$$

The Taylor series with remainder is then carefully evaluated (in a modified Horner form) so as to return a resulting interval with width at most 2 units in the last place (ULP).

The problem of implementing verifiably correct interval valued constraint contractors for elementary functions is an interesting and important one but a full treatment would take us too far afield from the main concerns of this paper. Nevertheless, by examining the table in Figure 10 one sees that most of the operators are for relatively simple operations ($\max(a, b)$ or boolean operators where $false/true$ are represented by 0/1 as in [6]). The only complex constraints are the arithmetic and elementary functions and hence the job of verifying the correctness of the CLIP system is again of manageable size.

6.4 Extending the constraint syntax

It is straightforward to add new constraints to CLIP by defining new constraints in terms of currently existing constraints, e.g. X/Y is actually defined in terms of multiplication by the following fact in the constraint parser:

```
function_def(A/B=X,(A=B*X)).
```

Similarly, raising a real to a rational power is defined in terms of two primitive constraints (evenpow(x,y) and oddpow(x,y)) which extend $\exp(y * \log(x))$ from the positive reals to the reals as either an even or odd function of x.

```
function_def(X^N=Z,evenpow(X,N)=Z) :-
  integer(N), (N mod 2) =:= 0.
function_def(X^N=Z,oddpow(X,N)=Z) :-
  integer(N), (N mod 2) =:= 1.
function_def(X^(A/B)=Z,(X^A=T,Z^B=T)) :-
  integer(A),integer(B),
  gcd(A,B,1),B\==0.
```

Extending the constraint syntax as described above can create subtle changes in the meaning of the constraints. For example, by defining A/B using multiplication, the constraint $A/0 = B$ is satisfiable and contracts A to 0. With a strict definition of division we would probably want this constraint to fail. However, this more relaxed semantics for division allows for one to make use of L'Hospital's rule to give values to functions defined by quotients where the numerator and denominator vanish. For example, we can define the relation for $y = (1 - \cos(x))/x^2$ by

```
c1(X,Y) :-  {Y = (1-cos(X))/X^2,
             Y = 1/2*(1 - X^2/12*(1 - X^2/30*(1 - X^2*[-1,1]/56)))}.
```

The second term in the constraint is the Taylor approximation to $(1 - \cos(x))/x^2$ and is valid for all x but only accurate near zero, and evaluates to 0.5 at zero while the constraint $Y = (1-\cos(0))/0^2$ is transformed to $0^2 * Y = (1 - cos(0))$ which is satisfiable but places no constraints on Y. If the strict definition of A/B had been used then the constraint $Y = (1 - \cos(0))/0^2$ would fail (as $Y = 0/0$ would have no solutions).

Another approach to adding primitive constraints, which is more complex, but still relatively straightforward, is to recompile the constraint solver after extending the constraint engine by adding a C procedure to the constraint engine (`clip.c`) to soundly contract the new constraint, and modifying a few lines in the constraint parser (`clip.pl`) to have the new contractor invoked from Prolog when the primitive constraint appears in a compound constraint.

7 CLIP semantics as Generalized Contraction

The theoretical foundation for using CLIP as a language for building meta-level contractors is based on the logical semantics of CLP(D) which underlies CLIP. In this section we show how this semantics allows one to view CLIP procedures as generalized contraction operators. The CLIP semantics are based on Lassez and Jaffar's general Constraint Logic Programming (CLP(D)) semantics over a constraint domain D [25]. After reviewing the syntax, semantics, and standard top down implementation of general CLP(D) programs we describe the semantics and implementation of CLP(Intervals).

Syntax of CLP(D) A constraint domain D is a first order theory T with equality together with an associated model M, such that T is the set of theorems of M. A CLP(D) program P is a sequence of rules of the form:

```
H := G1,...,Gn
```

where the head of the rule `H` is a standard Prolog goal and the `Gi` in the body are either Prolog goals or constraints. Prolog goals are terms of the form $g(e_1, \ldots, e_n)$ where the e_i are Prolog terms, where a Prolog term e is either a constant symbol or a variable or is a term of the form $f(e_1, \ldots, e_m)$ for Prolog terms e_i.

