The CLP($\mathcal{R}$) Programmer's Manual
Version 1.2

NEVIN C. HEINTZE [†]
JOXAN JAFFAR [‡]
SPIRO MICHAYLOV [*]
PETER J. STUCKEY [§]
ROLAND H.C. YAP [¶§]

[‡] *IBM Thomas J Watson Research Center*
PO Box 704
Yorktown Heights, NY 10598, U.S.A.

[†] *School of Computer Science*
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.

[*] *Department of Computer and Information Science*
The Ohio State University
Columbus, OH 43210-1277, U.S.A.

[§] *Department of Computer Science*
University of Melbourne
Parkville, Victoria 3052, Australia

[¶] *Department of Computer Science*
Monash University
Clayton, Victoria 3168, Australia

September 1992

Contents

Chapter 1

Introduction

This manual describes CLP($\mathcal{R}$) version 1.2, and at a number of places throughout this text changebars have been placed either to indicate new features in version 1.2 from version 1.1 or some changes in the manual. The CLP($\mathcal{R}$) language is an instance of the Constraint Logic Programming scheme defined by Jaffar and Lassez [10]. Its operational model is similar to that of PROLOG. A major difference is that unification is replaced by a more general mechanism: solving constraints in the domain of uninterpreted functors over real arithmetic terms. A working knowledge of PROLOG programming is assumed in this document; the book by Sterling and Shapiro [20] can serve as a suitable introductory text. Further technical information on CLP($\mathcal{R}$) is available on language design and implementation [12, 13], meta-programming [7] and delay mechanisms [14]. Additionally, much has been written about applications in electrical engineering [6, 18], differential equations [5, 8], temporal reasoning [1, 2, 3], protocol testing [4], structural analysis and synthesis [15], mechanical engineering [21], user interfaces [23], model-based diagnosis [24], options trading [16], music theory [9], molecular biology [22], *etc.*

This document is both an introductory tutorial and reference manual describing the compiler-based implementation of CLP($\mathcal{R}$). The reader experienced with PROLOG or CLP($\mathcal{R}$) may wish to skip to Chapter 4, and in particular, see the sample session in Section 4.6 to get started quickly. Compiled CLP($\mathcal{R}$) is an interactive system that compiles all programs and goals into CLAM code which is interpreted by a byte-code emulator that is part of the system. The system is portable in the sense that it will run on virtually all 32 bit UNIX$^{\text{TM}}$ machines with a reasonably standard C compiler, as well as many others.

We would like to emphasize that this manual describes a constantly-evolving, experimental system. Hence much of what is described is subject to change in future releases. Furthermore, the use of undocumented features is particularly dangerous.

Finally, we adopt some standard notational conventions, such as the name/arity convention for describing predicates and functors, `+` for input arguments, `-` for output arguments, and `?` for arguments that may be either input or output.

Chapter 2

Syntax and Simple Examples

A CLP($\mathcal{R}$) program is a collection of rules. The definition of a rule is similar to that of a PROLOG clause, but it differs in two important ways: rules can contain constraints as well as atoms in the body, and the definition of terms is more general. A goal is a rule without a head, as usual.

The body of a rule may contain any number of arithmetic constraints, separated by commas in the usual way. Constraints are equations or inequalities, built up from real constants, variables, `+`, `-`, `*`, `/`, and `=`, `>=`, `<=`, `>`, `<` where all of these symbols have the usual meanings and parentheses may be used in the usual way to resolve ambiguity. Unary arithmetic negation is also available, as are some special interpreted function symbols which will be described later. Any variable that appears in an arithmetic constraint is said to be an arithmetic variable, and cannot take a non-arithmetic value. These constraints may be thought of as built-in predicates written infix, but they are really much more powerful, as we shall see later. Goals are also similar to those in PROLOG, and may contain explicit constraints as well.

Comments in the program are either in the PROLOG style, beginning with a "`%`" and continuing to the end of the line, or also in the form of C style comments, starting with "`/*`" and ending with "`*/`" (comments can contain newlines). Unlike normal C comments, these can be nested so that code already containing comments can be commented easily.

2.1 Terms and Constraints

Syntactically, a term is either a simple term or a compound term constructed from simple terms. A term is then either an arithmetic term or a functor term. The simple terms are:

- Variable terms
  A variable is a sequence of alphanumeric characters (including "_"), either begins with an uppercase alphabetic character or an underscore "_". Variables consisting of an underscore only are anonymous variables and always represent a new variable. Variables that are longer than one character and begin with an underscore are the same as any other ordinary variable,[1] except that they are ignored for the purposes of style checking.

- Numeric constant terms
  This is a real number with an optional decimal point and optional integer exponent which may be positive or negative.

- Symbolic numeric constants
  These denote special constant values, eg. π and have the syntax `#`< *name* > where the name is just an atomic functor constant. The following are the special constants defined by default:

`#p`	$\pi =$	3.14159265358979323846	
`#p_2`	$\pi/2 =$	1.57079632679489661923	
`#p_4`	$\pi/4 =$	0.78539816339744830962	
`#e`	$e =$	2.7182818284590452354	
`#sqrt2`	$\sqrt{2} =$	1.41421356237309504880	
`#sqrt1_2`	$1/\sqrt{2} =$	0.70710678118654752440	
`#c`	$c =$	$2.99792458 * 10^{8}$	(speed of light in vacumn)
`#g`	$g =$	9.80665	(acceleration of gravity)
`#h`	$h =$	$6.626176 * 10^{-34}$	(Planck's constant)
`#ec`		$1.6021892 * 10^{-19}$	(elementary charge)

  There are also some handy metric conversion ratios predefined:

`#cm2in`	0.393701	(centimeters to inches)
`#km2mile`	0.62137	(kilometers to miles)
`#gm2oz`	0.03527	(grams to ounces)
`#kg2lb`	2.20462	(kilograms to pounds)
`#l2gal`	0.21998	(litres to imperial gallons)
`#l2usgal`	0.26418	(litres to US gallons)

  (Note that new constants can be created by using `new_constant/2`.)

- functor constant terms
  These are either a sequence of alphanumeric characters (including "_", starting with a lowercase letter; or a sequence of characters from the set,

$$\left\{ \texttt{\&*+-./:;<=>?@\^{}\~{}} \right\}$$

[1] These are not anonymous variables.

Also any sequence of characters delimited by single quotes "'" is allowed, e.g. `'foo + bar'` is a functor constant (atom) with that name including the blanks. The special constant "`[]`" denotes the empty list or nil. Note also that the special arithmetic function symbols, though having the same syntax, are arithmetic terms and not functor terms.

- String constant terms
  This is any sequence of characters delimited by double quotes ("). NOTE: At present the interpretation of strings in the syntax has not been finalized and all strings are being treated as functor constants (i.e. the single quote form). This differs from some PROLOG's which use this syntax as an alternative notation for lists.

An arithmetic term is either a variable, numeric constant or a compound term built up from arithmetic terms in the usual way using the arithmetic function symbols: `+`, `-`, `*`, `/`, `sin`, `arcsin`, `cos`, `arccos`, `pow`, `abs`, `min` and `max`. For example,

```
X
3.14159
42e-8
X + Y
sin(X + 2.0)
(X + Y) / 4
```

are all valid arithmetic terms. However,

```
f(a)
c + 5.0
cos(f(3))
```

are not. The arithmetic terms are interpreted as having their usual meaning as arithmetic expressions. Operator precedences for the arithmetic function symbols follow the normal convention[2]. Parentheses can be also used to escape the application of the default operator precedences.

Functor terms are either variable or functor constant terms or compound terms. A compound functor term has the form $f(t_1, t_2, \cdots, t_N)$ where $N \geq 0$, f is an N-ary uninterpreted functor and $t_1, t_2, \cdots, t_N$ are (not necessarily functor) terms. The functor is uninterpreted, meaning that the functor is simply to be treated as a symbolic constant, as opposed to the arithmetic terms, which are interpreted. The allowable syntax of the functor symbol f is that of any functor constant term. The other compound functor terms are lists, which are specified using the usual PROLOG list notation (`[L]`), for example "`[a, b]`". A dot

[2]User defined unary or binary operators in the standard PROLOG fashion using `op/3` are also supported.

notation for lists, as in "`a.b.[]`", may also be used. For example, the following are valid terms:

```
[a, 1+X]
f([3.12, g(a)])
f(c)
f(X)
f(3.14159)
g(22, h(4))
f(X + 3)
```

A *constraint* is either an *arithmetic constraint* or a *functor constraint*. The former is defined to be of the the form $t_1 \ \Delta \ t_2$ where t_1 and t_2 are arithmetic terms and Δ is one of the arithmetic relations `=`, `>=`, `<=`, `>`, and `<`. For example,

```
X > 5.0
X + Y + Z = 3
X <= Y
X = V
3 = sin(X)
1.234 + X < Y
```

are all valid arithmetic constraints, while the following are not.

```
c > Y
X = 3.0 < Y
pow(X = Y, 3)
4 < X < 5
```

A functor constraint is of the form $t_1 = t_2$ where each of t_1 and t_2 is either a variable or a functor term. We shall sometimes refer to a functor constraint as a functor equation below.

2.2 Some Simple Programs

Now we will look at some example programs without considering the details of their execution. The first example is a program expressing the relation `fib(N, X)` where `X` is the `N`th Fibonacci number.

```
fib(0, 1).
fib(1, 1).
fib(N, X1 + X2) :-
    N > 1,
    fib(N - 1, X1),
    fib(N - 2, X2).
```

To compute the 10th Fibonacci number, we can use the goal

```
?- fib(10, Z).
```

while to find out which Fibonacci number is 89, we can use the goal

```
?- fib(X, 89).
```

The next program describes the relationship between two complex numbers and their product. We will represent the complex number `X + iY` as the term `c(X, Y)`.

```
zmul(c(R1, I1), c(R2, I2), c(R3, I3)) :-
    R3 = R1 * R2 - I1 * I2 ,
    I3 = R1 * I2 + R2 * I1 .
```

Any of the following goals will return a unique answer. The first goal asks for the product of two complex numbers, while the other two ask for the result when one complex number is divided by another.

```
?- zmul(c(1, 1), c(2, 2), Z).
?- zmul(c(1, 1), Y, c(0, 4)).
?- zmul(X, c(2, 2), c(0, 4)).
```

Notice how both operations are described using the definition of complex multiplication, rather than writing a separate rule that divides complex numbers by first realizing the divisor and then multiplying. This declarative aspect will be an important feature of many of the programs we look at. Also notice that both of the programs we have seen so far have been invertible in the sense that it did not matter which terms in the goals were ground and which were not. This is a property that we will try to obtain as often as possible when we define programs or parts of programs. As a further example, the special `pow` function can be used to compute powers, roots and logarithms of an arbitrary base. The rules below for square root,

```
sqroot(X, pow(X, 0.5)):-
    X >= 0.
sqroot(X, -pow(X, 0.5)) :-
    X >= 0.
```

state that a non-negative number has a positive and negative square root. Finally consider the following program, which relates the key parameters in a mortgage.

```
mortgage(P, Time, IntRate, Bal, MP) :-
    Time > 0, Time <= 1,
    Bal = P * (1 + Time * IntRate/1200) - Time * MP.
mortgage(P, Time, IntRate, Bal, MP) :-
    Time > 1,
    mortgage(P*(1 + IntRate/1200) - MP, Time-1, IntRate, Bal, MP).
```

The parameters above are principal, life of the mortgage (in months), annual interest rate (%) which is compounded monthly, the monthly payment, and finally, the outstanding balance. The goal

```
?- mortgage(100000, 180, 12, 0, MP).
```

asks the straightforward query as to how much it would cost to finance a $100,000 mortgage at 12 percent for 15 years, and the answer obtained is `MP = 1200.17`. We can ask the question backwards:

```
?- mortgage(P, 180, 12, 0, 1200.17).
```

to obtain the expected answer `P = 100000`, or ask for how long a mortgage is needed:

```
?- mortgage(100000, Time, 12, Bal, 1300).
```

Here we get the answer `Time = 147.365`. The main point of this example, however, is that we can ask, not for the values of, but for the *relationship between* `P`, `MP` and `Bal`. For example,

```
?- mortgage(P, 180, 12, Bal, MP).
```

gives the answer

```
P = 0.166783 * Bal + 83.3217 * MP
```

This particular example illustrates how answer constraints may be viewed as a partial evaluation of the program. In this case, the equation above is the result of partially evaluating the program with respect to `Time = 180` and `I = 12`.

