

Notes on using CLIP

David K. Wittenberg and Timothy J. Hickey

April 2, 2004

Abstract

This is a start at documenting CLIP. Eventually this will grow into a manual. CLP(I) (Constraint Logic Programming over Intervals) is a constraint logic programming (CLP) language whose domain is the real numbers (Moore's Interval Arithmetic). CLP(F) is a CLP language whose domain is analytic functions over the reals. CLP(F) is written in CLP(I). CLIP is an implementation of CLP(I) built on top of GNU Prolog. The current distribution of CLIP includes both CLP(I) and CLP(F) by default.

Contents

1	Introduction	3
2	Using CLIP	4
2.1	Multiple Solutions and Non-determinism	5
2.2	Analytic constraints and ODEs	6
2.3	Complex ODEs	7
3	General Theory of CLP Constraint Domains	8
4	How Clip Works	9
5	The CLP(I) Constraint Domain	10
6	The CLP(F) Constraint Domain	11
7	CLIP Predicates and Commands	14
7.1	Command Descriptions	14
8	Implementation	18
8.1	CLP(I) constraint solving	18
8.2	Primitive Contraction	19
8.3	CLP(F) constraint solving	19
9	Known Bugs and Issues	22
9.1	Unification and Constraint Solving	22
9.2	Other bugs and issues	23
10	Quick Reference	25
10.1	CLIP commands	25
10.2	CLP(I) predicates and functions	26
10.3	CLP(F) predicates and functions	27

1 Introduction

CLP(I) is an interval-based constraint logic programming (CLP) language whose domain is the set of real numbers. The class of CLP languages (and their syntax and semantics) was introduced by Jaffar and Lassez in 1987 [JL87]. Jaffar and Maher provide an excellent survey [JM94] of the fundamental concepts of CLP. The idea of calculating over intervals of reals comes from Moore’s 1966 book on Interval Arithmetic [Moo66].

The idea of combining CLP and Interval Arithmetic was first conceived by Cleary [Cle87] but the first production quality CLP(I) interpreter was the BNR-Prolog system developed by Older, Vellino, and Benhamou [Res88], [BO97],[OV93]. BNR Prolog was designed to be verifiably correct in the sense that the intervals it returned were mathematically guaranteed to contain all solutions to the underlying arithmetic constraints. The system however was proprietary and the underlying algorithms were never published in the scientific literature.

CLIP was originally developed as an open source implementation of CLP(I) by Qun Ju and Tim Hickey [HJ], [HJ97] and was used in Qun Ju’s thesis [Ju98] as the foundation for a parallel implementation of CLP(I). CLIP has subsequently been extended by Tim Hickey, who added the CLP(F) language, which provides constraints over functions, plotting, and some other refinements. David Wittenberg has made minor changes and fixed some bugs.

CLIP is built on top of Prolog [Pro95], [DEDC96], and currently runs on GNU Prolog [Dia02] and ALS Prolog. The fundamental philosophy is to have a relatively small base of sound primitive constraint contractors which are simple enough so that one can argue convincingly, if not formally prove, that they are correct, and then build more complex solvers on top of the proven system. Since the complex solvers built on CLIP primitives are made up of sound simple solvers, they are also sound. An important feature of CLP languages is that they are theorem provers, so that each answer generated by a CLP program has a direct interpretation as a theorem about the underlying domain.

In CLIP, introducing new constraints is usually a matter of narrowing the interval corresponding to one or more variables. A poor choice of order of contractions will result in poor performance (either by taking a long time or by providing very loose bounds), but will not result in an unsound computation. Writing more complex solvers directly results in systems which are more complex, which makes it harder to construct a direct argument for their correctness.

CLIP currently runs on Linux on x386 architectures and Mac OS X on PPC architecture, using IEEE754 floating point [IEE85] using the “double” (64 bit) word length. Porting to other architectures should be simple, but using a larger floating point word would take a lot of (straightforward) work.

CLIP is an extension to Prolog, so all commands available in Prolog are available in CLIP. Prolog commands are distinguished from CLP(I) commands by the brackets used to enclose them. CLIP commands are written { *command* }. CLP(I) itself can then be extended.

The CLP(F) [Hic00] [Hic01] language introduces function variables in addition to real variables and has proved to be a useful tool for studying systems

defined partly by ODEs (Ordinary Differential Equations). The CLP(F) extensions are now a standard part of CLIP. (It is implemented as a metalevel interpreter in the CLP(I) language.) Constraints on functions and their derivatives are written $\{[C1, C2, \dots, Cn]\}$. This document describes CLIP, which implements CLP(I) and CLP(F). We attempt to label CLP(F) extensions as such. Two other extensions are also described – plotting.pl and solve.pl

CLIP can be considered to be a constraint engine over intervals and functions which interfaces to the Prolog engine (a constraint solver over general finite domains).

The CLP(F) language solves analytic constraints by soundly approximating sufficiently differentiable functions by power series with remainder terms and introducing arithmetic constraints among the Taylor coefficients of the functions at the endpoints, at points in the interval, and over the entire range.

CLIP follows the Prolog convention of using lower case identifiers for atoms and upper case for variables.

Note that CLIP follows the US convention of using a period to represent the decimal point, so that one tenth is written 0.1

This manual documents the most important features of CLIP, but currently omits a few features including vectors in CLP(F), the extended syntax for input constants in CLP(I), and the plotting libraries.

2 Using CLIP

In CLP(F) the constraint domain allows one to declare variables representing various analytic values including:

- *real numbers, X*
- *infinitely differentiable functions, F , on a finite interval $[a,b]$*
- *vectors of numbers, functions, or vectors*

The CLP(I) constraint language allows one to express any algebraic equality or inequality constraint among its variables. For example, the query:

```
| ?- {X^2=2,X>0}.
```

returns

```
X = 1.41421356237309... ? ;
```

```
no
```

```
| ?-
```

The CLIP interpreter represents the interval for X in a compact form. The ellipsis “...” indicates that all shown digits are correct and hence X must lie in the interval:

```
[1.41421356237309,
 1.41421356237310)
```

Also, note the standard Prolog feature that the user entered a semi-colon after the solution and the interpreter responded with “no” which indicates that there are no more solutions.

2.1 Multiple Solutions and Non-determinism

Sometimes there may be more than one solution to a given constraint. The constraint solver will indicate this by returning an interval that contains all solutions:

```
| ?- {X^2=2}.
X = [-1.41421356237309536751922678377,
      1.41421356237309536751922678377] ?
no
| ?-
```

The user must guess whether this is because the system has failed to narrow the interval around one solution or if this is a case where there are multiple solutions. Hickey and Wittenberg [HW99a] discuss methods of determining which is the case, and choosing an appropriate method to use in `solve_clip`, but this work is clearly incomplete. There is a moderately sophisticated solver `solve_clip(METHOD,VARS,N)` which allows one to specify the solving method, (See section 7.1 for a list of supported solving methods, and a description of each of them.) the list of variables that should be solved, and a parameter N representing how much work should be done (e.g. a maximum allowed width for intervals, or a maximum depth for a divide and conquer splitting routine).