Constraints are terms of the form

```
{C1,...,Cm}
```

where the Ci are atoms of the constraint theory T.

A query Q has the same form as the body of a CLP rule:

```
G1,...,Gn
```

and we sometimes write it as $Q(X)$ where X denotes the vector of variables which appear in the `Gi`.

Operational Semantics of CLP(D) The standard operational semantics for CLP(D) generates a sequence of answer constraints for each query Q to a program P. The sequence is generated by creating a search tree whose nodes are conjunctions of constraints and goal atoms. The answer constraints correspond to the leaves of this tree (ordered in a depth first left-to-right traversal), which are themselves constraints. Infinite branches in the search tree correspond to infinite loops in the program. The search tree is constructed using the following rule:

> *Start with a tree containing the query at the root and repeatedly take the leftmost unprocesssed leaf node and apply the following transformation: Let the node have the form:*
>
> $$C1, C2, ..., Cr, G1, G2, ..., Gs$$
>
> *where the Ci are constraints and the Gi are goals. If the conjunction of the Ci is unsatisfiable, mark the node as a failure node and continue. Otherwise, for every rule*
>
> `H := B1,...,Bn`
>
> *in the program P where the predicate of H equals the predicate of $G1$, add the node*
>
> $$C1, C2, ..., Cr, (G1 = H'), B1', ..., Bn', G2, ..., Gs$$
>
> *where the prime denotes the fact that the variables of the rule have been renamed to be separate from the variables of the goal, and the term $G1 = H'$ denotes the conjunction of equations obtained by equating the corresponding arguments of the atoms $G1$ and H'.*

Logical Semantics of CLP(D) The logical interpretation of a CLP(D) program is given by a generalization of Clarke's negation by failure semantics to CLP. In this semantics, the set of all rules

```
p(Ti) := B1i,...,Bni
```

in the program where H has a given predicate symbol p are viewed as specifying a logical sentence

$$\forall X \;\; p(X) \Leftrightarrow \bigvee_i \exists V_i \;\; ((X = T_i) \wedge B_{i1} \wedge \ldots \wedge B_{in_i})$$

where V_i denotes the set of all variables in the ith rule. Applying this operation to all rules in the program P gives rise to a first order theory P^* called Clarke's completion of P. A fundamental result of Jaffar and Lassez [25] is that in the case where the search tree of a program P is finite one has

$$P^* \cup T \models \left(\forall X Q(X) \Leftrightarrow \bigvee_j \exists V_j \;\; C_j(X, V_j) \right)$$

where $(C_j(X, V_j))_{j=1,...,m}$ are the answer constraints for the query Q to the program P, i.e. they denote the set of leaves of the search tree. In particular, if the query fails finitely, i.e. has a finite search tree with no satisfiable leaves, then the proposition that $Q(X)$ has no solutions is a logical consequence of $P^* \cup T$:

$$P^* \cup T \models \neg \exists X \;\; Q(X)$$

and so Q is unsatisfiable in the model M of the domain D. (In fact, much more is known about the relationship between the operational and logical semantics of CLP(D), even in the case of negation, but for our purposes we need only the well-known results shown above).

Logical Semantics of CLP(Intervals) For CLP(Intervals), the constraint domain is a theory of the reals which includes a typically large set of predicate and function symbols representing all of the standard mathematical functions. There are two key ideas behind the CLP(Intervals) approach:

- the satisfiability tests for the real constraints are performed using an interval arithmetic constraint solving algorithm which maintains a real interval I_X for each variable X and uses the constraints to contract the intervals without removing any solutions to the constraints. If some interval becomes empty, then the constraint has been proved to be unsatisfiable.
- the answer constraint, although present in the heap of the constraint solver, is not presented to the user. Rather, the set of intervals for the toplevel variables is displayed. The interval that these variables represent corresponds to a box in euclidean space with the property that every solution to the particular answer constraint is contained in that box. Some information is lost by omitting the symbolic answer constraints, but the result is easier to understand. For example, although CLP(R) was only able to handle linear constraints, it would often return a symbolic formula representing a linear relation it had inferred from the constraints. CLP(Intervals) systems do not usually have this capability.