2.3 The Type Issue

Informally, one of the two types in CLP($\mathcal{R}$) is given by the real numbers, and the other by the remaining ground (variable-free) terms. Strictly speaking, CLP($\mathcal{R}$) is a statically typed language in the sense that variables, uninterpreted functors and predicates in a program must be used in a consistent way with respect to their type. That is, each variable and each argument of every predicate and uninterpreted functor is first acknowledged to be of a certain type. The program is then considered to be ill-typed if, for example, a variable appears both in a functor constraint and an arithmetic constraint; similarly, the program is ill-typed if one occurrence of a predicate or uninterpreted functor has a number in the first argument while, in another occurrence, it has a functor term in the first argument.

For programming convenience, however, CLP($\mathcal{R}$) does not perform such type-checking at compile time. This decision is based on the fact that it is often useful to overload a symbol; for example, one may want a database `p` of both numbers and letters:

```
p(1).
p(2).
p(a).
p(b).
```

and one may run a goal containing `p(X)` and some constraints used for selection within the database. Note that by not performing type-checking, one can have a runtime type error. That is, an execution sequence which fails because of a "type clash". Often such failures indicate that there is an error in the program. The CLP($\mathcal{R}$) system will not distinguish such failures from failures obtained from well-typed constraints.

A straightforward way of thinking about the type issue when writing CLP($\mathcal{R}$) programs is that whenever an arithmetic term appears in a rule, for each variable `X` therein, we can implicitly add a corresponding atom `real(X)` to the body of the rule. The system predicate `real/1` is *true* just in case there is a real solution for `X` in the context of the current collection of constraints.

Chapter 3

Programming in CLP($\mathcal{R}$)

3.1 Preliminaries

Before we can look at more advanced programming examples, it is necessary to have some idea of how the programs are executed. This is similar in flavor to the way PROLOG programs are executed, but the basic operational step of unifying an atom with the head of a rule is replaced by something more general. In this preliminary section, we assume that all arithmetic constraints are *linear*; the general case is discussed in a later section.

The computation begins with a goal and an initially empty set of *collected constraints*. The usual left-right atom selection rule is used to select either an arithmetic constraint or an atom at each stage. When a constraint is selected, it is added to the set of collected constraints, and it is determined whether the resulting set has a solution. If there is no solution, backtracking takes place in the usual way. On the other hand, when an atom is selected, the set of rules is searched in the usual top-down fashion, each time matching that atom with the head of some rule. Such a match is realized by an equation between these two atoms; such an equation is treated like any equation between terms.

As before, it is required that the system of constraints collected so far has a solution. In general, solving this equation proceeds at first by unifying the syntactic parts of the terms in the usual way. However, these terms may contain arithmetic terms. As arithmetic terms have a special meaning, they are not unified syntactically, but rather an equation between them is solved in the domain of real arithmetic.

Let us consider some examples. We start with a program that has no explicit constraints or arithmetic terms, effectively written in PROLOG.

```
p(f(c)).
q(g(X)) :-
    p(f(X)).
?- q(Y).
```

As the computation proceeds, the collected constraint set and current goal are as follows:

```
{} ?- q(Y).
{q(Y) = q(g(X)) } ?- p(f(X)).
{q(Y) = q(g(X)), p(f(X)) = p(f(c)) } ?- .
```

Note that only one successful path is shown here. Also, as we will discuss in more detail later, the "answer" to this query is just the set of constraints collected, but "projected" onto the goal variables, in this case `Y`. So the answer to the above query is

```
Y = g(c).
```

Now consider a program that includes both arithmetic terms and explicit constraints:

```
p(10, 10).
q(W, c(U, V)) :-
    W - U + V = 10,
    p(U, V).
?- q(Z, c(X + Y, X - Y)).
```

and again we only look at one successful path of the execution:

```
{} ?- q(Z, c(X + Y, X - Y)).
{q(Z, c(X + Y, X - Y)) = q(W, c(U, V)) } ?- W - U + V = 10, p(U, V).
{q(Z, c(X + Y, X - Y)) = q(W, c(U, V)), W - U + V = 10 } ?- p(U, V).
{··· , p(U,V) = p(10, 10) } ?- .
```

The answer for this derivation is

```
Y = 0, X = 10, Z = 10.
```

and we should notice that, as expected, it does not contain any mention of the variables U, V, and W. Also note that, in general, the answers need not give values to variables, and it is possible to get an answer constraint like

```
X + Y + Z = 0, X > Y.
```

This facility is a very important and useful feature of CLP($\mathcal{R}$) as we will illustrate later.

3.2 Delay of Nonlinear Constraints

In the above discussion of the operational model, we saw how each operational step results in one or more constraints being added to the collected constraint set, and the new set being checked for satisfiability. Because of efficiency requirements, there is a limit to how sophisticated the decision algorithm for constraints can be, and consequently the collected constraint set may get too complicated for the decision algorithm. In particular, consider a case when the collected constraint set is solvable, but one constraint is added that makes the set so complicated that it is not practical to decide whether it has remained solvable.

A naive approach to dealing with this problem is simply to disallow expressions that can result in such complexity. This is tantamount to disallowing all nonlinear constraints. The loss in expressive power is, however, unacceptable. Instead, CLP($\mathcal{R}$) allows nonlinear constraints but keeps them in a *delayed constraint set*. More precisely, at each operational step, instead of blindly adding each constraint to the collected constraint set and incurring the cost of performing a satisfiability test, we remove certain constraints that would make the set too complicated. We keep these removed constraints in the delayed constraint set. Additionally, at each step it is possible that some constraint in the delayed constraint set need no longer be delayed because of new information. In this case it should be moved from the delayed constraint set to the collected constraint set and the usual solvability check made. Note that, in general, the notion of which expressions are "too complicated" is dependent on the implementation. In CLP($\mathcal{R}$) only the nonlinear constraints are delayed.

Now let us consider an example where the collected constraint set is initially empty; then suppose we obtain the constraint

```
V = I * R.
```

This is placed in the delayed constraint set. Continuing, if the next constraint is

```
V = 10
```

it may be added to the collected constraint set, but note that it is still not easy to decide whether the two constraints together are solvable Now consider what happens if the next constraint is

```
R = 5.
```

This gives us enough information to make the delayed constraint linear, so we simply remove this constraint from the delayed constraint set, place it in the collected constraint set, and check that it is solvable, which of course it is. Note that the delayed constraint set may have contained other constraints, which may have to remain there until much later. Also note that because of this delay mechanism, we may continue through a certain computation sequence even though the collected and delayed constraint sets together are not solvable. In the worst case it can result in an infinite loop. This is the price we pay for an efficient decision algorithm.

As we have already stated, in the CLP($\mathcal{R}$) system a linear equation or inequality is always considered to be sufficiently simple to be solved immediately, but nonlinear constraints are delayed until they become linear. This includes the functions `sin/1`, `arcsin/1`, `cos/1`, `arccos/1`, `pow/2`, `max/2`, `min/2` and `abs/1` which are delayed until they become simple evaluations in one direction or another. This means that `sin` and `cos` require the input to be ground, while `pow` requires at least two out of three arguments to be ground, except in cases such as

```
X = pow(Y, Z)
```

where Z $= 0$. The reason is that Y^0 is defined to be 1 for all values of Y. Note that while this is sufficient to determine the value of X, Y remains non-ground. There are similar special cases when Z is 1, and when Y is 0 or 1. The functions `arcsin` and `arccos` are delayed until either the input is ground or the result of the function is ground. They are also different in that they are functions and the input domain for `arcsin` ranges from $-\pi/2$ to $\pi/2$ and `arccos` from 0 to π whereas `sin` and `cos` are defined for any number in radians. Thus `sin` and `cos` behave as relations which is non-invertible while `arcsin` and `arccos` are true functions which are invertible. See Section 5.2 for a more precise definition of the delaying conditions for the different nonlinear functions.

As a final example, consider the `mortgage` program in Chapter 2, and consider the goal:

```
?- mortgage(120, 2, IR, 0, 80).
```

This will give rise to nonlinear constraints, and the system returns a quadratic equation as the answer constraint:

```
80 = (0.1*IR + 40) * (0.000833333*IR + 1)
```

and indicates that this is an unsolved answer. Note that while CLP($\mathcal{R}$) cannot determine whether this equation is solvable, the equation indeed describes the correct answer.

3.3 The CLP($\mathcal{R}$) Operational Model

We now precisely but informally define the operational model of CLP($\mathcal{R}$). A goal `G` is written in the form `C, D ?- E` where `C` is a satisfiable collection of constraints, `D` a collection of nonlinear constraints called the *delayed constraints*, and `E` a sequence of atoms and constraints. In what follows, we define how such a goal is reduced into another in the context of an ongoing derivation.

In reducing a goal `C, D ?- E`, CLP($\mathcal{R}$) either selects an element from `E`, call this a *forward* reduction, or selects a constraint from `D`, call this a *wakeup* reduction. Initially, `C` and `D` are empty, and CLP($\mathcal{R}$) attempts to make a forward reduction.

Forward reductions

If `E` is empty, then we say that the goal is *terminal*, and no more reduction of the goal is possible. If `D` is also empty, then the derivation is *successful*; otherwise, the derivation is *conditionally successful* (depending on the nonlinear constraints).

Now consider the case where `E` is nonempty; let `E0` denote the first element of `E` and let `E2` denote the remaining subsequence of `E`.

If `E0` is an atom, then `E0` will be selected for atom reduction in the manner described above. First, an appropriate program rule will be selected. The atom and rule head will then be matched, giving rise to a collection of constraints, which we will write as `M1 & M2` where `M1` consists only of linear constraints and `M2` only of nonlinear ones. The new goal consists of (a) `C & M1` in its first component; (b) `D & M2` in its second component, and (c) the body of the rule and `E2`, sequenced in this order, in its third component.

If `E0` is a linear constraint, then the reduced goal is `C & E0, D ?- E2` providing `C & E0` is satisfiable; otherwise there is no reduced goal and the derivation is *finitely failed*.