Here, to find the discrete set of solutions one must apply a divide-and-conquer approach where one divides the interval into subintervals and searches for solutions in each one. This is done using the “queue” method of the `solve_clip` solver and typing a semicolon after each solution that it finds:

```
| ?- {X^2=2},solve_clip(queue,[X],0.000001).
X = 1.41421356237309... ? ;
X = -1.41421356237309... ?
(10 ms) no
| ?-
```

The “no” answer at the end indicates that there are no more solutions to that query.

The constraint language for real variables allows any equations and inequalities constructed using the arithmetic operators and the standard mathematical functions (sin, cos, tan, asin, acos, atan, $\exp(e^x)$, log, exponentiation, integral powers X^Y , and others).

For example, the following query demonstrates the use of the forward checking solver `fwchk`, which divides the domain into a set of K boxes (initially $K = 1$) and in each step it divides each box into 2^V sub-boxes, where V is the number of variables in `VARS`, and applies the default narrowing procedure. Any

boxes that are proved to contain no solutions are discarded and the result is returned as the smallest box containing all of the remaining boxes. An example of the narrowing done by this method to solve $x^x = 1 + \cos(x) \wedge x > 0$ is:

```
| ?- {X**X = 1+cos(X), X>0},solve_clip(fwchk,[X],N).
N = 0 X = REAL(0,inf) ? ;
N = 1 X = 1.247504654353... ? ;
N = 2 X = 1.24750465435333... ? ;
N = 3 X = 1.24750465435333... ?
(1720 ms) yes
| ?-
```

Here N is the number of steps taken.

2.2 Analytic constraints and ODEs

CLP(F) also allows one to constrain functions by functional equations involving many of the same arithmetic operators and mathematical functions as CLP(I) supports. In addition, one can constrain a function or its derivatives to take specific values at specific points and to have a range that lies within an interval.

Consider the following mathematical constraint Q on the function variable F and real variables A and E :

$$Q(F, A, E) \equiv (F \in \mathcal{H}([0, 1]), F' = F, F([0, 1]) \subseteq [-100, 100], F(0) = 1, F(A) = 2, F(1) = E)$$

Q can be represented and solved by presenting the following constraint to the CLP(F) interpreter:

```
| ?- type([F],function(0,1)), {[ ddt(F,1)=F, F in [-100,100],
eval(F,0)=1,eval(F,A)=2, eval(F,1)=E ]}.
```

where the `type` predicate indicates that $F \in \mathcal{H}([0, 1])$, i.e., F is an analytic function in some open neighborhood of the interval $[0, 1]$. The output given by CLP(F) after 0.76 seconds on a 1 GHZ Mac TiBook is

```
A = 0.6931471... E = 2.7182818... ;
(760 ms) no
| ?-
```

which represents the following answer constraint:

$$C(F, A, E) \equiv (A \in [0.6931471, 0.6931472] \wedge E \in [2.7182818, 2.7182819])$$

The soundness of the CLIP interpreter implies that it has proven a theorem about the query and its solution constraint:

$$\forall F, A, E \quad Q(F, A, E) \Rightarrow C(F, A, E)$$

$$\begin{aligned}
Q(a, b, c, d, k_1, k_2, k_3, k_4, t_1, t_2) \equiv & \\
\exists f_1, f_2 \quad & I = [t_0, t_1], f_1, f_2 \in \mathcal{H}([t_1, t_2]), \\
& E = 0.0000000001, k_3 < x_{20}, \\
& f'_1 = k_1 - k_2 \sqrt{f_1 - f_2 + k_3} \\
& f'_2 = k_2 \sqrt{f_1 - f_2 + k_3} - k_4 \sqrt{f_2} \\
& f_1(t_0) = a, f_1(t_1) = b, f_2(t_0) = c, f_2(t_1) = d, \\
& f_1([t_1, t_2]) \subseteq [E, 1000], f_2([t_1, t_2]) \subseteq [E, 1000],
\end{aligned}$$

Figure 1: A complex non-linear ODE constraint

In other words, if F , A , and E represent a solution to Q , then they must satisfy the answer constraint C . Note that one cannot infer from this theorem that Q has any solutions at all. In this particular case, Q clearly does have a solution

$$F(t) = \exp(t), \quad A = \ln(2), \quad E = e$$

which of course satisfies the answer constraint C .

The function F is then constrained to be equal to its first derivative, and to take the value 1 at 0 and to take values in $[-1000, 1000]$ for all $x \in [0, 1]$. The variables A and E are not declared to be functions and hence are real constants by default. They are constrained so that $F(A) = 2$ and $F(1) = E$. The constraint solver finds A and E to 7 decimal digits of precision and also finds an interval for F not shown here, that specifies intervals for its first 10 derivatives at 0 and 1, and for the range of its first 10 derivatives over $[0, 1]$. The number of derivatives (10) can be set to any value N (but space and time complexity grows quadratically with N).

2.3 Complex ODEs

The CLP(F) solver can also handle very complex non-linear differential equations as it based on a “brute force” reduction of the analytic constraints into arithmetic constraints which are solved with a simple interval arithmetic constraint solver. For example, we [HW03] model a system consisting of a fluid with temperature $A(t)$ which is heated by a heating element whose temperature $B(t)$ has a non-linear component $\sin(B(t))$ in its defining ODE. This system is modeled by the following procedure:

```

ode2((T0,A0),[I,[Alpha,Beta,Gamma,Delta]],A,(T1,A1)) :-
  type([A,B],function(O,I)),
  {[ ddt(A,1) = Alpha * A + Beta + Gamma*B,
    ddt(B,1) = Delta*(B + 0.1*sin(B)),
    eval(A,0)=A0,   eval(A,T)=A1,

```

```

    eval(B,0)=1,
    A in [-1.0E100,1.0E100],
    B in [-1.0E100,1.0E100],
    T=T1-T0,    T in [0,I]
  ]}.

```

3 General Theory of CLP Constraint Domains

A CLP Constraint domain D is specified by giving the syntax and semantics of its underlying constraint language.

Syntactically, constraints in a domain D are a conjunction of atomic formulas in a first order language L_D . The language is specified by giving the predicate symbols, function symbols, constant symbols, and variable symbols of the language. (In general the language may require a multi-sorted logic to cleanly handle variables of different types.)