Generalized Contraction The semantics allows us to view a CLIP program as defining higher level predicates on the reals, and the interpreter can be used as a sound contractor for a generalized constraint language. Indeed, given a set X of variables, initially bound to $[-\infty, \infty]$, and a query $Q(X)$ to a program P, the interpreter contracts the intervals associated to X without removing any solutions to the query.

This simplicity of the CLIP semantics and the fact that it is defined in terms of contraction, greatly simplifies the implementation of sound contractors, since CLIP will automatically contract the intervals of variables in mathematical formulae, without removing any solutions.

The Forward Checking Contractor One particularly useful operation that can be implemented using a CLP intervals interpreter is the forward checking contractor for a CLP procedure q. This contractor takes a constraint $(X \in I)$, which is a shorthand for a tuple $X = (X_j)$ of variables with a tuple of constraints $(L_j \leq X_j) \wedge (X_j \leq U_j)$, and computes all answers constraints $X = I_k$ to the query

$$(X \in I) \wedge q(X)$$

It then returns the constraint $X = J$, where J is the smallest box that contains all I_k. If the query does not have a finite search tree, then this operation does not terminate. If, on the other hand, it does terminate, then we can conclude that the interval I can be contracted to the subinterval J without removing any solutions to q, formally:

$$P^* \cup T \models \forall X \quad (((X \in I) \wedge q(X)) \Rightarrow X \in J)$$

8 Future Work

Multivariate Taylor contractions. We have shown that the simple Taylor contractors can be implemented in a declarative style using CLIP. The convergence of multivariable Interval Newton contraction is widely reported to be greatly improved by preconditioning the equations using an approximate inverse to the Jacobian. In Newton/Numerica and in Interval Arithmetic algorithms, this involves proving the correctness of a multidimensional interval arithmetic function operating on sets of reals (see, e.g. [31], p. 177). Using CLIP, this contraction can be expressed as the following mathematical constraint, which holds whenever $f(x) = 0$:

$\forall a \in U$, $\forall B \in M_n$, $\exists t \in [0,1]^n$, $\exists \xi \in (\mathcal{R}^n)^n$ s.t.,

$$\begin{aligned} x &= a - (B \cdot f)(a) - M(\xi) \cdot (x - a) \\ \xi_i &= a + t_i * (x - a) \end{aligned}$$

$$\begin{aligned} M(\xi) &= B * Df(\xi) - I_n \\ Df(\xi) &= \left(\frac{\partial f_i}{\partial x_j}(\xi_i)\right) \end{aligned}$$

where M_n is the space of $n \times n$ real matrices, $I_n \in M_n$ is the $n \times n$ identity matrix, $B \cdot f$ represents the matrix B applied to the vector valued function f, $M(\xi) \cdot (x - a)$ is again a matrix applied to a vector, and $B * Df(\xi)$ represents a product of the matrices B and $Df(\xi)$.

We are currently extending CLIP to allow us to implement preconditioned multivariate contractors in the same declarative style. This will require adding vector and matrix operations to the CLIP constraint language, but the same approach used for the univariate contractor should extend to the multivariate case.

To get good convergence one generally lets a be the midpoint of x and selects B to be an approximate inverse to the Jacobian of f at a, then $M(\xi)$ is a matrix of small norm applied to an interval $x - a$ of small width. Thus the right hand side will generally have much smaller width than the left hand side (x) and so substantial contractions are often possible. Note however that this formula holds for any matrix B and any point $a \in U$ and correctness does not rely on any special properties of these parameters. Thus, this contraction formula is really a family of sound contractors.

In order to declaratively express more complex constraints we will also need to adopt a summation syntax similar to that used in Numerica. We also plan to look for a constraint version of the Moore-Penrose Newton contractor used in Cucker and Smale's sound constraint solver [9].

Existence Proofs. Another powerful feature of the Newton system is that it is able to prove that certain boxes contain solutions to a given constraint set. This type of proof requires more sophisticated techniques than we have used in this paper since it deals explicitly with properties of *sets* of reals, whereas to define contraction constraints we only needed to specify properties of real numbers themselves. We have developed a theory of "functional constraints" in CLIP in another context [18] and we are currently working on expressing existence proofs as functional constraint problems. Such an extension would allow CLIP to solve global optimization constraints as in Newton and Numerica.