Finally, if `E0` is a nonlinear constraint, then the reduced goal is `C, D & E0 ?- E2`. That is, the constraint `E0` is simply delayed.

Wakeup reductions

Let the goal at hand be `C, D ?- E`. This reduction step starts by considering whether there is a delayed constraint `D0` in `D` which is in fact linear. That is, `C` implies that `D0` is equivalent to a linear constraint. If there is no such delayed constraint, then no reduction is performed.

Otherwise, consider the case in which `C` is inconsistent with this linear constraint. Here reduction is not possible and a finitely failed derivation is obtained. However, if `C` is consistent with the linear constraint, then the reduced goal is `C & D0, D2 ?- E` where `D2` is result of deleting `D0` from `D`.

3.4 Meta–programming

In the context of Prolog, meta–programming refers to the destruction and construction of rules and terms, and the examination and modification of the rulebase. All of the same issues arise in CLP($\mathcal{R}$). However, some extra facilities are needed because of the special nature of arithmetic terms and constraints. Furthermore, some of the remaining ones must be modified. For example, without such extra facilities and modifications, there is no way that a CLP($\mathcal{R}$) program can distinguish the two terms `p(3 - 1)` and `p(1 + 1)` since they are semantically identical.

More specifically, the extra facilities and modifications are needed to:

- make arithmetic terms be interpreted syntactically, by introducing a coded form;
- convert coded forms of arithmetic terms into the appropriate arithmetic terms;
- obtain a coded form of [some projection of] the current constraint set;
- add appropriate constraints to asserted rules;
- examine the rulebase completely syntactically.

3.4.1 `quote/1` and `eval/1`

First we introduce the macro-like operator `quote/1`. This is expanded in an outer-most first fashion when expressions are first read. The argument of the `quote` operator is translated to a version in which all arithmetic operators are translated to a special coded form, which is not otherwise directly accessible to the programmer. This coded form can then be treated like a functor term. In this discussion, such coded forms of arithmetic function symbols will be be represented with a caret over them. For example, the rule

```
p(X, Y, quote(X + Y)).
```

would be read in as

$$\texttt{p(X, Y, X } \widehat{+} \texttt{ Y).}$$

and so on. Furthermore, the `quote` operator passes through all other function symbols, constants, variables etc. without changing them. Thus for example, the rule

```
q(X,Y) :- X = quote(f(g(Y), 2 * Y)).
```

becomes

`q(X,Y) :- X = f(g(Y), 2` $\widehat{*}$ `Y).`

Of course, the original form of the rule is always shown when listing the database, etc., but when printing a term, coded function symbols are printed preceded by a caret[1]. For example, the query `?- q(X, 5).` to the above rule would yield the answer `X = f(g(5), 2 ^* 5)`. Note that that the caret form of coded terms cannot be input directly, but only through the use of `quote`. Additionally, to facilitate manipulating programs which themselves use meta-programming facilities, we need coded forms of the `quote` operator itself, as well as the new `eval` interpreted function symbol, which will be described below. This is why `quote` is expanded outer-most first. For example,

`P = quote(p(quote(X + Y), X + Y))` expands to
    `P = p(` $\widehat{\texttt{quote}}$ `(X` $\widehat{+}$ `Y), X` $\widehat{+}$ `Y)).`

Thus an occurrence of `quote` that appears within the scope of another `quote` will be translated to $\widehat{\texttt{quote}}$, and will not be quote-expanded. The `eval` interpreted function can be coded by using `quote` as well, for example,

`X = quote(eval(1 + 2))` gives
    `X =` $\widehat{\texttt{eval}}$ `(1` $\widehat{+}$ `2).`

Now, the major linguistic feature for meta–programming with constraints is the interpreted function symbol `eval` which converts a coded term to the term it codes. It passes through uninterpreted function symbols, other than those that are coded forms of interpreted ones, without changing them. Likewise for constants and interpreted function symbols. Some examples:

`X = 1` $\widehat{+}$ `2, U = eval(X)` implies
    `U = 3.`
`X = Y` $\widehat{+}$ `Z, U = eval(X)` implies
    `U = eval(Y) + eval(Z).`
`X = Y` $\widehat{+}$ `Z, U = eval(X), Y = 1, Z = 2` implies
    `U = 3.`

The function `eval` has no effect on uninterpreted functors. For example, the goal

```
?- X = f(a, g(c)), U = eval(X).
```

results in both `U` and `X` being `f(a, g(c))`. However,

```
?- X = f(Y, g(c)), U = eval(X).
```

results in `U` being `f(eval(Y), g(c))`, as the "best" representation of terms containing `eval`

[1]In this manual, we take the liberty of placing the caret as an accent for readability

is that with `eval` pushed inwards as far as possible.

Formally, the meaning of `quote` and `eval` are given by the axioms:

$$\begin{array}{lcl}
\texttt{eval}(\ \widehat{f}\ (t_1, \cdots, t_n)) & = & f(\texttt{eval}(t_1), \cdots, \texttt{eval}(t_n)),\ n \geq 0 \\
\texttt{eval}(g(t_1, \cdots, t_n)) & = & g(\texttt{eval}(t_1), \cdots, \texttt{eval}(t_n)),\ n \geq 0 \\
\texttt{eval}(\widehat{\texttt{quote}}(t)) & = & t
\end{array}$$

where f ranges over all arithmetic function symbols, g ranges over all uncoded function symbols different from `eval`, and t, $t_1, \cdots, t_n$ range over terms.

In general, deciding the satisfiability of constraints involving `quote` and `eval` is a non-trivial problem. Consider for example the two equations:

$$\begin{array}{lcl}
f(eval^2(x), eval^2(y)) & = & f(\widehat{\texttt{quote}}(eval^4(y)), \widehat{\texttt{quote}}(eval^3(x))) \\
f(eval^3(x), eval^4(y)) & = & f(\widehat{\texttt{quote}}(eval^2(y)), \widehat{\texttt{quote}}(eval^2(x)))
\end{array}$$

The first of these constraints is solvable, while the second is not. There is in fact an algorithm to deal with such constraints in their full generality. However, for efficiency reasons, CLP($\mathcal{R}$) implements a partial algorithm: maintaining constraints so that `eval` appears only in the form `X = eval(Y)`, *these equations are delayed until the argument of* `eval` *is constructed.* In fact, the delay of such `eval` equations is implemented in much the same way as nonlinear equations.

For example, consider the goal

```
?- X = quote(U + 1), eval(X) = 5, Y = eval(U) - 5.
```

After the first constraint, `X` is equal to `U` $\widehat{+}$ `1`, but after the second constraint, `eval` goes as far through `X` as it can, so we obtain the simplified constraint `eval(U) + 1 = 5`, which is further simplified to `eval(U) = 4`. Hence the third constraint results in `Y` being `-1`.

However, if the goal were permuted to

```
?- eval(X) = 5, Y = eval(U) - 5, X = quote(U + 1).
```

the first and second constraints both result in delayed `eval` constraints. The third constraint wakes the first delayed `eval` since `X` is now constructed, resulting in the constraint `eval(U) + 1 = 5` again, which, together with the second delayed `eval` constraint — which is not awakened — results in `Y` being grounded to `-1` again.

As a final example, consider the goal

```
?- eval(X) + eval(Y) = 4, eval(X) - eval(Y) = 1.
```

which is rather silly in isolation, but could arise as the result of a longer computation. In this case, the answer constraints are `eval(X) = 2.5, eval(Y) = 1.5` although the values of `X` and `Y` cannot be determined uniquely. For example, `X` might be `2.5`, or `1` $\hat{+}$ `1.5`, etc. It should be noted that the `eval` mechanism described here is an approximation to that proposed in [7].

3.4.2 `rule/2`, `retract/1` and `assert/1`

Next we consider how these basic facilities may be used for reasoning about programs (see also Section 4.8 which describes how to use the dynamic code facilities). The canonical application for such reasoning is the meta-circular interpreter, discussed in detail in [7]. Like the `clause/2` predicate of Prolog, we require a system predicate `rule/2` such that the goal `?- rule(H, B)` behaves as if there were facts `rule(E, F)` for each rule `E :- F` in the program (and of course `rule(A, true)` for each fact `A`).

There is, however, one aspect of `rule` which has no analog in `clause`: arithmetic function symbols will become coded. More precisely, the system predicate `rule` behaves as if there were facts `rule(quote(E), quote(F))` for each rule `E :- F` in the rulebase (and `rule(quote(A), true)` for each fact `A`). We note that a direct analog to `clause` can be written in terms of `rule`:

```
analog_to_clause(H, B) :-
    functor(H, Name, Arity),
    functor(H1, Name, Arity), % rule needs a constructed head
    eval(H) = eval(H1),
    rule(H1, eval(B)).
```

In a similar fashion, the CLP($\mathcal{R}$) system predicate `retract/1` is like that in PROLOG but differs in that one matches arithmetic function symbols with their coded forms. As before, a direct analog to the PROLOG's `retract` can be written as follows:

```
analog_to_retract(eval(R)) :-
    functor(R, Name, Arity),
    functor(R1, Name, Arity), % retract needs a constructed argument
    eval(R) = eval(R1),
    retract(R1).
```

Now consider the following example program:

```
(a)     p(1, 1.5).
(b)     p(X, Y) :- Y = 2 * X.
(c)     p(X, 2 * X).
(d)     p(X, 2 + X).
```

The goal `?- retract(quote(p(X, 2*X)))` removes only the rule (c). The goal

```
?- analog_to_retract(p(X, 2*X))
```

on the other hand, should remove rules (c) and (d).

As explained in [7], `assert/1` in CLP($\mathcal{R}$) differs from that in PROLOG not just because of term codings; additional constraints may have to be added to the asserted rule. For example,

```
?- X + Y > 2, assert(p(X, Y)).
```

results in the rule

```
p(X, Y) :- X + Y > 2.
```

As another example, the goal:

```
?-  X + Y = 2, X >= 0, Y - 2*X <= 2, X > W, Y - X >= 1,
    assert(p(X, Y)).
```

asserts the rule:

```
p(X, Y) :- Y = -X + 2, X <= 0.5, -X <= 0.
```

Note that a considerable simplification of the initial constraints has occurred. More generally, this supports a technique of constraint partial evaluation. This technique consists of executing a query, and then using the simplified form of the answer constraints to construct new rules. These new rules represent a specialization of the program with respect to that query. For example:

```
resistor(V, I, R) :- V = I * R.
?-  resistor(V, I1, R1), resistor(V, I2, R2),
    I = I1 + I2,
    assert( parallel_resistors(V, I, R1, R2)).
```

results in the assertion of a rule describing the equivalent voltage-current relationship of a pair of resistors connected in parallel[2]:

[2] The actual names of variables in the rule being asserted will be internally constructed names but we will use the original ones for clarity

```
parallel_resistors(V, I, R1, R2) :-
    V = I2 * R2,
    V = (I - I2) * R1.
```

The facilities we have discussed for adding rules to the database have provided no control over the exact syntax of the rule added. For example constraints may be simplified and/or rearranged before the rule is added. It is particularly important in some applications to have complete control over the syntax of rules added to the database. This control is provided by using a coded form of the rule to be asserted, where `assert` of a coded rule is defined to add the rule that is coded. For example, the goal

```
?- assert(quote( p(X, X + X) :- X - 3 > 0 )).
```

asserts the rule

```
p(X, X + X) :- X - 3 > 0.
```

In contrast, the goal

```
?- assert(p(X, X + X) :- X - 3 > 0).
```

could, for example, add the (semantically equivalent) rule:

```
p(X, Y) :- Y = 2*X, Z = X - 3, Z > 0.
```

3.5 Output

An important feature of the CLP($\mathcal{R}$) system is its ability to output the collected constraints of a successful derivation in a simpler form. In a typical derivation, thousands of constraints may be collected, and printing them out without simplification would lead to an unusable answer. When a derivation succeeds the output module of CLP($\mathcal{R}$) is invoked to print the constraints relating the variables in the goal. The module can also be invoked using the system predicate `dump([X,Y,...,Z])`, discussed later.