The semantics for a CLP Constraint domain is given by a specific model ϕ_D for the language L_D , i.e., a concrete interpretation of the predicate, function, constant, and variable symbols. So ϕ_D is a map from the symbols to a corresponding set of predicates, function, and constants. The theory T_D of the domain is the set of all first order formulas in L_D which are true under the interpretation ϕ_D .

A CLP Constraint solver is an algorithm which tests for unsatisfiability of constraints. It must be correct, but it doesn't have to be complete, i.e. for any constraint C the solver either determines that the constraint is unsatisfiable, or it makes no claim about the constraint's satisfiability. Thus, if the solver determines that a constraint $C(X)$ is unsatisfiable, then there is a proof that

$$T_D \models \neg \exists X C(X)$$

but the solver might not be able to detect all unsatisfiable constraints.

An interval constraint solver approximates the solution set for a constraint by assigning an interval to each variable in the constraint. For the purpose of this section, we can think of an interval for a general domain to a subset of the domain (possibly with some additional restrictions).

An interval constraint solver proves unsatisfiability by applying contraction algorithms which attempt to shrink the intervals without removing any solutions. If an interval for one of the variables is shrunk to the empty set, then the constraint is unsatisfiable. In general if an interval constraint solver is given a constraint $C(X)$ on a tuple X of variables and it contracts the variables from an initial tuple I to a subset J , then one can infer that

$$T_D \models \forall X. (C(X) \wedge X \in I) \Rightarrow X \in J$$

That is, the contraction algorithms can be viewed as mechanical theorem provers for a simple class of formulas and hence the CLP constraint solver is itself a theorem prover.

A CLP(D) program P can be interpreted as a first order theory T_P using Clark's completion semantics [Cla78]. In this semantics, each predicate in the program is replaced by a rule

$$\forall X p(X) \Leftrightarrow (\exists Y_1 q_1(X, Y_1)) \wedge \dots$$

where each clause of the program is viewed as a conjunction q_1 of atomic formulas over the variables X from the head and some new variables Y introduced in the clause. If a CLP(D) interpreter generates m interval solutions I_1, \dots, I_m to a query $Q(X)$ then one can infer that the interpreter has produced a proof that

$$T_D \cup T_P \models \forall X. Q(X) \Rightarrow \bigvee_j X \in I_j$$

For example, let P is the following CLP(D) program over the domain D of reals:

$p(X, Y, N) :- \{N=0, Y=\cos(Z), \exp(Z+Y)=X\}.$
 $p(X, Y, N) :- \{N>0, M=N-1, Y= \exp(Y)+\cos(Y1)\}, p(X, Y1, M).$

then its Clark semantics is

$$\begin{aligned} \forall X, Y, N. p(X, Y, N) \Leftrightarrow & \exists Z (N = 0 \wedge Y = \cos(Z) \wedge e^{Z+Y} = X) \\ & \vee (N > 0 \wedge M = N - 1 \wedge Y = e^Y + \cos(Y_1) \wedge p(X, Y_1, M)) \end{aligned}$$

In the next two sections we discuss the syntax and semantics of the CLP(I) and CLP(F) languages.

4 How Clip Works

CLIP maintains a dequeue (double ended queue) of active constraints as well as a stack of all constraints, a stack of interval-valued variables and some other control structures to handle backtracking (a stack of choice points, a trail of bindings).

After a new constraint is added to the constraint stack and enqueued in the dequeue, the solver iteratively processes constraints in the dequeue and applies a contractor for that constraint to attempt to narrow (shrink, contract, ...) some of the intervals in the variable stack without removing any solutions to the constraint in question.

If the narrowing results in a variable X being contracted by more than some fixed amount specified by the tuning parameter `sensitivity`, then all constraints involving X are put on the dequeue. If the change to a X was more than the tuning parameter `stack_sensitivity`, the constraints are added to the front of the dequeue (i.e. pushed), otherwise they are added to the end (i.e. enqueued). CLIP does not add a constraint to the dequeue if it is already there.

CLIP continues to narrow intervals until the dequeue is empty. A naive implementation of this solver would be subject to freezing up when the solver enters a cycle of making very small changes to a set of variables with negligible

progress (as measured in the decrease in the total size of the interval). To eliminate this problem (and guarantee a maximum return time for each call to the constraint solver), CLIP introduces the `max_narrow` tuning parameter. Once `max_narrow` operations occurred, CLIP continues narrowing, but only puts constraints on the dequeue if the change in the one of the variables in the constraint is more than the `insensitivity` tuning parameter. Note that `sensitivity`, `stack_sensitivity` and `insensitivity` are relative, not absolute changes, and all of these parameters can be modified inside `clip` using `set_clip`. Also, the precise semantics of these parameters is tricky since they must apply to bounded intervals as well as those where one endpoint is infinite.

5 The CLP(I) Constraint Domain

The CLP(I) constraint language defines a constraint as a conjunction of atomic formulas in the language specified below. Constraints in CLP(I) are enclosed with curly braces to indicate that they are to be processed by the constraint solver and not the usual Prolog engine. Thus, all CLP(I) constraints have the form:

$$\{C_1, C_2, \dots, C_n\}$$

where the C_i are atomic constraints.

The CLP(I) constants include the standard representations of decimal numbers and the semantics of a CLP(I) constant is somewhat complex. If the decimal number constant c is in fact exactly representable by a floating point number f , then the the CLP(I) semantics assigns the symbol c to the value f . For decimal number symbols c without an exact floating point representation, there are two floating point numbers f_0 and f_1 such that $f_0 < c < f_1$ and the semantics of CLP(I) maps c to some unspecified number strictly between f_0 and f_1 . Note that in this case, syntactic constants are actually represented by slightly constrained variables.

CLP(I) variables are represented by capitalized identifiers (as in Prolog) and correspond to real numbers under the CLP(I) model.

The integers are naturally embedded in the reals and we identify the booleans with the subset $\{0, 1\}$ with 0 being false and 1 being true. Thus, $integer(X)$ is true iff X is an integer and $boolean(X)$ is true if and only if X is 0 or 1. We could actually define these in terms of the other constraints:

$$integer(X) \equiv \sin 2\pi(X/2) = 0 \quad boolean(X) \equiv integer(X) \wedge X \in [0, 1]$$

CLP(I) has a very rich language of functions which include those shown in Table 2.

The atomic constraints CLP(I) constraints are constructed from the predicates in Table 1. These represent the usual predicates on reals, with one exception – the “in” predicate, which we discuss in more detail below.

