CLIP: A CLP(Intervals) dialect for Metalevel Constraint Solving

Timothy J. Hickey

Michtom School of Computer Science, Brandeis University, USA,
tim@cs.brandeis.edu

Abstract. CLIP is an implementation of CLP(Intervals) built on top of Prolog. It has been developed using the simplest model of constraint solving in which constraints are decomposed into sets of primitive constraints and a simple constraint queue is used to repeatedly apply primitive constraint contractions until some termination condition applies (e.g. no significant change occurs, or a preset limit on the number of contractions is reached). The simple semantics and implementation make it relatively straightforward to prove that the system is sound. In this paper we show how to implement more complex constraint solvers in CLIP. 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.

1 Introduction

Historically, there have been two approaches to implementing Interval Arithmetic constraint solvers, represented by the two systems: CLP(BNR) and Newton. In CLP(BNR), [5, 21, 20, 4, 2], 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 contract each primitive constraint until some termination condition is satisfied. In Newton, Numerica, and similar systems [3, 23, 22, 9, 8] the constraint solver is much more powerful than the underlying CLP(BNR) solver, but in some cases it can take hours to solve a single constraint.

We argue the merits of the CLP(BNR) approach where one first implements a fast CLP(BNR) system and then builds a Newton/Numerica-style constraint solver in this system. The main benefits of such a two step approach are

- it is relatively straightforward to provide a convincing proof of the correctness of the combined system, whereas proving the correctness of a Newton solver implemented in traditional languages such as C or Java would be much more difficult

– new solvers can be rapidly prototyped since the necessary code is much simpler and more concise in CLP(Intervals) than in imperative or functional languages (e.g. C, Java, Scheme).
– 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.

We have successfully used this approach to build constraint solvers for a large class of Ordinary Differential Equation (ODE) constraints [10, 12]. Here we show that some of the Newton/Numerica style contractors can be easily and efficiently implemented in a CLP(Intervals) system.

In this paper, we first describe our implementation of CLIP, a CLP(Intervals) system using a simple queue method for applying sound primitive constraint contractors [11]. We then give some benchmarks on its performance, illustrate the use of CLIP in implementing Newton/Numerica contractors with several concrete examples. Finally, we discuss directions for future research.

2 CLIP syntax and semantics

Syntactically, CLIP is an extension of GNU prolog [7] 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.

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. CLIP is designed so that the solver is powerful enough to be able to always solve constraints that require only simple propagation, e.g.,

```
 ?- {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...
```

Moreover, the solver will 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 [13] 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. These parameter are discussed in more detail below. There is also (quite importantly) a method to add all constraints back onto the queue and reinvoke the constraint solver. This guarantees that constraint narrowing which was prematurely halted by the constraint engine can be resumed at a later time.

```
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)
```

Fig. 1. CLIP constraint language

2.1 I/O syntax

The difference between base 2 and base 10 representations of numbers requires most decimal constants to be represented by non-point floating point intervals. 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}.
```

If X were represented by the nearest floating point number to 1.1, then the query would fail because $Z = (X - 1) * 10 - 1$ would be about $2^{-50} \neq 0$, when clearly it should succeed if the semantics are really given by the real numbers. To avoid these problems, we follow CLP(BNR)[2, 5] 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 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`} or {`X = '1.1'`} will bind `X` to the smallest interval with floating point bounds which contains X.
- {`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.
- {`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)`.

Similar conventions are used for printing intervals.

2.2 CLIP semantics as Generalized Contraction

Let P be a CLIP program and let $Q(X)$ be a CLIP query (where X is a vector of interval variables). Suppose the CLIP interpreter returns with a finite number of interval boxes $B_1, B_2, \ldots, B_k$, and then fails. The soundness of the general CLP semantics [17] implies that

$$\mathcal{T} \cup P^* \models \forall x. \left(Q(x) \Rightarrow x \in \bigcup_i B_i \right)$$

where P^* is Clark's completion of P and $\mathcal{T}$ is the theory of real constraints (i.e. $\mathcal{T}$ is the set of all first order sentences in the CLIP constraint language which are true for the real numbers.) Thus, the set $\bigcup_i B_i$ contains all solutions of the query $Q(x)$. It may not be the case, however, that each box B_i contains some solution of Q.

This semantic property of CLIP allows us to view a CLIP interpreter 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.

3 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 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 for the primitive narrowing and interval arithmetic and 4000 for the constraint solver).

3.1 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. 2. 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.