The CLP($\mathcal{R}$) system attempts to simplify the constraints in two ways: by projecting the constraints onto a set of *target* variables (those appearing in the original goal or given by the user in the argument of `dump`), and by eliminating redundancy in the constraints. Ideally the output constraints will only involve target variables and be free of redundancy, but this will not always be possible.

Recall that there are constraints of four different forms:

- functor constraints, e.g. `X = f(Y, a, g(Y))`
- linear equations, e.g. `3*X + 4*Y = 6`
- linear inequalities, e.g. `3*X > 4 + Y`
- non-linear equations, e.g. `X = Y * Z, T = pow(U, V), U = eval(V)`[3]

Each of these constraint types is handled differently and in turn.

3.5.1 Outline of Algorithm

In this section, we outline how the output is obtained to give a flavor of the kinds of simplifications and reductions that are possible in the answer constraints.

Functor equations are handled first, and in much the same way as in PROLOG. The constraints are stored in solved form using bindings, and printing the simplest form of each target variable simply involves printing their term representation. For example

```
?- X = f(Y, Z), Z = g(a, Y), dump([X, Y]).
```

results in the output

```
X = f(Y, g(a, Y)).
```

Note that there is no equation for `Y` since it is its own term representation. With functor equations, it is not always possible to present the output in terms of target variables alone, and some non-target variables are printed out using an internal name. For example,

```
?- X = f(Y, Z), Z = g(a, Y), dump([X]).
```

results in an output such as

```
X = f(_h6, g(a, _h6)).
```

Linear equations are used to substitute out non-target variables in the following manner. If E is a linear equation containing non-target variable X, then we rewrite E into the form $X = t$ and substitute t for X in all the other constraints (including functor equations, inequalities and non-linear equations). Consider, for example

```
?- T = 3 + Y, X = 2 * Y + U, Z = 3 * U + Y, dump([X, T, Z]).
```

First, we eliminate Y using the first equation $Y = 3 - T$ and obtain

[3] Delayed constraints involving `eval` are treated like nonlinears.

```
X = 2 * T - 6 + U, Z = 3 * U + T - 3.
```

Then we eliminate U using the the first equation and obtain

```
Z = 3*X - 5*T + 15.
```

This is the final answer since only the variables `X, T` and `Z` remain.

Linear inequalities are more difficult to handle than linear equations. We will not go into the details of how variables can be eliminated from inequalities except to mention that a variation of Fourier-Motzkin elimination [19] with some improvements is used (see [11] for more details). In general, eliminating variables from inequalities can be expensive and the projection can contain an exponential number of inequalities.

We finally deal with the nonlinear equations. In general, the algorithm here simply outputs each nonlinear equation unless it has been used as a substitution. We will not define formally what exactly constitutes a substitution, but will discuss some examples. Recall that each non-linear constraint takes the form $X = Y * Z$, $X = sin(Y)$, $X = cos(Y)$, $X = pow(Y, Z)$, $X = max(Y, Z)$, $X = min(Y, Z)$ or $X = abs(Y)$. Each of these equations can be used to substitute for X if X is a non-target variable. For example,

```
?- Y = sin(X), Y = cos(Z), dump([X,Z]).
```

leads to the output

```
sin(X) = cos(Z).
```

As in the case for functor equations, we cannot in practice eliminate all non-target variables appearing in non-linear constraints. As before, we display any non-target variable using an internal name.

A Complete Example

Consider the goal

```
?-  X = f(V, M), V = a, N = 2 * T, Y = 4 * T, Z = R + T, M = N * R,
    Y + Z >= U, U > T, U >= R + N,
    dump([X, Y, Z]).
```

First we eliminate V by substitution obtaining

```
X = f(a, M), N = 2 * T, Y = 4 * T, Z = R + T, M = N * R,
Y + Z >= U, U > T, U >= R + N
```

Next we eliminate N using the second constraint obtaining

```
X = f(a, M), Y = 4 * T, Z = R + T, M = (2 * T) * R,
Y + Z >= U, U > T, U >= R + 2 * T
```

Next we eliminate T using the second constraint obtaining

```
X = f(a, M), Z = R + 0.25 * Y, M = (0.5 * Y) * R,
Y + Z >= U, U > 0.25 * Y, U >= R + 0.5 * Y
```

Next we eliminate R using the second constraint obtaining

```
X = f(a, M), M = (0.5 * Y) * (Z - 0.25 * Y),
Y + Z >= U, U > 0.25 * Y, U >= Z + 0.25 * Y
```

Next we eliminate U from the inequalities (and here the individual steps taken may not be so obvious), obtaining

```
X = f(a, M), M = (0.5 * Y) * (Z - 0.25 * Y),
0.75 * Y + Z > 0, 0.75 * Y >= 0
```

Finally, we eliminate M using the second constraint, and as output we obtain (after performing some straightforward scaling) the constraints

```
X = f(a, (0.5 * Y) * (Z - 0.25 * Y)),
0 < Z + 0.75 * Y,
0 <= Y
```

We finally remark that we can obtain an empty output using the algorithm just outlined. This indicates that there are no restrictions on the values that the target variables can take. For example,

```
?- T = 3 + Y, X = 2 * Y + U, Z = 3 * U + Y, dump([X, Z]).
```

results in no constraints at all. In such cases, the distinguished predicate `real/1` is then used to indicate that certain variables are arithmetic, and that no further constraints are upon them. In this example, we will output the constraints

```
real(X), real(Z).
```

3.5.2 The `dump` System Predicates

The basic facility for output in CLP($\mathcal{R}$) is the system predicate `dump/1`, mentioned above, whose argument is a list of target variables. Note that, to use this predicate, the target variables must appear explicitly in the argument (as in `dump([A, B])`) and not be passed in (as in `X = [A, B], dump(X)`). This is because the names of the target variables are actually used in the output. The ordering of variables in the list is used to specify a priority on the variables with the later variables having a higher priority. Since `dump` outputs constraints, there are many equivalent forms of the same set of constraints and the priority ordering is used to express higher priority variables in terms of the lower ones. This gives one form of control over the output from `dump`. For example, the goal

```
?- X = 2 * Y + 4, dump([X, Y])
```
gives `Y = 0.5 * X - 2`

whereas the reverse order would give back the original constraint.

The predicate `dump/2` is a refinement of `dump/1`, and is designed to be far more flexible. Its first argument is, as before, a list of target variables. Its second argument is a list of constants to be used in place of the original target variables in the output. For example,

```
?- Names = [a, b], Targets = [X, Y], X > Y, dump(Targets, Names).
```

results in the output `a > b`. This predicate is useful when the names of target variables are known only at runtime. More precisely, the operation of `dump/2` is as follows: let the first and second arguments be the lists $[t_1, \cdots, t_n]$ and $[u_1, \cdots, u_n]$, where the t_i and u_i are *arbitrary* terms. Construct new variables $T_1, \cdots, T_n$, and add to the current collection of constraints the equations $T_1 = t_1, \cdots, T_n = t_n$. Now obtain a projection of the augmented constraints w.r.t. $T_1, \cdots, T_n$. Finally, output this projection renaming each target variable T_i by its new name u_i.

In meta-programming it can be useful to obtain the coded form of the constraints with respect to given target variables. This facility is provided by the system predicate `dump/3`. There are three arguments because it is not sufficient to simply provide the variables to be projected upon (1st argument) and the variable that receives the coded form (3rd argument). The 2nd argument is a list of terms that are to replace the original variables in the coded form, and hence the lengths of the two lists must be the same. For example,

```
?-  NewVars = [A, B, C], Targets = [X, Y, Z], X > Y + Z,
    dump(Targets, NewVars, Answer).
```

results in the binding `Answer = [ ^- A ^+ B ^+ C < 0]`.

There are two reasons for having such a second argument. First, it is very inconvenient to manipulate a coded form containing variables that have the original arithmetic constraints still imposed on them — in particular, printing such a term leads to highly counter-intuitive results. Second, in many cases it is more convenient to manipulate ground representations of the coded forms. That is, with syntactic constants replacing the variables. The terms resulting from manipulation can then have the original (or other) variables substituted into place easily.

We conclude with a larger example. We will assume that the predicate `p/2` sets up a constraint such that the first argument is a (polynomial) function of the second, and that `diff/2` implements symbolic differentiation on coded forms of arithmetic constraints. Then, to find the turning point of the functional relationship established by `p/2`, we can use the following goal:

```
solve(DYDX,X) :- eval(DYDX) = 0.
p(Y, X) :-
    T = X + 1,
    Y = T * T.
?-  p(Y, X),                          % collect a function Y(X)
    dump([Y, X], [V, U], Z),          % get coded form of Y(X)
    Z = [C], C =.. ['=', V, RHS],     % assume Z of the form [V = f(U)]
    diff(RHS, DVDU),                  % symbolic differentiation
    solve(DVDU, U),                   % find extremum
    printf("Turning point: X = %, Y = %\n", [U, V]).
```

3.6 Some Programming Techniques

Here we collect a number of small programs that serve to illustrate some interesting programming techniques.

A Crypto-arithmetic Puzzle

Consider one of the standard crypto-arithmetic puzzles. We require an injective assignment of digits $0, 1, \cdots, 9$ to the letters S, E, N, D, M, O, R, Y such that the equation

```
  S E N D
+ M O R E
---------
M O N E Y
```

holds. The program first imposes certain constraints on the values. Then it tries to assign possible values to the letters. The problem is combinatorially explosive and so a naive generate and test solution would be very inefficient. In contrast, the straightforward program below runs quickly in CLP($\mathcal{R}$).