The usual use of the “in” predicate is $X \text{ in } [A, B]$ which has the meaning $A \leq X$ and $X \leq B$ One can get a reasonable semantics for $X \text{ in } Y$ where Y

is a real variable or expression by using the domain of real-valued functions on an unspecified set S instead of the domain of real numbers. The real numbers a are embedded in this domain as constant functions f_a with $\forall t. f_a(t) = a$ and the F in G relation is interpreted as range inclusion $F(S) \subseteq G(S)$. The real operators are extended to functions using a pointwise semantics: $(F \circ G)(t) = F(t) \circ G(t)$. Since the reals are embedded in this function space, one can introduce a type system with both real and function variables. This semantics plays a central role in the CLP(F) constraint domain described in the next section.

Symbol	Arity	Pos	Description
$S = T$	2	In	equals
$S < T$	2	In	less than
$S \leq T$	2	In	less than or equal
$S \geq T$	2	In	greater than or equal
$S \neq T$	2	In	not equals
$\text{integer}(T)$	1		Integer
$\text{boolean}(T)$	1		Boolean
$S \text{ in } T$	2	In	Containment

Table 1: CLP(I) constraint predicates where S and T are CLP(I) constraint terms

Pos is “In” for infix operators.

Note that the $<, \leq, =$ are functions as well as predicates. As function they return 0 or 1 and so you can write

$$(X < 1) + (Y < 1) + (Z < 1) = 2$$

to represent the constraint that exactly two of the three variables X, Y, Z are smaller than 1.

6 The CLP(F) Constraint Domain

Constraints in CLP(F) all have the form

$$\{[C_1, C_2, \dots, C_n]\}$$

where the C_i are atomic CLP(F) constraints. The variables that appear in CLP(F) constraints are of two types. Either they are real variables (as in CLP(I) constraints) which represent real numbers or they are function variables which represent infinitely differentiable functions defined on a finite interval $[a, b]$.

The function variables must be declared outside of the CLP(F) constraint. They are declared using a CLP(F) declaration as follows:

```
decls([F1, ..., Fn], function(A, B))
```

Symbol	Arity	Pos	Description
$S + T$	2	In	addition
$S * T$	2	In	multiplication
$S - T$	1		unary minus
$\exp(T)$	1		e^T
$\text{sq}(T)$	1		square
$\text{abs}(T)$	1		absolute value
$\text{sgn}(T)$	1		sign of argument, -1, 0, or 1
$\text{max}(S, T)$	2		Maximum
$\text{min}(S, T)$	2		Minimum
$\text{floor}(T)$	1		Floor
$\text{ceil}(T)$	1		Ceiling
$S \text{ or } T$	2	In	Logical OR (and S,T are boolean, i.e. in {0,1})
$S \text{ and } T$	2	In	Logical AND (and S,T are boolean)
$S \text{ xor } T$	2	In	Logical eXclusive OR (and S,T are boolean)
$S \text{ not } T$	1		Logical Negation (and T is in {0,1})
$\sin(T)$	1		Sine
$\cos(T)$	1		Cosine
$\tan(T)$	1		Tangent
$S < T$	2		Less than function mapping to {0,1}
$S \leq T$	2		Less than or equals function
$S = T$	2		Equals function
$\text{sin2pi}(T)$	1		returns $\sin(2 \cdot \pi \cdot X)$
$\text{cos2pi}(T)$	1		returns $\cos(2 \cdot \pi \cdot X)$
$\text{tan2pi}(T)$	1		returns $\tan(2 \cdot \pi \cdot X)$
$\text{evenpow}(S, T)$	2		if $S > 0$ then S^T ; if $S = 0$ then 0; if $S < 0$ then $(-S)^T$
$\text{oddpow}(S, T)$	2		if $S > 0$ then S^T ; if $S = 0$ then 0; if $S < 0$ then $-(-S)^T$
$\text{psqrt}(T)$	2		the positive squareroot of T

Table 2: Clip functions

This states the the F_i are infinitely differentiable functions in an open set containing the interval $[A, B]$.

Atomic CLP(F) constraints include all CLP(I) constraints on real variables and in addition include the following, where F, G are expressions of type function and S, T are expressions of type real. The list of functions available in CLP(F)

Symbol	Arity	Pos	Description
$F = G$	2	In	equality of functions
$F = T$	2	In	function F is the constant function T
$T = F$	2	In	function F is the constant function T
$S = T$	2	In	real numbers S and T are equal
<code>identity(F)</code>	1		F is the identity function
$F \leq G$	2	In	$F(t) \leq G(t)$ for all $t \in [A, B]$
$F \text{ in } T$	2	In	$F([A, B]) \subseteq T$
$F \text{ in } G$	2	In	$F([A, B]) \subseteq G([A, B])$

Table 3: CLP(I) constraint predicates where F, G are functions and S, T are reals

includes all CLP(I) functions on reals and add the operators in Table 4. Note that when functions and reals are combined (as in $F + X$) the result is a function obtained by interpreting the real value as a constant function.

Symbol	Arity	Pos	Description
$F + G, F + T, T + F$	2	In	addition
$F * G, F * T, T * F$	2	In	multiplication
$F - G, F - T, T - F$	1		subtraction
$F/G, F/T, T/F$	1		division
<code>exp(F)</code>	1		$e^{F(t)}$
<code>log(F)</code>	1		$\log(F(t))$
<code>sin(F)</code>	1		$\sin(F(t))$
<code>cos(F)</code>	1		$\cos(F(t))$
<code>tan(F)</code>	1		$\tan(F(t))$
<code>ddt(F, n)</code>	2		n th derivative of F , n an integer constant
<code>eval(F, T)</code>	2		$F(T)$

Table 4: CLP(F) functions

There are also a few tuning parameters and printing procedures CLP(F) adds:

`set_degree(N)` sets to N (default 10) the degree of the Taylor approximation polynomial used for functions.

`print_ps_clip/1` prints the power series of its argument (defined in `ode.pl`)

7 CLIP Predicates and Commands

CLIP is an extension of GNU Prolog in which several new predicates have been added to the base system. The most important of these new predicates are the constraint predicates for `CLP(I) - { }` and `{[]}` and `decls`. The other predicates are mostly commands that allow one to interact with the constraint engine – getting/setting tuning parameters and other information stored in the constraint engine. They don't affect the logical semantics of the program, but they may affect the performance.

The basic predicates and commands in CLIP are described in section 10.

7.1 Command Descriptions

`'$INT'(I)` refers to the interval variable whose index in the constraint engine is `I`.