Efficiency. The current CLIP implementation offers many opportunities for significant optimization at every level. At the lowest level, the primitive Interval Arithmetic operations and contractors are written in C, but since they make use of directed rounding operators, they cannot, to our knowledge, be safely compiled on any currently available C compiler with the C optimization switches set. There have been proposals to incorporate directed rounding into a dialect of Java (see, e.g., Joseph Darcy's Borneo project at Stanford), but for now the only way to attain efficient code containing directed rounding operations is hand compilation. At the level of translating constraint expressions into primitive constraints, the natural optimization would be to compile the constraints to assembly language just as Dìaz has done with the Finite Domain constraints in GNU Prolog.

At the level of implementing the high level constraint contractors in CLIP, no attempt was made at this stage to minimize the number of constraints generated. We have made a preliminary investigation into the use of a minimal sequence of Taylor contractions for solving particular constraints efficiently and to maximal accuracy [21]. It would be interesting to build in Lhomme's optimizations [29] into CLIP as well, perhaps at the metalevel. In particular, making use of symbolic algebra techniques to compute normal forms for the derivatives would probably both lessen the number of constraints generated and improve the precision of the computed intervals.

Acknowledgements

The author would like to thank Qun Ju for her contributions to the CLIP system and the three anonymous referees for their detailed and insightful comments on interval arithmetic and constraint solving. Their efforts have led to a much improved paper.

References

[1] F.S. Acton, Real computing made real: Preventing Errors in Scientific and Engineering calculations, Princeton University Press, Princeton, New Jersey 1996.

[2] Applied Logic Systems. CLP(BNR) manual. `www.als.com` 1999.

[3] F. Benhamou and D. McAllister and P. Van Hentenryck CLP(Intervals) Revisited Proceedings of ILPS'94, 1994.

[4] F. Benhamou and L. Granvilliers. Automatic Generation of Numerical Redundancies for Non-Linear Constraint Solving. Reliable Computing, vol 3(3):335-344, 1997. Kluwer Academic Publishers.

[5] M. Berz and G. Hoffstätter. Computation and Application of Taylor Polynomials with Interval Remainder Bounds. Reliable Computing, 1998.

[6] F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer, and Boolean constraints. *Journal of Logic Programming*, 32:1–24, 1997.

[7] Bell Northern Research. CLP(BNR) Reference and User Manuals. 1988

[8] P. Codognet and D. Dìaz. wamcc: Compiling Prolog to C. International Conference on Logic Programming, 1995.

[9] F. Cucker, and S. Smale Complexity Estimates Depending on Condition and Round-Off Error, Journal of the ACM, 46:1, pp. 113-184, 1999.

[10] Y. Deville and M. Janssen and P. Van Hentenryck Consistency Techniques in Ordinary Differential Equations. In M. Maher and F-F. Puget, editors, Principles and Practice of Constraint Programming (CP98), Lecture Notes in COmputer Science, Vol. 1520, pages 162-176, Springer-Verlag, 1998.

[11] IC Parc. RIA: ECLiPSe Real Number Interval Arithmetic. In the ECLiPSE Library Manual. 2000.

[12] `fdlibm` The Freely Distributable Mathematics Library.
`http://netlib.bell-labs.com/netlib/fdlibm/`

[13] F. Goualard, F. Benhamou and L. Granvilliers. An Extension of the WAM for Hybrid Interval Solvers. Journal of Functional and Logic Programming, 1999.

[14] D. Dìaz. GNU Prolog. `www.gnu.org/software/prolog` 1999.

[15] L. Granvilliers. A Symbolic-Numerical Branch and Prune Algorithm for Solving Non-linear Polynomial Systems. Journal of Universal Computer Science vol. 4(2):125-146, 1998. Springer Science Online.

[16] T. Hickey, CLP(F) and Constrained ODEs, in the Workshop on Constraint Languages and their use in Problem Modeling, ECRC Tech. Report, 1994.