The program illustrates how CLP($\mathcal{R}$) can be used to advantage in solving problems over integer domains. Because the unsolvability of constraints in $\mathcal{R}$ implies their unsolvability over the integers, CLP($\mathcal{R}$) can prune the search space significantly without the expense of invoking an integer solver. For CLP programs in general, the key issue is the trade-off between the power and the speed of the constraint-solver: powerful solvers entail smaller search spaces but are costlier to run. For CLP($\mathcal{R}$) in particular, the use of a real-number-based solver to approximate constraint-solving over a discrete or finite domain is one important realization of this trade-off.

```
solve([S, E, N, D, M, O, R, Y]) :-
        constraints([S, E, N, D, M, O, R, Y]),
        gen_diff_digits([S, E, N, D, M, O, R, Y]).

constraints([S, E, N, D, M, O, R, Y]) :-
        S >= 0, E >= 0, N >= 0, D >= 0, M >= 0, O >= 0, R >= 0, Y >= 0,
        S <= 9, E <= 9, N <= 9, D <= 9, M <= 9, O <= 9, R <= 9, Y <= 9,
        S >= 1, M >= 1,
        C1 >= 0, C2 >= 0, C3 >= 0, C4 >= 0,
        C1 <= 1, C2 <= 1, C3 <= 1, C4 <= 1,
        M = C1,
        C2 + S + M = O + C1 * 10,
        C3 + E + O = N + 10 * C2,
        C4 + N + R = E + 10 * C3,
        D + E = Y + 10*C4,
        bit(C1), bit(C2), bit(C3), bit(C4).

bit(0).
bit(1).

gen_diff_digits(L) :-
        gen_diff_digits(L, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]).
gen_diff_digits([], _).
gen_diff_digits([H | T], L) :-
        select(H, L, L2), gen_diff_digits(T, L2).
select(H, [H | T], T).
select(H, [H2 | T], [H2 | T2]) :-
        select(H, T, T2).

?- solve(S, E, N, D, M, O, R, Y).
```

Critical Path Analysis

This program uses local propagation to compute start, completion and float times for a project network. Significantly, the constraint paradigm allows the program to compute these values by making only one pass of the project network, as opposed to the three passes that would be needed using a conventional programming language.

Most of the program is basically parsing the input and building an adjacency graph out of the network. Then the latest completion time and earliest starting time for every node is simply the minimum of the time required for the outgoing events and maximum of the time of the incoming events.

```
cpm(Network, Graph, Latest) :-
        build(Network, Graph),
        early_late(Graph, Graph, End, Latest),
        Latest >= End,
        analyse(Graph, Graph).

cpm(Network, Graph) :-
        build(Network, Graph),
        early_late(Graph, Graph, End),
        analyse(Graph, Graph).

% Build adjacency graph out of the network ... build([], Graph) :- ...
% Get early start times and latest completion times
% early/4 is used when a ending time is given
% otherwise early/3 assumes that the early start time
% for the end node is equal to the latest completion time

early_late([], _, _, _).
early_late([ad(I, Es, Lc, To, From) | T], G, End, Latest) :-
        setearly(From, To, G, End, Es),
        setlate(To, G, Latest, Lc),
        early_late(T, G, End, Latest).

early_late([], _, _).
early_late([ad(I, Es, Lc, To, From) | T], G, End) :-
        setearly(From, To, G, End, Es),
        setlate(To, G, End, Lc),
        early_late(T, G, End).

setearly([], _, _, _, 0).
setearly([ed(V, C, _, _, _, _) | T],[], G, Es, Es) :-
        !,
        getnode(V, G, Es1, _),
```

```
        setmax(T, G, Es1 + C, Es).
setearly([ed(V, C, _, _, _, _) | T], _, G, End, Es) :-
        getnode(V, G, Es1, _),
        setmax(T, G, Es1+C, Es).

setmax([], _, Max, Max).
setmax([ed(V, C, _, _, _, _) | T], G, Max0, Max) :-
        getnode(V, G, Es1, _),
        setmax(T, G, max(Max0, Es1 + C), Max).

setlate([], _, Last, Last).
setlate([ed(V, C, _, _, _, _) | T], G, Last, Lc) :-
        getnode(V, G, _, Lc1),
        setmin(T, G, Lc1-C, Lc).

setmin([], _, Min, Min).
setmin([ed(V, C, _, _, _, _) | T], G, Min0, Min) :-
        getnode(V, G, _, Lc1),
        setmin(T, G, min(Min0, Lc1 - C), Min).

% Search graph for the early & late times for a node
getnode(I,[ad(I, Es, Lc, _, _) | T], Es, Lc).
getnode(I,[H | T], Es, Lc) :-
        getnode(I, T, Es, Lc).

% Compute the other times:
%               Ls - latest start time
%               Ec - earliest completion time
%               Tf - total float time
%               Ff - free float time

analyse([], G).
analyse([ad(I, Es, Lc, To, _) | T], G) :-
        analyse_times(To, Es, Lc, G),
        analyse(T, G).

analyse_times([], _, _, _).
analyse_times([ed(V, C, Ls, Ec, Tf, Ff) | T], Esi, Lci, G) :-
        getnode(V, G, Esj, Lcj),
        X = Esi + C,
        Ls = Lcj - C,
        Ec = Esi + C,
```

```
        Tf = Lcj - X,
        Ff = Esj - X,
        analyse_times(T, Esi, Lci, G).

print_analysis(G) :- ...
```

A goal might be

```
?-      cpm([
                [n1, n2, 4], [n1, n3, 3], [n1, n4, 4], [n2, n5, 7],
                [n2, n3, 1], [n2, n7, 8], [n3, n5, 4], [n4, n6, 2],
                [n5, n6, 1], [n5, n7, 3], [n6, n7, 4]], G),
        print_analysis(G).
```

A Simple Circuit Solver

The following program performs DC analysis on circuits containing resistors, voltage sources and diodes. The circuit analysis is decomposed in a hierarchical fashion. The individual components are modelled directly by constraints such as Ohm's law. Then the components are connected together and the global circuit constraints on the currents and voltages, as specified by Kirchoff's laws, are used to define the whole circuit.

```
solve_dc(C, L) :-
        solve(C, [], L),
        solve_current(L).

% solve for every circuit component
solve([], L, L).
solve([[Comp, Name, Par, Nodes] | T], In, Out) :-
        connect(Name, Nodes, Volts, Amps, In, Tmp),
        component(Comp, Par, Volts, Amps),
        solve(T, Tmp, Out).

% sum of currents at each node are zero
solve_current([]).
solve_current([n(N, V, IList) | T]) :-
        kcl(IList, 0),
        solve_current(T).
```

```
kcl([], 0).
kcl([(Name, I) | T], X) :-
        kcl(T, I + X).

% connect the arcs which meet at a node
connect(Name, [], [], [], L, L).
connect(Name, [N | T], [V | VR], [I | IR], In, Out) :-
        add_arc(Name, N, V, I, In, Tmp),
        connecting(Name, T, VR, IR, Tmp, Out).

% create the voltage and currents
add_arc(Name, N, V, I, [], [n(N, V, [(Name, I)])]).
add_arc(Name, N, V, I, [n(N, V, IList) | T],
                       [n(N, V, [(Name, I) | IList]) | T]).
add_arc(Name, N, V, I, [X | T], [X | T1]) :-
        add_arc(Name, N, V, I, T, T1).

component(resistor, R, [V1, V2], [I, -I]) :-
        V1 - V2 = I*R.
component(voltage_source, V, [V, 0], [I, -I]).
component(diode, in914, [V1, V2], [I, -I]) :-
        diode(in914, [V1, V2], [I, -I]).
diode(in914, [V1, V2], [I1, I2]) :-
        V = V1 - V2, V < -100,  DV = V+100,  I1 = 10*DV - 0.1.
diode(in914, [V1, V2], [I1, I2]) :-
        V = V1 - V2, V >= -100, V < 0.6, I1 = 0.001*V.
diode(in914, [V1, V2], [I1, I2]) :-
        V = V1 - V2, V >= 0.6, DV = V - 0.6, I1 = 100*DV - 0.0006.
```

A sample query which returns the currents and voltages in L

```
?-      R1 = 100, R2 = 50, V = 20,
        solve_dc([[voltage_source, v1, V, [n1, ground]],
                  [resistor, r1, R1, [n1, n2]],
                  [resistor, r2, R2, [n2, ground]],
                  [diode, d1, in914, [n2, ground]]], L).
```

Chapter 4

Using the System

The user interface of compiled CLP($\mathcal{R}$) is very much like that of a usual Edinburgh-style Prolog interpreter. In other words, it is quite possible to use this system while almost completely ignoring the fact that it is compiler-based. In fact, there is no such thing as an interpreted mode and all code (static and dynamic) is compiled. All goals are compiled (quickly) before being executed, and any consulted file is immediately compiled. The rulebase is available for inspection (except for protected rules) and can be modified dynamically as long as the relevant relations have been declared to be *dynamic* as described below. Normally the user will find that consulted files take a little longer than usual to be read in (because they are being compiled) and that programs will usually run much more quickly and use less space than in an interpreter. Symbolic debugging is still possible, as are all other aspects of interactive programming. However, the user may also take special advantage of the compiler by creating `clam` files that contain compiled CLP($\mathcal{R}$) code that can be loaded extremely quickly and do not include the overhead of the original program text, although this rules out certain operations. In short, the system is intended to get the best of both worlds by combining the flexibility of an interpreter with the efficiency of a compiler. The experienced PROLOG user may want to skip directly to Section 4.6 which illustrates many of the features, syntax and user interface of CLP($\mathcal{R}$) using an example session.

Note: Creation of CLAM files has not yet been implemented. However, compilation in CLP($\mathcal{R}$) is relatively quick.

The first operation CLP($\mathcal{R}$) performs is to load the distinguished library file `init.clpr`. This file must either be in the current working directory, or in a directory whose path name is defined via the environment variable `CLPRLIB`, or in a directory whose path name is specified during installation. This last alternative is explained in Chapter 6.

4.1 Command Line Arguments

The syntax of a command-line is

`clpr` [*options*] [*filename*]

where *filename* contains a CLP($\mathcal{R}$) program. The following explains the various *options* available:

`-cs <n>`
Specify size of code space (default 128,000).

`-hs <n>`
Specify size of heap (default 200,000).

`-ls <n>`
Specify size of local stack (default 100,000).

`-ss <n>`
Specify maximum number of solver variables (default 128,000).

`-ts <n>`
Specify size of trail (default 100,000).

`-z <r>`
Set internal notion of zero to this small number. Numbers between $\pm r$ are taken to be equivalent to zero.

`-r <int>`
Specify a random number seed.

4.2 Filenames

Filenames consulted or read as an input stream may have an optional implicit suffix added to the filename. The default suffix is usually ".clpr" (".clp" for MS/DOS or OS/2) depending on the installation. This may be changed by the use of the environment variable CLPRSUFFIX, which can be set to a list of suffixes separated by colons, e.g. ".clpr:.clp". First, the original filename is tried and if that cannot be read then a suffix is added in the order specified by the list of suffixes. (Note that in version 1.1 and earlier of CLP($\mathcal{R}$), only the specified filename was used without any implicit suffixes, but the behavior here is compatible).

4.3 Queries

After the system has been initialized, it will prompt the user for a query. It will continually accept user goals and solving for them until the session is terminated with a `halt/0` or if it encounters the end of file (eg: `^D` on UNIX or `^Z` on MSDOS). This again is similar to the style of most PROLOG systems. If the user goal failed then the "`*** No`" message is output, otherwise the query is successful and the resulting answer constraints (the constraints on variables in the query) are output. A successful query will also display a "`*** Yes`" message, but if there are other alternatives to try for the query then the "`*** Retry?`" message is displayed and the user is prompted to either press carriage return or enter "." (or "n") to accept the answers, or ";" (or "y") to cause backtracking. A different prompt is displayed if delayed (nonlinear) remain at the end of the execution. The message "`*** Maybe`" replaces "`*** Yes`" and "`*** (Maybe) Retry?`" replaces "`*** Retry?`" to indicate that the satisfiability of the nonlinear constraints remaining has not been decided by CLP($\mathcal{R}$). Execution of a query can be interrupted at any time by using the interrupt keycode (`^C` usually).[1] A buffer of the last 50 goals is kept, and may be examined by the `history/0` (or `h/0`) predicate. An old query may be re-executed by just entering its history number as a goal (eg: `?- 5.`).