- `{ }/1` takes the included list of constraints and does as much narrowing as it can.
- `{[]}/1` takes the included list of function constraints and adds them to the current constraints (defined in `ode.pl`)
- `help_clip/0` prints a list of available commands.
- `set_cp_clip/0` sets a choice point in the constraint engine.
- `print_clip/1` prints its argument as an interval rather than as an index into the constraint engines variable stack.
- `reset_clip/0` sets clip to its original state. In particular, it clears the dequeue. Dequeue usage can be a problem on certain calculations. Note that `reset_clip` does *not* change the values which have been set with `set_clip`.
- `get_bounds_clip/3` (`Var`, `Lo`, `Hi`) returns the bounds of interval `Var` in `Lo` and `Hi`. This only works on variables which are bound to an interval. For variables which are not bound, use `'$INT'(Num)` where `Num` is the index in the constraint engine of the interval variable.
- `get_hex_bounds_clip/3` gets the bounds as two hex numbers
- `get_bit_bounds_clip/3` gets the bounds as two bit strings (represented as 4 unsigned shorts).

- `dump_clip/1` provides information on CLIP's state. `dump_clip(stats)` gives statistics: number of choice points, number of constraints, value of `maxcon`, number of variables, value of `maxvar`, value of `conbot`, value of `trail`, `var_dep`, `narrows`, `changes`, and `big changes`.

`dump_clip(all)` lists all the state information it has, which includes `stats`, and also gives information on each variable, including its type, range, and the last change to it.

It lists the active constraints and variables (Note that constants are variables whose numbers start at 99999 and go down, variables start at 0 and go up) and choice points.

It then provides statistics on CLIP's state: the number of choice points (essentially the number of vertices in the tree of execution), the number of constraints active (`con`), the maximum number of constraints (`maxcon`) at any one time, the bottom of the constant storage area (`conbot`), the number of narrowing operations performed so far (`narrows`), the number of primitive narrowing operations (`prim`), the number of times an interval changed (`changes`), and the number of times an interval changed by more than 10% (`big changes`).

!!! what other arguments does it take??

The values described by `dump_clip` can all be reset by `reset_clip`.

- `set_clip/2 (Var, Val)` is used to set the control variable `Var` to `Val`. Note that variables that can take non-integer numbers must have at least one digit before and at least one digit after the decimal point. Values which can be set are
 - `accuracy` (default: 0.0001) is the minimum interval size. A variable whose size is less than `accuracy` is not requeued for more narrowing.
 - `sensitivity` (default: 0.05) The minimum change to an interval which will cause CLIP to put that interval back on the stack of intervals to narrow further (until `max_narrow` narrowings have occurred). `sensitivity` can safely be set to 0.0 so that any change in an interval will cause the interval to be requeued. Note that `sensitivity` is a fraction, not an absolute number.
 - `max_narrow` default: (10 000) - the maximum number of narrowings using `sensitivity` to determine whether to continue narrowing (used to prevent infinite loops). After `max_narrow` steps, CLIP will only add an interval to the stack of work remaining if the change was greater than `insensitivity`. `insensitivity` should be significantly greater than `sensitivity`. Note that `sensitivity` is a fraction, not an absolute number.
 - `insensitivity` (default: 0.125) The minimum change necessary to requeue an interval after `max_narrow` narrowings have been done. Note that `insensitivity` is a fraction, not an absolute number.

- `narrow_debug` boolean. (default: false) causes CLIP to show the value of the interval being narrowed before and after each narrowing operation. This generates an enormous amount of data, so it's only useful for debugging relatively small programs.
 - `finite_bounds` boolean. (default: false) determines whether new variables should have finite or $[-\text{maxreal}, +\text{maxreal}]$ bounds.
 - `stack_sensitivity` (default: 0.25) If an interval is narrowed by more than `stack_sensitivity`, then it is put on the front of the dequeue to be re-narrowed immediately. If the narrowing was by less than `stack_sensitivity`, but more than `sensitivity`, it is put on the end of the dequeue. If the change is less than `sensitivity`, it is not requeued. Note that `stack_sensitivity` is a fraction, not an absolute number.
- `get_clip/2` is used to read the control values which `set_clip` can set as well as some state description. A useful idiom is `get_clip(A,B), write(A=B),nl, fail`. This gets all the parameters, writes the name and value of one, writes a new line, then gets the next value. Some values which can be checked with `get_clip`, but not set with `set_clip` are:
 - `var_top` initial value: 1
 - `constraint_top` initial value: 0
 - `constant_top` initial value: 99 999
 - `max_vars` default 100 000
 - `plot2_solve(M, [X,Y],E,F)` (defined in `plot.pl`)
 - `fplot([X,Y],E,F)` generates GNUPLOT dataset from 2D constraints
 - `fplot([X,Y,Z],E,F)`
 - `plot3_solve(M, [X,Y,Z],E,F)` generates 3dvplot datasets from 2D constraints.
 - `plot_param([X,Y,Z],Params,E,F)`
 - `narrow_all_clip(N)` Adds all constraints to the dequeue, and then performs N narrowing operations.
 - `absolve(L,S,F)` Tries to narrow an interval by checking to see if there are any solutions near the bottom of the interval. If not, it removes the bottom section of the interval. This is repeated until removing a piece of that size from the bottom of the interval would remove an area in which there are solutions. It then does the same for the top of the interval. L is a list of intervals to test, (S, F) give the starting and ending sizes of the piece to check. If F is less than S , it does nothing. Absolve starts by checking for solutions in a piece of size $1/S$ times the size of the original

interval and reduces the size of the piece by half until it reaches one of size $1/F$. By convention S and F are powers of 2.

`absolve` prints characters to show its progress. “-” means that it successfully removed a piece from the low end of the interval “,” that it failed to remove a piece from the low end of the interval “+” that it removed a piece from the high end of the interval “;” that it failed to remove a piece from the high end of the interval Whenever it finishes absolving 1 interval, it prints “*”

- `contract_vars(Vs, G)` solves the given constraints (Vs) for the goal G, and then cleans the stack of all the temporary constraints used. All the variables in Vs must be intervals. Returns the union of the solutions. Defined in `contract.pl`
- `solve_clip(Solver, Term, Eps)` Legal solvers are:
 - `queue` Splits each variable which is too large, then continues going through the list splitting each variable once per time until all the variables are small enough.
 - `fwchk` (Forward Checking Solver) A divide and conquer solver which divides the domain into a set of K boxes (initially $K = 1$) and in each step it divides each box into 2^V sub-boxes, where V is the number of variables in `VARs`, and applies the default narrowing procedure. Any boxes that are proved to contain no solutions are discarded and the result is returned as the smallest box containing all of the remaining boxes.
 - `seq` this traverse a search space by putting the variables into a queue and sequentially splitting the top variable in the queue and the moving in to the back of the queue
 - `incr` – this does a breadth first search (like `fwchk`) of the solution space, but after each breadth-first splitting it reports on each connected component of the solution space before going deeper.
 - `bf`

`Term` is the set of variables to work on, and `Eps` is a work parameter, whose meaning varies according to which solver is used.