```
{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)).
!(G) :- G,!.
```

Fig. 2. Toplevel of `clip.pl`

3.2 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 constraints have been narrowed and when that number exceeds a user selected bound, the definition of significant change is made much more stringent so that the queue will rapidly be emptied. Typically this `max_narrow` parameter is set at 1000. The engine can also be told to add all constraints to the active queue. 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.

The primitive constraint contractors (shown in Fig 3) 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 [15, 14], and we are preparing a proof of the correctness of the contractors for the special functions (exp, log, sin, etc.).

```
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)
```

Fig. 3. Primitive narrowing routines used by the constraint engine

4 Higher level contractors

Next we look at the problem of implementing the Newton contraction algorithms in CLIP. We will examine closely the Taylor contractor and will describe not only its implementation but its correctness proof as well. 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.

4.1 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, continuously differentiable on an interval I must satisfy the following property:

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

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 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

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

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.

An example Consider the case of the Broyden Banded Function example (the first example cited in [23]) which is to solve the following system of n equations in $\{x_1, \ldots, x_n\}$

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

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.

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

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 below. 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.

Invoking Contractions The CLIP implementation of simple Taylor contractions described below 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 20 variable instance of the Broyden problem we would give the following query:

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

where the predicate `broyden(N,Vs,Es)` constructs (symbolically) the Broyden system `Es` on the `N` variables `Vs`, the `make_contractor` predicate creates a contractor as described below, and the `iterate(T,N,C)` operator calls T at most N times or until C is true.

The CLIP system solves this 20 variable Broyden problem and returns an answer for `Vs` with 6 decimal digits of accuracy in about 4 seconds (on a 500 Mhz Gateway 2000). The Taylor contractor `T` gets transformed into about 7K primitive constraints on 6K variables. A total of about 1.1 million primitive constraint narrowings are invoked. We suspect that the 7K primitive constraints could be significantly reduced by using a smarter differentiation routine and by making use of common subexpression optimizations. Fig. 4 shows the statistics for various values of N. We also include the Numerica times, as listed in [22] for some unspecified machine, for solving the Broyden problems.

It is interesting to note that this CLIP query is able to solve the Broyden problem using only Taylor contraction. In [23], the authors state that Interval Newton alone is not able to solve this problem, and they use this to argue the importance of the (fairly complex) box-consistency operators they employ. It appears that the Taylor constraint method offers an alternative approach, which at least for this problem is simpler and comparably efficient. In the final remarks we describe various strategies for improving the performance of the Taylor (and other) contractors.

Implementing Contractions We will now show how to define 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

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

N	5	10	20	40	80
prim constraints	851	2720	6905	16835	42815
variables	756	2415	5998	14109	34001
prim contractions	0.4M	0.6M	1.1M	2.2M	4.6M
CPU time	1.0s	1.9s	3.9s	9.4s	25.3s
Numerica CPU time	--	0.5s	1.6s	4.2s	9.8s

Fig. 4. Taylor Contractions for the Broyden Example

for each equation in $f = 0$ 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. 5 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. Observe that the derivative DF is computed both symbolically (as a solution to a symbolic set of equations) and numerically (as an interval resulting from the contraction of those equations). This versatile approach allows one to solve $F'(X) = 0$ for X using the same `diff` predicate. 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.

```
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).

taylor([X],[A],C) :-
  forward_check([X],(midpoint(X,M),{A=M,C})).
```

Fig. 5. Implementation of the Taylor Contraction

4.2 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:
$\forall a, x \in U, \exists t \in [0,1], \exists \xi \in U$:

$$f(x) = f(a) + Df(\xi) \cdot (x - a)$$
$$\xi = a + t * (x - a), \;\; t \in [0,1]$$

where $Df(z) = \left(\frac{\partial f}{\partial x_j}(z)\right)$ is the 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.

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

The performance data for the centered form contraction is shown in Fig. 6. Again, we have included the Numerica times, as listed in [22] for some unspecified machine, for solving the Broyden problems. The current implementation is completely unoptimized, and we suspect that the time and space requirements will decrease significantly after applying some simple optimizations.

N	5	10	20	40	80
prim constraints	678	2083	6528	21658	77118
variables	609	1864	5829	19416	69393
prim contractions	0.2M	0.4M	1.0M	1.9M	4.1M
CPU time	0.5s	1.2s	3.2s	7.9s	24.9s
Numerica CPU time	---	0.5s	1.6s	4.2s	9.8s

Fig. 6. Centered Taylor Contractions for the Broyden Example

4.3 Absolving

We are currently implementing several other contractors described in [23], including `NARROW_BOX_XX`, with `XX` in {`NE, DE, TE`}. Each of these contractors makes use of a well-known general contraction strategy which has gone by many names. It is the main idea behind box-consistency of Newton/Numerica, and has been called the squash algorithm in the RIA solver of the ECLiPSe CLP system, and was called **absolving** in early implementations of CLP(BNR) [20]. For the purpose of this paper, we will call this method **absolving**.

The `absolve` method contracts the interval for a variable by removing small subsets of the current solution interval near the endpoints in which it can prove there are no solutions. This algorithm can be easily expressed in CLIP as shown in Fig. 7.