For every top-level query. There is also an *implicit dump* on the variables in the goal, i.e. the set of answer constraints using those variables are printed, with the exception that anonymous variables and also other variables beginning with an "_" are ignored. No implicit dump is performed for goals embedded in a file. (Note that the output constraints differs from many PROLOG systems which display the variable bindings produced from execution.)

4.4 Loading/consulting and reconsulting programs

A CLP($\mathcal{R}$) source program can be loaded using the `consult/1` predicate or the more convenient notation `[`<*list of filenames*>`]`, e.g. `[myprog, mytest]` at the top level prompt loads those two files. Loading a program compiles all the rules in that file, makes the new predicates in it available for use and also executes any embedded goals. Unlike some PROLOG systems where consulted files are interpreted and compilation is done using a different method, all consulted predicates in CLP($\mathcal{R}$) are compiled (usually fairly quickly). Note that filenames may have an implicit suffix added as in Section 4.2. Filenames which are specified directly should consist entirely of lowercase characters and any other kind of filename, eg. a pathname, should be surrounded by single quotes.

Reconsulting a file with `reconsult/1` or the notation `['`<*list of filenames*>`]` will if it

[1] It is however not absolutely safe to interrupt at any time, and occasionally at critical stages an interrupt may cause the system to be internally inconsistent

encounters previous definitions, erase them and replace them by the new definitions. By default, a predicate which is redefined will generate a warning. This may be turned off by executing the system predicate `warning(redefine_off)`. Some PROLOG systems use an alternative notation `[-`*filename*`]` but in CLP($\mathcal{R}$) this conflicts with unary minus. Also in some systems, consulting and reconsulting are combined together. In CLP($\mathcal{R}$) consulting a previously consulted file with active definitions will result in warning messages and redefinitions will be ignored.

The special filename `user` denotes that the file to be consulted or reconsulted is read from standard input. This allows direct entry of rules which is handy for quick modifications from the top query level. More on the organization of consulted files is contained in Section 4.7.

4.5 Style Checking and Warnings

CLP($\mathcal{R}$) programs can be optionally checked against some stylistic conventions, also called style checking. The purpose of the style checking is to give a warning that the program may potentially contain some common "bugs" when the style rules are not followed. It is important to remember that these are merely warnings and a program may be perfectly correct otherwise. There are three different kinds of style checking that can be applied — `single_var`, `discontiguous`, `name_overload`.[2] The option `all` covers all three styles. By default, style checking is on and individual style checking can be turned on (off) with `style_check/1` (`no_style_check/1`), e.g. `no_style_check(all)` turns off all style checking.

The different style conventions are as follows:

`single_var` — This warns if a variable is used only once within a rule and may possibly indicate that a variable has been mispelled. Anonymous variables (`_`) and also variables prefixed with an underscore are ignored. An example error is the rule "`p(X, Y)`" gives the following warning message:

```
    Warning: Style check, singleton variables, rule 1 of q/2
+++ X, Y
```

`discontiguous` — This style check assumes that all the different rules defining a predicate occur contiguously within a file and warns if there is another intervening rule. Common bugs which can result when this style check is not followed can be mispelling the name of a rule, or substituting a "." to end a rule when a "," was meant to continue the rule, e.g. the program "`p(X) :- X > 0. q(X). p(0) :- r(X).`" where there the intent is for a comma to be before `q/1` gives the following warning message:

[2]The first two options are similar to that in Quintus Prolog. The last is different.

```
Warning, <stdin>:1: Style check, p/1 is not contiguous
```

`name_overload` — This checks whether the same predicate name is defined with different arities. While it is not uncommon to have different predicates of different arities with the same name, it may also be indicative of an incorrect number of arguments, e.g. the program "p(0,0). p(1). p(2,2)." gives the following warning message:

```
Warning: rule overloading, same name, different arity:
+++ p/1, p/2
```

(Note that when this option has been disabled and then re-enabled, then rules which were defined before style checking was enabled will also generate warnings. The additional warnings can be disabled by using the special system predicate `$clear_style_check/0`. `style_check(all_reset)` also does this, clearing all previous warnings and turns on style checking.)

Another kind of warning is given when a rule is defined in more than one file. The basic unit of compilation is a single file and all the occurences of rules for a predicate have to be defined within the same file. The exception is that when a file is being reconsulted, then the new definitions replace the old ones. The compiler will simply ignore all additions to an existing previously compiled predicate and by default a warning is given. See also `warning/1` to control whether warnings are given.

4.6 Sample Session

This is a sample session with the CLP($\mathcal{R}$) system. Some extra information is given using comments after the % character.

```
% clpr

CLP(R) Version 1.2
(c) Copyright International Business Machines Corporation
1989 (1991) All Rights Reserved

1 ?- f(X,Y) = f(g(A),B).        % some simple ``unification''

B = Y
X = g(A)

*** Yes
```

```
2 ?- X = Y + 4 , Y = Z - 3, Z = 2.      % simple arithmetic evaluation

Z = 2
Y = -1
X = 3

*** Yes

3 ?- X + Y < Z, 3 * X - 4 * Y = 4, 3 * X + 2 * Y = 1.

Y = -0.5
X = 0.666667
0.166667 < Z

*** Yes

4 ?- X + Y < Z, 3 * X - 4 * Y = 4, 2 * X + 3 * Z = 1.

Y = -1.125*Z - 0.625
X = -1.5*Z + 0.5
-0.0344828 < Z

*** Yes

5 ?- history.
1       f(X, Y) = f(g(A), B).
2       X = Y + 4, Y = Z - 3, Z = 2.
3       X + Y < Z, 3 * X - 4 * Y = 4, 3 * X + 2 * Y = 1.
4       X + Y < Z, 3 * X - 4 * Y = 4, 2 * X + 3 * Z = 1.

*** Yes

6 ?- 2.                                  % run second goal again
X = Y + 4, Y = Z - 3, Z = 2.

Z = 2
Y = -1
X = 3

*** Yes
```

```
7 ?- ['examples/fib'].          % consult (load) a program

>>> Sample goal: go/0

*** Yes

8 ?- ls fib.                    % look at the program

fib(0, 1).
fib(1, 1).
fib(N, X1 + X2):-
        N > 1,
        fib(N - 1, X1),
        fib(N - 2, X2).

*** Yes

9 ?- fib(5,F).                  % only one answer to this

F = 8

*** Retry?;

*** No

10 ?- F > 7, F < 9, fib(N,F).  % only ask for the first answer

N = 5
F = 8

*** Retry?
11 ?- [`'examples/mortgage'].  % use "`" to reconsult

>>> Sample goals: go1/0, go2/0

*** Yes

12 ?- ls.               % look at the entire rulebase

h:-
        history.
```

```
fib(0, 1).
fib(1, 1).
fib(N, X1 + X2):-
        N > 1,
        fib(N - 1, X1),
        fib(N - 2, X2).

go:-
        printf(\nFib(14) = , []),
        ztime,
        fib(14, X),
        ctime(T1),
        printf(% (Time = %)\n, [X, T1]),
        printf(Fib-1(610) = , []),
        ztime,
        fib(Y, 610),
        ctime(T2),
        printf(% (Time = %)\n, [Y, T2]).

mg(P, T, I, B, MP):-
        T = 1,
        B = P + P * I - MP.
mg(P, T, I, B, MP):-
        T > 1,
        mg(P * (1 + I) - MP, T - 1, I, B, MP).

go1:-
        ztime,
        mg(999999, 360, 0.01, 0, M),
        ctime(T),
        printf(Time = %, M = %\n, [T, M]).

go2:-
        ztime,
        mg(P, 720, 0.01, B, M),
        ctime(T),
        printf(Time = %\n, [T]),
        dump([P, B, M]).

*** Yes

13 ?- [‘’examples/mortgage’].
```

```
Warning: mg/5 has been redefined

>>> Sample goals: go1/0, go2/0

*** Yes

14 ?- ls.

h:-
        history.

fib(0, 1).
fib(1, 1).
fib(N, X1 + X2):-
        N > 1,
        fib(N - 1, X1),
        fib(N - 2, X2).

go:-
        printf(\nFib(14) = , []),
        ztime,
        fib(14, X),
        ctime(T1),
        printf(% (Time = %)\n, [X, T1]),
        printf(Fib-1(610) = , []),
        ztime,
        fib(Y, 610),
        ctime(T2),
        printf(% (Time = %)\n, [Y, T2]).

mg(P, T, I, B, MP):-
        T = 1,
        B = P + P * I - MP.
mg(P, T, I, B, MP):-
        T > 1,
        mg(P * (1 + I) - MP, T - 1, I, B, MP).

go1:-
        ztime,
        mg(999999, 360, 0.01, 0, M),
        ctime(T),
        printf(Time = %, M = %\n, [T, M]).
```

```
go2:-
        ztime,
        mg(P, 720, 0.01, B, M),
        ctime(T),
        printf(Time = %\n, [T]),
        dump([P, B, M]).

*** Yes

15 ?- go2.
Time = 0.25

M = -7.74367e-06*B + 0.0100077*P

*** Retry?

16 ?- [user].
p(X) :- writeln(X).
^D
*** Yes

17 ?- p(hello).
hello

*** Yes
```

4.7 Organization of Consulted Files

Slightly more care than usual must be taken in organizing program files in compiled CLP($\mathcal{R}$). A file consists of a number of *chunks*. Each chunk consists of a zero or more rules (defined in the usual way) possibly followed by a goal. That is, a goal always closes off a chunk, and the end of the file closes off the last chunk if a goal has not done so. A relation may not span more than one chunk unless it has been declared to be *dynamic* (see below) before the first rule defining it. Defining a relation statically in more than one chunk will generate a warning message stating that the new definitions will be ignored is given. However if one is reconsulting then the new definitions will replace the ones defined in the previous chunk. A warning message that the redefinition has taken place is also given. However, if such a redefinition during a reconsult is not possible when the earlier definition has been protected (using the `prot/2` predicate), in which case a warning is printed and the new definition is ignored. The motivation for this restriction is that the state of the rulebase needs to be well defined whenever a goal is encountered in the consulted file.

There may be three kinds of goals in any consulted file. All three kinds are considered to be identical (and behave in the usual way) when they are encountered in a source file that is being consulted. However, they are different when a source file is first compiled and when the `.clam` file is consulted. All goals of the form `:- goal` are only executed during the compilation stage. Those of the form `::- goal` are only executed during the consultation of the compiled code, and the goals of the traditional form `?- goal` are executed twice: once during compilation and once during consultation. In summary:

`:- goal.`
: *is executed during compilation of the source file.*

`::- goal.`
: *is executed during consultation of the* `.clam` *file.*

`?- goal.`
: *is executed during compilation and at runtime.*

The first kind of goal might be used for compiler directives and messages to whoever is watching while some code is being compiled. The second kind might be used for making a program run itself straight after it is loaded. Finally, the third kind of goal is useful for things like operator declarations which need to be present for the remainder of a program to parse correctly and also when the program is running so that terms will print correctly, etc. An embedded goal that fails during execution will generate a warning message (see also `warning/1`).