- `solve_clip(Term, Eps)` Calls `solve_clip/3` using `queue` as the solver.
- `decls/2`
- `print_ps/1` prints the power series of its argument by showing the endpoints a, b of the interval $I = [a, b]$ is defined on as well as the values of its first n derivatives at a at b and on the interval I .

8 Implementation

In this section we give a brief overview of how the CLP(I) and CLP(F) constraint solvers are implemented.

8.1 CLP(I) constraint solving

The CLP(I) constraint solver is based on a relatively simple model. At any point in time the set of constraints that have been encountered in the current branch of the Prolog search tree are stored in a stack of primitive constraints called the constraint store. Each new constraint that is added to the system is decomposed into a conjunction of primitive constraints and these constraints are added to the constraint store and the variables that appear in the constraint are pushed onto a stack of interval variables. These new primitive constraints are also put into a queue of active constraints.

The constraint solver then attempts to contract the intervals in the interval variable stack by processing the constraints in the constraint queue. If one of the constraints in the queue becomes unsolvable (i.e. an interval contracts to the empty set), then backtracking is triggered as the solver has proved that the current set of constraints is unsatisfiable.

On backtracking the constraints pushed on the stack since the last choice point are popped off, as are the new constraint variables, and any contractions of interval variables are also replaced with their original values using a binding trail stack.

The decomposition of constraints into primitive constraints is a simple process of introducing temporary variables for each subexpression, e.g.

$$X^2 + Y^2 = 25$$

would be mapped to the following conjunction of primitive constraints

$$X^2 = T_1, Y^2 = T_2, T_1 + T_2 = T_3, T_3 = 25$$

where the constant 25 has been compiled into a constraint variable with a constant value.

The constraint solving algorithm simply processes the constraint in the queue by

- taking off the first primitive constraint $C(X_1, X_2, X_3)$ in the queue,
- applying a contraction algorithm for that constraint on its arguments X_1, X_2, X_3 ,
- if any of the variables X_i are contracted by a sufficiently large amount, then all constraints that depend on them (except for C itself), are added to the queue
- the constraint solving continues until either a maximum number of contractions has been performed (`max_narrow` — a tuning parameter), or a constraint has been found to be unsatisfiable.

8.2 Primitive Contraction

The contraction algorithms for the primitive constraints have been implemented very carefully making full use of the directed rounding capabilities of the underlying processors so as to contract their intervals without removing any solutions. Some of these contraction algorithms have published correctness proofs, others have been proved correct only by the authors, but have not been peer reviewed. Also, the programs implementing the algorithms have not been formally proved correct.

As a simple example of contraction, consider the contraction operator for the addition constraint $X + Y = Z$, then a correct contraction algorithm for this constraint is

$$\begin{aligned} X &\leftarrow X \cap (Y + Z) \\ Y &\leftarrow Y \cap (Z - X) \\ Z &\leftarrow Z \cap (Z - Y) \end{aligned}$$

where

$$\begin{aligned} Y+Z &:= [\text{addL0}(y_{\text{lo}},z_{\text{lo}}),\text{addHI}(y_{\text{hi}},z_{\text{hi}})] \\ Z-X &:= [\text{subL0}(z_{\text{lo}},x_{\text{hi}}),\text{subHI}(z_{\text{hi}},x_{\text{lo}})] \\ Z-Y &:= [\text{subL0}(z_{\text{lo}},y_{\text{hi}}),\text{subHI}(z_{\text{hi}},y_{\text{lo}})] \end{aligned}$$

where `addL0`, `addHI`, `subL0`, `subHI` are operations that add or subtract two floating point numbers and the round to the nearest floating point below (for LO) or above (for HI) the actual sum or difference.

A similar approach is used for multiplication, division, and the exponential and trigonometric constraints (although these require that the math libraries for computing `exp`, `sin`, etc be rewritten to return intervals that are certain to contain the actual value).

The `integer(X)` constraint is even easier to implement:

$$X := [\text{ceiling}(x_{\text{lo}}),\text{floor}(x_{\text{hi}})]$$

Note that for soundness all we need to verify is that the contractors do not remove any solutions from the constraint. They do not have to remove all non-solutions!

The library of contractors is available as from the opensource project `interval.sourceforge.net` as the C library `smathlib` and it has been tested under Linux on x86 and Mac OS X on Power PC.

8.3 CLP(F) constraint solving

The CLP(F) constraint solver is implemented in CLP(I). The key idea is to soundly approximate infinitely differentiable functions defined on an interval $I = [A, B]$ by providing intervals for the values of the first n derivatives at the

end points, as well as intervals containing the range of their derivatives over the entire interval I .

More precisely, we approximate a functions by a set of $3n+5$ intervals where n is a tuning parameter (default value of 10). The idea is to use an abstraction operator $\gamma_n(f)$ that maps a function f on $I = [a, b]$ to the following tuple

$$\begin{aligned} &a, b, \\ &f(a), f'(a), f''(a), \dots, f^{(n)}(a) \\ &f(I), f'(I), f''(I), \dots, f^{(n)}(I) \\ &f(b), f'(b), f''(b), \dots, f^{(n)}(b) \end{aligned}$$

in $\mathbf{R}^2 \times \mathbf{R}^{2n+2} \times \mathbf{S}^{n+1}$ where S is the set of real intervals and $f^{(n)}(I)$ denotes the range of the n th derivative of f on the interval I . We can then approximate a set U of functions by a tuple Z of $3n+5$ intervals such that $\gamma_n(f) \in Z$ for each $f \in U$. We can then compute approximations of functional expressions by computing operations on their γ_n approximations. If F is a tuple of intervals such that $\gamma_n(f) \in F$ we say, by abuse of language, that $f \in F$. Thus, if $f \in F$ and $g \in G$, then $f + g \in F + G$ where $F + G$ is obtained by adding the corresponding intervals in F and G .