[17] T. Hickey, CLIP: an implementation of CLP (Intervals),
`http://interval.sourceforge.net/prolog`, 1999.

[18] T. Hickey, Analytic Constraint Solving and Interval Arithmetic, Proceedings of POPL'00, Boston, MA, Jan. 2000.

[19] T. Hickey, CLIP: a CLP(Intervals) Dialect for Metalevel Constraint Solving, Proceedings of PADL'00. Springer-Verlag, "Lecture Notes in Computer Science", vol. 1753, 2000.

[20] T. Hickey, Z. Qiu, M.H. van Emden. Interval Constraint Plotting for Interactive Visual Exploration of Implicitly Defined Relations. Journal of Reliable Computing Vol. 6, No. 1, 2000, pp. 81-92.

[21] T. Hickey and D. Wittenberg, Validated Constraint Compilation, Brandeis University, Tech Rep. CS-99-201, April, 1999.

[22] T. Hickey, H. Wu, and M.H. van Emden, A Unified Framework for Interval Constraints and Interval Arithmetic, in Principles and Practice of Constraint Programming – CP98, M. Maher and J-F. Puget (eds.), Springer-Verlag, LNCS v. 1520, pp. 250-264, 1998.

[23] T. Hickey, Q. Ju, and M.H. van Emden, Interval Arithmetic: from Principles to Implementation, to appear in the Journal of the ACM.

[24] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *Numerical Toolbox for Verified Computing I.* Springer-Verlag, 1993.

[25] J. Jaffar and J.L. Lassez, Constraint Logic Programming. in Proceedings of the 14th ACM Symposium on the Principles of Programming Languages, 1987.

[26] R. Klatte, U. Kulisch, A. Wiethoff, C. Lawo, M. Rauch. C-XSC - A C++ Class Library for Extended Scientific Computing. Springer-Verlag, Heidelberg (ISBN 3-540-56328-8), New York (ISBN 0-387-56328-8), 1993.

[27] J. Lee and T. Lee. A WAM-Based Abstract Machine for Interval Constraint Logic Programming. IEEE International Conference on Tools with Artificial Intelligence, 1994.

[28] V. Lefèvre, J.-M. Muller and A. Tisserand. The Table Maker's Dilemma. IEEE Transactions on Computers, 1998.

[29] O. Lhomme, A. Gotlieb and M. Rueher. Dynamic Optimization of Interval Narrowing Algorithms. Journal of Logic Programming, 1998.

[30] R.E. Moore, Interval Analysis. Prentice-Hall, 1966.

[31] A. Neumaier. Interval Methods for Systems of Equations. Cambridge University Press. 1990.

[32] W. Older and F. Benhamou. Programming in CLP(BNR). Proceedings of PPCP'93, 1993.

[33] W. Older and A. Vellino, Constraint Arithmetic on Real Intervals, in Constraint Logic Programming: Selected Research. Colmerauer, A. and Benhamou, F. (eds), MIT Press 1993.

[34] D. Priest Fast Table-Driven Algorithms for Interval Elementary Functions ARITH13, Asilomar, CA, 1997.

[35] PrologIA. Prolog IV Constraints Inside. 1996.

[36] J.-F. Puget and M. Leconte. Beyond the Glass Box: Constraints as Objects. Proceedings of International Logic Programming Symposium, 1995.

[37] Sun Microsystems, Inc. Interval Arithmetic Programming Reference. Sun WorkShop 6 Fortran 95 Part No. 806-3595-10 May 2000, Revision A.

[38] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: A Modeling Language for Global Optimization.* MIT Press, 1997.

[39] P. Van Hentenryck, D. McAllester, D. Kapur. *Solving Polynomial Systems Using a Branch and Prune Approach*, SIAM Journal on Numerical Analysis, 34(2), 1997.

[40] P. Van Hentenryck, L. Michel and F. Benhamou. Newton - Constraint Programming over Nonlinear Constraints. Science of Computer Programming, 1998.

[41] A. Ziv, Fast Evaluation of Elementary Mathematical Functions with Correctly Rounded Last Bit, ACM Transactions on Mathematical Software, Vol. 17, No. 3, September 1991.