4.8 Static and Dynamic Code

A CLP($\mathcal{R}$) program is divided into static rules, which do not change, and dynamic rules, which allow the rulebase to be modified via `assert/1` and `retract/1` as well as by consulting. As mentioned above, static rules/code cannot span more than one chunk. Dynamic code on the other hand can be defined anywhere and dynamic rules can be added by asserting them during execution or by consulting a program file, which behaves as if those definitions were asserted. The only requirement for rules intended to be dynamic is that the particular predicate name has to be pre-declared using `dynamic/2` which ensures that all uses of this predicate are now dynamic, e.g. `?- dynamic(foo, 2).` The first argument is the name of the predicate and the second is its arity[3]. Every dynamic declaration has to occur before any use of a dynamic predicate is made (including `rule`, `assert` and `retract`), otherwise an

[3]Most PROLOG's use the name/arity convention to specify this but this could be confused with division, hence the two argument form is used

error is generated with any of the preceeding system predicates and any use of that predicate is assumed to be static. Declaring a predicate to be dynamic allows the use of `rule/2` to inspect the rulebase, `assert/1` to add new rules and `retract/1` to delete rules.

The operational semantics of the assert, rule and retract family of system predicates is that any modifications to the rulebase occur immediately and are immediately available for use[4]. This is called the immediate update view [17]. Consider the following example:

```
?- dynamic(p,0).
p :- assert(p), fail.
```

This will cause the goal "`?- p.`" to succeed. Apart from the dynamic declaration and the immediate update semantics, there is no difference between static and dynamic code and they may be used interchangeably, e.g. both can be listed with `ls/1`. Dynamic code is also compiled but is generally not as efficient as static code and also less deterministic. Also note that the semantics of `assert`, `rule` and `retract` are an enhancement of that in PROLOG (see Section 3.4.2).

4.9 Debugging Support

The debugging facilities in this version of CLP($\mathcal{R}$) are rudimentary.

`codegen_debug`
: This is a compiler directive, which includes debugging instructions in subsequently generated code. It should be active before the file to be debugged is consulted.

`codegen_nodebug`
: This is a compiler directive that turns off the generation of debugging code in subsequent compilation.

`spy`
: This switch makes all relations compiled under `codegen_debug` visible to the debugger. Protected rules are never visible.

`spy(+P, +A)`
: This switch makes the relation for predicate `P` with arity `A` visible to the debugger if it was compiled under `codegen_debug`. It cannot be applied to protected relations.

[4] The operational semantics of dynamic code may vary considerably between different PROLOG systems hence one should not place undue reliance on it.

`spy([P1(+A1),...,Pn(+An)])`
Like `spy/2`, except a list is supplied of the predicates to be spied on where the `Pi`'s are the predicate names and the `Ai`'s their arity.

`nospy`
Makes all relations invisible to the debugger.

`nospy(+P, +A)`
Makes the relation for predicate `P` with arity `A` invisible to the debugger.

`nospy([P1(+A1),...,Pn(+An)])`
Like `nospy/2`, except a list is supplied of the predicates to be spied on where the `Pi`'s are the predicate names and the `Ai`'s their arity.

`trace`
Activates printing. All subsequent attempts to search a relation visible to the debugger will result in a message being printed. The message is the same regardless of whether this is a first or subsequent attempt to satisfy a goal.

`notrace`
De-activate printing.

4.10 Notes on Efficiency

Here we indicate some key features that can significantly affect efficiency. Some of them are unsound in general, and hence extreme care should be taken when using them. Novice programmers may (and probably should) skip this section entirely.

- *Indexing*
  CLP($\mathcal{R}$) employs first argument indexing for constructed functor terms as well as real numbers. Using indexing can result in significant speedups.

- *Tail recursion*
  Last call optimization is employed, and hence procedures that are tail-recursive will not increase local stack usage.

- *Logical disjunction "`;/2`" and if-then-else "`->/2`"*
  These are implemented at the meta-level and hence are not particularly efficient.

- *Dynamic code*
  Dynamic code is slower than static code and is also less deterministic. Cuts can be used to make it more deterministic. Also, since dynamic code is compiled, asserting large terms may not be very fast.

- *Garbage collection*
  Not implemented as yet.

- *Implicit equalities*
  The solving of inequalities that imply some implicit equations can be controlled using `implicit/0`, `noimplicit/0`, `partial_implicit/0` (see Section 5.1.7).

- *Asserting a rule*
  The predicate `assert/1` involves incorporating the constraints that relate the variables in that rule (see Section 3.4.2). This is less efficient than if the constraints were not taken into account. The `fassert` family of special predicates ("fast assert") performs assertion without incorporating arithmetic constraints (see Section 5.1.7), as in PROLOG.

4.11 Notes on Formal Correctness

The following identifies the main reasons why the CLP($\mathcal{R}$) implementation does not perfectly conform to the idealized CLP scheme.

- No occurs check during (functor) unification;
- Depth-first search (loss of completeness);
- Floating point: because this implementation of CLP($\mathcal{R}$) makes use of double precision floating point arithmetic, some problems may be caused by artifacts such as roundoff. The most common problem is that a constraint used as a test (in that all variables are ground) unexpectedly fails because of round-off. This is dealt with by adjusting the amount of slack that the system allows in numerical comparisons, using the `-z` command line option.
- Nonlinear and meta-level constraints are delayed.

Chapter 5

Built-In Facilities

5.1 System Predicates

5.1.1 Rulebase

`op(+P, +T, +S)`

Declares the atom `S` to be an operator of type `T` with precedence `P`. The type can be used to specify prefix, postfix and binary operators using the positional notation: `fy, fx, yf, xf, yfy, xfy, yfx, xfx`; where the "`f`" specifies the operator and the "`y`" and "`x`" the arguments. A "`y`" specifies that the topmost functor/operator in the subexpression be of the same or lower precedence than the operator "`f`", and "`x`" specifies that it is to be strictly lower. The precedences must range between $\{0 \ldots 1200\}$. where a 0 precedence removes the operator.

(See also Section 5.3 for some examples.)

`listing`
`ls`

List the rules of the entire rulebase that are currently visible.

`listing +P`
`ls +P`

List the currently visible rules for the predicate `P`, of all arities.

`consult(+F)`
`[+F]`

Read the file `F` and add rules that it contains to the database. Goals in the file are handled in a way that is described in Section 4.7. If the filename is specified as `user` then the standard input is used instead of a file. The form `[F]` takes a list of filenames while `consult/1` takes only a single file. When the file `F` cannot be read then a possible list of file suffixes is added using the CLPRSUFFIX environment variable (see Section 4.2). By default, a ".clpr" file extension is used. (Not currently implemented: If the file has a `.clam` extension it is expected to be clam code and is loaded appropriately. If it has no extension and a version with a `.clam` extension exists it is given preference.)

`reconsult(+F)`
`[‘+F]`
Same as `consult`, but if a predicate already has rules defining it from before, they are deleted before the new ones are added, and a warning message is printed. Note that `[-F]`, which is a common synonym for `reconsult` in PROLOG systems, cannot be used (since it means negative `F`).

`retractall`
Delete entire unprotected portion of the rulebase.

`retractall(+H)`
Delete all currently visible rules with heads matching `H`. Static code cannot be deleted with `retractall/1`.

`asserta(+R)`
Add rule `R` to the rulebase before all others defining the same predicate. Note that coded terms become uncoded in the rulebase. See Section 3.4.2 for more information on meta-coding of rules and differences with the usual PROLOG semantics.

`assertz(+R)`
`assert(+R)`
Add rule `R` to the rulebase after all others defining the same predicate. Note that coded terms become uncoded in the rulebase. See Section 3.4.2 for more information on meta-coding of rules and differences with the usual PROLOG semantics.

`rule(+H,?B)`
True if the rule `H:-B` is in the currently visible part of the rulebase. Finds the next matching rule on backtracking. Note that the rules in the rulebase are coded before matching is done. See Section 3.4.2 for more information on meta-coding of rules and differences with the usual PROLOG semantics.

`deny(+H,?B)`
Delete rule matching `H :- B` from the currently visible part of the rulebase. Also tries again on backtracking. It is similar to `retract/1` and both `H` and `B` are coded terms. See Section 3.4.2 for more information on meta-coding of rules and differences with the usual PROLOG semantics.

`retract(+R)`
Delete rule matching `R` from the currently visible part of the rulebase. Like `rule/2`,

this has a "coded view" of the rulebase. See Section 3.4.2 for more information on meta-coding of rules and differences with the usual PROLOG semantics.

`prot(+P,+A)`
Protect all rules for predicate `P` with arity `A` in the rulebase. This makes them look like system predicates to the user. In particular, they cannot be listed, asserted or retracted.

`prot([P1(+A1),...,Pn(+An)])`
Same effect as `prot/2` described above, but takes a list of predicate names `Pi` with arities `Ai` in parentheses.

5.1.2 Control

`!`
The dreaded cut. As usual, its use is not recommended. It is often more appropriate to use `once/1`.

`fail`
Always fails.

`true`
Always succeeds.

`repeat`
Always succeeds, even on backtracking.

`+B1 , +B2`
Logical conjunction.

`+B1 ; +B2`
Logical disjunction. A cut inside one of these will behave very strangely. That is, it will behave as if the two sides of the ";" are separate rules. (Note that because `;/2` is currently implemented as a meta call it may sometimes not behave as if it was defined using an auxiliary predicate. This can occur if there is an arithmetic term that causes failure. The following short example illustrates the difference between `try/3` and `try1/3` for the goal `?- try(X, 1, 0)`,

```
try(X, Y, Z) :- X=Y/Z ; X=1.
try1(Y/Z,Y,Z). try1(1,Y,Z).
```

This may possibly change to be the same in some future version.)

`+C -> +B1 ; +B2`
If `C` then call `B1` otherwise call `B2`. Uses unsafe negation. Inefficient, since it uses `call/1`. A cut inside one of these will behave very strangely.

5.1.3 Meta Level

`call(+X)`

Usual meta level call, behaving as if the predicate `X` appeared directly in the body of a rule or goal. Note that this form must be used – it is not permissible to simply put a variable in the body of a rule. Both static and dynamic code can be used with `call`. In this version, a cut inside a `call` is ignored. Also, `printf/2` and `dump/1` cannot be used inside `call`. Both these restrictions can be avoided by simply redefining them using a subsidiary rule.

`not(+X)`

Unsafe negation. It is implemented using `call/1`, so it is also likely to be rather slow.

`dump(+L1, ?L2, ?L3)`

Similar to `dump/2` (see Section 3.5.2); the first argument `L1` represents the target variables and the second argument `L2` represents new variables. The difference with `dump/2` is that (a) the projection is meta-coded (cf. Section 3.4), and (b) this projection is not output but rather constructed as the third argument `L3` (cf. Section 3.5.2). Note that `dump/3` does change the current collection of constraints.

`once(+X)`

This is equivalent to `call(X), !` and unfortunately right now it is implemented that way as well. Only the first answer to the query `X` is considered.

`nonground(?X)`

True if `X` is not a ground term.

`ground(?X)`

True if `X` is a ground term.