Thus, the decls predicate

`decls([F],function(A,B))`

is implemented by binding F to a tuple of interval variables:

$$\begin{aligned} F = & \\ &[[A, (F00, F01, F02, \dots, F0n)], \\ &[I, (R0, R1, R2, \dots, Rn)], \\ &[B, (F10, F11, F12, \dots, F1n)]] \end{aligned}$$

where I is the interval $[A, B]$. Since it is assumed that F is infinitely differentiable, we also add Taylor constraints which relate the values of F and its derivatives at A to the Taylor coefficients at B . The remainder terms are computed from the intervals bounding the ranges of the derivatives on $[A, B]$. For example, we add the following constraints:

$$\begin{aligned} T &= B - A, \\ F10 &= F00 + T * Z1, Z1inR1, \\ Z1 &= F01 + T/2 * Z2, Z2inR2, \\ Z2 &= F02 + T/3 * Z3, Z3inR3, \\ &\dots \\ I &= [A, B], \\ R0inF00 + I * R1, \end{aligned}$$

$$R1inF01 + I/2 * R2,$$

$$R2inF02 + I/3 * R3,$$

...

and we also add constraints expressing the $F0j$ in terms of the $F1j$, and vice versa. These constraints allow information about the function at one end point to be propagated to the remainder terms and to the other endpoint. The $R0$ term can be bounded using the syntax:

$$F \text{ in } [L, H]$$

which adds the constraints $L \leq R$ and $R \leq H$. The value of the function at any point X in $[A, B]$ can be expressed using the eval function:

$$\text{eval}(F, X) = Y$$

which adds the Taylor constraints for expressing Y in terms of $(X-A)$, the derivatives at X , and the remainder terms (Rj), as well as $(X-B)$, the derivatives at Y , and the remainder terms (Rj).

The constraint $ddt(F, 1) = F$ is converted into two primitive constraints $ddt(F, 1) = G$ and $G = F$. The primitive constraint $ddt(F, 1) = G$ is converted into the obvious set of corresponding constraints

$$F10 = G00,$$

$$F20 = G10, \dots$$

$$R1 = S0,$$

$$R2 = S1, \dots$$

$$F11 = G01,$$

$$F21 = G11, \dots$$

Other function operators are treated similarly, for example $F * G = H$ generates constraints on the Fij , Gij and Hij as well as their remainder terms expressing the corresponding relations among the derivative functions:

$$H00 = F00 * G00$$

$$H01 = F00 * G01 + F01 * G00$$

$$H02 = F00 * G02 + F01 * G01 + F02 * G00$$

... ..

The exponential and trigonometric functions are handled by reducing them to other functional equations, e.g.

$$F = \exp(G) \Rightarrow F' = G' * F \wedge F(A) = e^A \wedge F(B) = e^B \wedge F([A, B]) = e^{[A, B]}$$

and so the constraint $F = \exp(G)$ is converted to

```

ddt(F,1)=G*F
eval(F,A)=exp(A)
eval(F,B)=exp(B)
F in exp([A,B])

```

The equality constraint among functions is soundly approximated by setting the corresponding intervals in their γ_n approximation equal and the inequality constraint $F < G$ is converted to inequalities

```

eval(F,A) < eval(G,A)
eval(F,B) < eval(G,B)
eval(F,Z) < eval(G,Z), Z in [A,B]

```

Note that the constraint solver only needs to be sound not complete (i.e. contractions need only keep all solutions, they don't have to remove all non-solutions). Thus, we are free to transform any constraint $C(X)$ into a weaker constraint $D(X)$ as long as $C(X) \Rightarrow D(X)$, because if $D(a)$ is proved to be false (and hence a is eliminated), then we will know that $C(a)$ is likewise false and can be eliminated.

9 Known Bugs and Issues

9.1 Unification and Constraint Solving

Probably the most aggravating of the problems with the CLIP implementation is that it currently does not support a tight integration with GNU Prolog's unification. Thus, two interval variables can only be unified by an explicit constraint $\{X=Y\}$. In particular, the following query will succeed:

```

?- {X=X, Y=Y, X=Y}.
X=Y
?-

```

But if we move the last equality out of the constraints, it fails:

```

?- {X=X, Y=Y}, X=Y.
no
?-

```

More subtle is that head unification is converted into an implicit equation which will fail. Thus if p is a procedure defined by:

$$p(X, X).$$

then the following query will also fail.

```

?- {X=X, Y=Y}, p(X,Y).
no
?-

```

This bug can be fixed by using some recent extensions to GNU Prolog that allow one to add unification hooks, but for now it requires a careful (manual) separation of interval variables from tree variables which guarantees no interval variables are implicitly unified in the head of any rule.

9.2 Other bugs and issues

- `help_clip` fails to write out the constraint operators. This may be a result of porting from Sicstus Prolog and using \wedge , but GNU Prolog does define \wedge in what seems to be the correct way.
- `absolve` fails when the interval it's working on is very small. It appears that when the interval it's checking is very small (perhaps 1 ULP??) it goes into an infinite loop.
- It would be nice if `>` worked on functions. For now, we use `F in [0, _]` for $F \geq 0$.
- Related to the previous item, one cannot have a function `f` equal a constant `c`. A workaround is to say `f = c + 0 * f`
- `psqrt` is non-analytic at 0. Currently we simply define `psqrt` only for values greater than some small ϵ , but that's not rigorous.
- When reading non-integer numbers, even if the number (say 0.5) has an fp representation, the interval read in is at least 2 ULPs wide, 1 on each side of the correct value.
- Not really a bug, but when you increase Prolog's stack size, `Clip` does not take advantage of the extra memory. Perhaps we should re-compile `clip` with larger arrays to take advantage of the larger memories that have become standard. As of Jan. 2004, `clip`'s total static memory usage is about 20 MB. - we could easily up all the static declarations by a factor of 5 or 10.
- If you don't specify a range for a function, all sorts of intervals grow much larger than they should. Perhaps a warning would help??
- Prolog warns about singleton variables, but if you get too enthusiastic about putting underscores in front of variables, there is no warning that you have two variables with the same name `_X` and they are treated as separate variables.
- `Clip` is much more sensitive than it should be to the order in which constraints are listed. This may be a bug, or it may be a place for improvement.
- When reading integers which are too large, `clip` sometimes reads positive integers as negative. Presumably it is writing an unsigned int, which is then read as a signed int.

| ?- {T=1000000000}.

T = -73741824 ?