```
% 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_chk(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}.
```

Fig. 7. Implementing Absolve

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 and 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.

5 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 [5], 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).
choosevar(roundrobin,Vs,Ws) :- dequeue(V,Vs,Tmp),enqueue(V,Tmp,Ws).
split(V) :- midpoint(V,M), {V>=M}.
split(V) :- midpoint(V,M), {V<M}.
dequeue(V,[V|Vs],Vs).
enqueue(V,Vs,Ws) :- append(Vs,[V],Ws).
```

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.

6 Specifying compound contractions

We are currently developing a language for specifying compound contractions. 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 [23] 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.

7 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. [19], 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.,

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

where M_n is the space of $n \times n$ real matrices, and $I_n \in M_n$ is the $n \times n$ identity matrix.

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. One interesting approach would be to use the finite domain solver of GNU prolog to generate the indices for complex summations which would then be evaluated in CLIP. Adding such a syntax for expression summation in which the indices would be specified and determined using CLP(FD) could be an interesting and powerful feature. Newton and Numerica both have a summation syntax, but to our knowledge they do not use a solver as sophisticated as CLP(FD) to evaluate the indices.

We also plan to look for a constraint version of the Moore-Penrose Newton contractor used in Cucker and Smale's sound constraint solver [6].

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 [12] 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 are plans 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 [13]. 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 underlying interval arithmetic routines.

References

1. Forman 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. Frédéric Benhamou and David McAllister and Pascal Van Hentenryck CLP(Intervals) Revisited Proceedings of ILPS'94, 1994.
4. Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and Boolean constraints. *Journal of Logic Programming*, 32:1–24, 1997.
5. Bell Northern Research. CLP(BNR) Reference and User Manuals. 1988
6. Cucker, F. and Smale, S. Complexity Estimates Depending on Condition and Round-Off Error, Journal of the ACM, 46:1, pp. 113-184, 1999.
7. Daniel Dìaz. GNU Prolog. `www.gnu.org/software/prolog` 1999.
8. 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.
9. 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.
10. T. Hickey, CLP(F) and Constrained ODEs, in the Workshop on Constraint Languages and their use in Problem Modeling, ECRC Tech. Report (defunct), 1994.
11. T. Hickey, CLIP: an implementation of CLP (Intervals), `www.cs.brandeis.edu/~tim/clip`, 1999.
12. T. Hickey, Analytic Constraint Solving and Interval Arithmetic, to appear in the Proceedings of POPL'00, Boston, MA, Jan. 2000.
13. T. Hickey and D. Wittenberg, Validated Constraint Compilation Brandeis University, Tech Rep. CS-99-201, April, 1999.
14. 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.
15. T. Hickey, Q. Ju, and M.H. van Emden, Interval Arithmetic: from Principles to Implementation, Brandeis University Tech Report. CS-99-202, July, 1999.
16. R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *Numerical Toolbox for Verified Computing I*. Springer-Verlag, 1993.
17. Jaffar, J. and Lassez, J. L., Constraint Logic Programming. in Proceedings of the 14th ACM Symposium on the Principles of Programming Languages, 1987.
18. Moore, R. E., Interval Analysis. Prentice-Hall, 1966.
19. A. Neumaier. Interval Methods for Systems of Equations. Cambridge University Press. 1990.
20. William Older and Frédéric Benhamou. Programming in CLP(BNR). Proceedings of PPCP'93, 1993.
21. Older, W. and Vellino, A., Constraint Arithmetic on Real Intervals, in Constraint Logic Programming: Selected Research. Colmerauer, A. and Benhamou, F. (eds), MIT Press 1993.
22. Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: A Modeling Language for Global Optimization*. MIT Press, 1997.
23. Pascal Van Hentenryck, David McAllester, Dipak Kapur. *Solving Polynomial Systems Using a Branch and Prune Approach*, SIAM Journal on Numerical Analysis, 34(2), 1997.