10 Quick Reference

10.1 CLIP commands

```
{ }/1  
help_clip/0  
set_cp_clip/0  
reset_clip/0  
get_bounds_clip/3  
dump_clip/1  
set_clip/2  
get_clip/2  
narrow_all_clip/1
```

```
ode.pl adds:  
{ [ ] }/1  
set_degree/1  
decls/2  
print_ps_clip/1
```

```
solvers.pl adds:  
print_clip/1  
solve_clip/3  
solve_clip/2
```

```
plotting.pl adds:  
plot2_solve/4  
fplot/3
```

```
contract.pl adds:  
contract_vars/2
```

10.2 CLP(I) predicates and functions

Symbol	Arity	Pos	Description
$S = T$	2	In	equals
$S < T$	2	In	less than
$S \leq T$	2	In	less than or equal
$S \geq T$	2	In	greater than or equal
$S \neq T$	2	In	not equals
$\text{integer}(T)$	1		Integer
$\text{boolean}(T)$	1		Boolean
$S \text{ in } T$	2	In	Containment

Symbol	Arity	Pos	Description
$S + T$	2	In	addition
$S * T$	2	In	multiplication
$S - T$	1		unary minus
$\exp(T)$	1		e^T
$\text{sq}(T)$	1		square
$\text{abs}(T)$	1		absolute value
$\text{sgn}(T)$	1		sign of argument, -1, 0, or 1
$\text{max}(S, T)$	2		Maximum
$\text{min}(S, T)$	2		Minimum
$\text{floor}(T)$	1		Floor
$\text{ceil}(T)$	1		Ceiling
$S \text{ or } T$	2	In	Logical OR (and S,T are boolean, i.e. in {0,1})
$S \text{ and } T$	2	In	Logical AND (and S,T are boolean)
$S \text{ xor } T$	2	In	Logical eXclusive OR (and S,T are boolean)
$S \text{ not } T$	1		Logical Negation (and T is in {0,1})
$\sin(T)$	1		Sine
$\cos(T)$	1		Cosine
$\tan(T)$	1		Tangent
$S < T$	2		Less than function mapping to {0,1}
$S \leq T$	2		Less than or equals function
$S = T$	2		Equals function
$\text{sin2pi}(T)$	1		returns $\sin(2 \cdot \pi \cdot X)$
$\text{cos2pi}(T)$	1		returns $\cos(2 \cdot \pi \cdot X)$
$\text{tan2pi}(T)$	1		returns $\tan(2 \cdot \pi \cdot X)$
$\text{evenpow}(S, T)$	2		if $S > 0$ then S^T ; if $S = 0$ then 0; if $S < 0$ then $(-S)^T$
$\text{oddpow}(S, T)$	2		if $S > 0$ then S^T ; if $S = 0$ then 0; if $S < 0$ then $-(-S)^T$
$\text{psqrt}(T)$	2		the positive squareroot of T

10.3 CLP(F) predicates and functions

Symbol	Arity	Pos	Description
$F = G$	2	In	equality of functions
$F = T$	2	In	function F is the constant function T
$T = F$	2	In	function F is the constant function T
$S = T$	2	In	real numbers S and T are equal
<code>identity(F)</code>	1		F is the identity function
$F \leq G$	2	In	$F(t) \leq G(t)$ for all $t \in [A, B]$
$F \text{ in } T$	2	In	$F([A, B]) \subseteq T$
$F \text{ in } G$	2	In	$F([A, B]) \subseteq G([A, B])$

Symbol	Arity	Pos	Description
$F + G, F + T, T + F$	2	In	addition
$F * G, F * T, T * F$	2	In	multiplication
$F - G, F - T, T - F$	1		subtraction
$F/G, F/T, T/F$	1		division
<code>exp(F)</code>	1		$e^{F(t)}$
<code>log(F)</code>	1		$\log(F(t))$
<code>sin(F)</code>	1		$\sin(F(t))$
<code>cos(F)</code>	1		$\cos(F(t))$
<code>tan(F)</code>	1		$\tan(F(t))$
<code>ddt(F, n)</code>	2		n th derivative of F , n an integer constant
<code>ddt(n, F)</code>	2		n th derivative of F , n an integer constant
<code>eval(F, T)</code>	2		$F(T)$

References

- [BO97] Frédéric Benhamou and William J. Older. Applying interval arithmetic to real, integer, and boolean constraints. *Journal of Logic Programming*, 32(1):1–24, Jul 1997.
- [Cla78] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, New York, NY, 1978.
- [Cle87] J.G. Cleary. Logical arithmetic. *Future Computing Systems*, 2:125–149, 1987.
- [DEDC96] Pierre Deransart, AbdelAli Ed-Dbali, and Laurent Cervoni. *Prolog: The Standard; Reference Manual*. Springer Verlag, 1996.
- [Dia02] Daniel Diaz. *GNU Prolog Manual*, 1.7 edition, Sep 2002.
- [Hic00] Timothy J. Hickey. Analytic constraint solving and interval arithmetic. In *POPL'00 ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 338–351, 2000. published as vol. 27 of SIGPLAN notices.
- [Hic01] Timothy J. Hickey. Metalevel interval arithmetic and verifiable constraint solving. *Journal of Functional and Logic Programming*, 2001(7), October 2001. <http://danae.uni-muenster.de/lehre/kuchen/JFLP/articles/2001/S01-02/JFLP-A01-07.pdf>.
- [HJ] Timothy J. Hickey and Qun Ju. clip 1.0 a CLP(Intervals) interpreter, based on Sicstus Prolog. interval.sourceforge.net/interval/prolog/clip.
- [HJ97] Timothy J. Hickey and Qun Ju. Efficient implementation of interval arithmetic narrowing using IEEE arithmetic. Technical report, Brandeis University CS Dept, April 1997. www.cs.brandeis.edu/~tim/narrow_multiply/paper.ps.
- [HW99a] Timothy J. Hickey and David K. Wittenberg. Validated constraint compilation. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming - CP'99*, volume 1713 of *Lecture Notes in Computer Science*, pages 482–483, 1999. A longer version is [HW99b].
- [HW99b] Timothy J. Hickey and David K. Wittenberg. Validated constraint compilation. Technical Report CS-99-201, Computer Science Department, Brandeis University, April 1999. URL: www.cs.brandeis.edu/~tim/Papers/cs99201.ps.gz.

- [HW03] Timothy J. Hickey and David K. Wittenberg. Using analytic CLP to model and analyze hybrid systems. Technical Report CS-03-240, Brandeis University, 2003. <http://www.cs.brandeis.edu/~dkw/papers/cs03-240.pdf>, Accepted to FLAIRS 04.
- [IEE85] IEEE. IEEE standard 754-1985 for binary floating-point arithmetic. *SIGPLAN*, 22(2):9–25, 1985.
- [JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings 14th ACM Symposium on the Principles of Programming Languages*, pages 111–119, 1987.
- [JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
- [Ju98] Qun Ju. *A Sound Interval Constraint Logic Programming System*. PhD thesis, Brandeis University, May 1998.
- [Moo66] Ramon E. Moore. *Interval Analysis*. Prentice-Hall, 1966.
- [OV93] W. Older and A. Vellino. Constraint arithmetic on real intervals. In A. Colmerauer and F. Benhamou, editors, *Constraint Logic Programming: Selected Research*. MIT Press, 1993.
- [Pro95] ISO Prolog standard iso/iec 13211-1, information technology — programming languages — prolog — part 1: General core. available from www.iso.org/iso/en/, 1995.
- [Res88] Bell Northern Research. *BNR Prolog user guide and reference manual*. Bell Northern Research, 1988.