`nonvar(?X)`

True if `X` is not a variable: i.e, it has been constructed or grounded.

`var(?X)`

True if `X` is a variable. It may have been involved in an arithmetic constraint, but has not been grounded or constructed.

`?X == ?Y`

True if `X` and `Y` are bound to exactly the same term. In particular, variables in equivalent positions must be identical. For example `?- X == Y` fails while `?- X = Y, X == Y` succeeds.

`atom(?X)`

True if `X` is an atom — that is, a functor constant (including the empty list).

`atomic(?X)`
True if `X` is an atom or real number.

`functor(?X)`
True if `X` is constructed with a functor.

`real(?X)`
Enforces a constraint that `X` can take real values; it is equivalent to any tautologous arithmetic constraint involving `X`, eg: `X + 0 = X`.

`arithmetic(?X)`
True if `X` is constrained to have a real value. Note that this is just a passive test, as opposed to `real/1`.

`?T =.. ?L`
`T` is a term and `L` is the term expanded as a list. (Also known as `univ/2`). This predicate can be used to both decompose and construct terms. For its use either the first argument must be constructed (a nonvar), or the second argument must be a list of fixed length whose first element is a functor constant.

`functor(?T, ?F, ?A)`
`T` is a term, `F` and `A` are the name and arity of the principle functor of `T`. Either `T` must be constructed or `F` must be a functor constant (not a real number) and `A` must be a nonnegative integer.

`arg(+N, +T, ?A)`
`A` is the $\mathtt{N}^{th}$ argument of term `T`. `N` must be a positive integer and `T` a compound term. If `N` is out of range the call fails.

`occurs(-V,?T)`
`V` is a variable occurring in term `T`.

`floor(+R, -I)`
`R` must be a real number, and `I` is the largest integer smaller than or equal to `R`.

`dynamic(+P,+A)`
Declares the predicate `P` with arity `A` to be dynamic, so that rules can be added and deleted at will.

5.1.4 Input/Output

In this section, non-ground variables will either be printed with a specified name (like that in the argument of `dump/1`), or if one is not specified they are printed in one of the following formats:

`_h%d`
: Heap variable.

`_s%d`
: Local stack variable.

`_t%d`
: Parametric variable in solver.

`_S%d`
: Slack variable in solver.

Input/Output facilities are as follows.

`dump(+L)`
: List the collection of current constraints on the current output stream, projected with respect to the target variables in the list `L`. The list `L` must be explicitly supplied, that is, it is written syntactically as the argument of `dump`. The ordering of variables in the list is used to represent the priority of the target variables (see Section 3.5.2).

`dump(+L1, +L2)`
: A more flexible version of `dump/1`, without its syntactic restriction. Its first argument `L1` represents the target variables, and its second argument `L2`, which must be ground, represents the new names to be used in the output. The elements of these two lists can be arbitrary terms. (See Section 3.5.2 for further explanation.) Note that `dump/2` does not change the current collection of constraints.

`nl`
: Send a newline character to the current output stream.

`print(?T)`
`write(?T)`
: Print the term `T`, according to `op` declarations, on the current output stream.

`writeln(?T)`
: The same as `write(T), nl`.

`printf(+F,+L)`
: Print the terms in the list `L` on the current output stream in the format given by the string `F`. The behavior is similar to the `printf` library function in C. Every character except for the special escape or argument patterns will be printed unchanged on the output. The special escape characters begin with a "\" and are:

XXX	the character represented by the octal number *XXX*
`\n`	a new line
`\r`	carriage return
`\b`	backspace
`\f`	form feed
X	any other character *X* appears unchanged

The argument patterns all begin with "%" and are used to denote the formatting for each of corresponding terms in the list `L`. A "%%" denotes a single percent. Otherwise the format takes the form of an optional field width and optional precision followed by one of the C printf conversion characters. More precisely this can be described with the regular expression:

```
%[[-][0-9]*][\.[0-9]*][fegdoxcus%]
```

The integral specifiers will print the real number, which has been rounded to an integer using the "even" rounding rule. An empty list is needed if no variables are to be printed. As a convenience, a single "%" may be used instead of a specific argument format and a default format appropriate to that particular argument will be used (with numbers the default is printf format "%g"). For example,

```
printf("X = % Y =%3.2g\n", [X, Y]).
```

`printf_to_atom(?A, +F, +L)`
: Like `printf/2` except that instead of being printed `A` is equated with an atom whose string is the same as what would otherwise be printed.

`read(-X)`
: Read a term from the current input and bind the variable X to it. Any variables in the input term are deemed to be disjoint from variables appearing in the rule. If an end of file is read, the term `?-(end)` is returned. Finally, the term obtained is in quoted form. That is, any arithmetic operators are treated syntactically.

`see(+F)`
: Make `F` the current input file.

`seeing(?F)`
: True when `F` is the current input file.

`seen`
: Close current input file. Revert to "user" (standard input).

`tell(+F)`
: Make `F` the current output file.

`telling(?F)`
: True when F is the current output file.

`told`

Close current output file. Revert to "user" (standard output).

`flush`

Flush the buffer associated with the current output file.

5.1.5 Unix-Related Facilities

`fork`

Split the current process. Fails in one child and succeeds in the other. Not available under MS/DOS[1] and OS/2. [2]

`pipe(+X)`

Create a pipe named `X`. For use with `see`, `tell`, etc. Not available under MS/DOS or OS/2.

`edit(+F)`

Invoke the default editor on file `F`, and then reconsult the file. Under UNIX[3] the default is "`vi`", under MS/DOS and OS/2 it is "`edit`". If the environment variable EDITOR is set then that is used instead.

`more(+F)`

Run the file `F` through the "`more`" utility or what the environment variable PAGER has been set to.

`halt`

Exit from the CLP($\mathcal{R}$) system.

`clpr`

True. Used to test if the program is executing in the CLP($\mathcal{R}$) system.

`abort`

Abort execution of the current goal.

`sh`

Invoke an image of "`sh`" on UNIX systems. On MS/DOS or OS/2, starts a sub-shell of "`command.com`" or what the environment variable COMPSEC as been set to.

`csh`

Invoke an image of "`csh`" under UNIX systems. On MS/DOS or OS/2 behaves the same as `sh/0`.

[1] MS/DOS is a trademark of Microsoft Corporation

[2] OS/2 is a trademark of IBM corporation

[3] UNIX is a trademark of Bell Laboratories.

`oracle(+F,+P1,+P2)`
Run the executable binary file `F` and set up a pipe `P1` for writing to the process and a pipe `P2` for reading from the process. These pipes will be attached to the processes standard input and standard output respectively. Not available on MS/DOS or OS/2.

5.1.6 Miscellaneous Facilities

`history`
Print last 50 command line goals.

`history +N`
Print last `N` command line goals.

`h`
Short for `history/0`.

`N`
Run the command line goal at position `N` in the history list. This may only be used as toplevel command.

`new_constant(+A, +N)`
Sets the numeric symbolic constant `A` to the value `N`. The constant name is specified without a "#", e.g. `?- new_constant(my_constant, 5)`. A warning is printed if the value of a known constant is changed and the warning can be turned off with `warning(warning_off)`.

`srand(+X)`
Set random number seed to the real number X.

`rand(-X)`
Generate uniformly distributed random number 0 and 1 inclusive and bind it to `X`. The quality of the routine used is not guaranteed.

`ztime`
Zero the CPU time counter.

`ctime(-T)`
Binds `T` to the elapsed CPU time since the counter was last zeroed. `T` should have been uninstantiated.

`style_check(+A)`
Style checking warns about possible program errors. It is to be used with `A` being one of `single_var`, `discontiguous`, `name_overload` and `all`. A warning is given when the style check rule is violated. The option `all` turns on both the checks. The special

option `reset_all` clears all previous pending warnings which may have accumulated if style checking was off and turns on full style checking. See Section 4.5 for details.

`no_style_check(+A)`
The reverse of `style_check/1` and turns off the corresponding options `single_var`, `discontiguous`, `name_overload` and `all`.

`$clear_style_check`
Clears any pending old style check warnings that may occur when style checking is turned from off to on. Usually it is reasonable not to need to use this and this is more meant for special uses.

`warning(+A)`
The behavior when an error occurs can be modified with `warning/1`. By default, when an error occurs a warning error message is printed and execution is aborted back to the top level. The various options for warning change this behavior. The options for `A` must be one of `abort`, `continue`, `warning_on`, `warning_off`, `redefine_on` and `redefine_off`. The options `continue` or `abort` control whether or not execution is aborted back to the top level on an error. The printing of warning messages is controlled by `warning_on` and `warning_off`, while `redefine_on` and `redefine_off` control whether or not redefinitions of predicates during a reconsult issue a warning. The option `abort` overrides `warning_on` and warning messages are displayed when `abort` is active. Otherwise the paired options here behave indepently.

5.1.7 Special Facilities

These are unsupported facilities which may be used to gain more efficiency under certain circumstances or are experimental in nature. They should be used with care and may change or disappear.

`fassert(+R)`
Like `assert/1` but it does not take into account meta-level constraints or arithmetic constraints and is like assert in PROLOG. Consequently it is faster than `assert/1` but makes less sense when there are constraints involved. When the rules are ground, fassert behaves the same as assert.

`fasserta(+R)`
`fassertz(+R)`
Ditto for `asserta/1` and `assertz/1`.

`$call(+X)`
Meta level call on a single user-defined predicate only. No compound goals or system predicates are allowed.

`implicit`

Implicit equalities are detected. This is the default. A set of inequalities can sometimes be equivalent to some equations; and these are known as implicit equalities. A trivial example of an implicit equation is the following:

`X >= 0, X <= 0` is equivalent to `X = 0.`

The `implicit/0` flag controls whether these implicit equations are detected by the constraint solver. One caveat to note with the use of these flags is that switching them on or off should be applied betweem different goal executions and not during an actual execution. Another important point is that, when there are nonlinear constraints, turning off implicit equations may lead to delayed constraints not being awakened.

`noimplicit`

Turns off detection of implicit equalities. These are equations which are implied by the collection of inequality constraints. The implication of this is that delayed constraints which would otherwise be awakened may continue to be delayed instead. Constraint solving may or may not be faster with `noimplicit`.

`partial_implicit`

Detects only some implicit equalities. This may be faster than `implicit`.

`set_counter(+C, +V)`

This is a global counter which is not changed by backtracking. Sets the counter with the atomic name `C` to the real number value `V`. The counter name can be any atomic name.

`counter_value(+C, ?V)`

`V` is equated with the value of counter `C`.

`add_counter(+C, +V)`

The counter `C` is incremented by `V`.

5.2 Nonlinear and Delayed Constraints

This section describes the form of the delaying conditions for examples of the various nonlinear constraints given below. In some of the functions below, `sin`, `arcsin`, `cos`, `arccos`, there will be values of `X` and `Z` which fall outside the range of that function. Such invalid values will cause the constraint to fail and by default a "Out of range" value is generated. See `warning/1`.

`Z = X * Y`

Delays until `X` or `Y` is ground.

`Z = sin(X)`
Delays until `X` is ground.

`Z = arcsin(X)`
Delays until `X` or `Z` is ground.

`Z = cos(X)`
Delays until `X` is ground.

`Z = arccos(X)`
Delays until `X` or `Z` is ground.

`Z = pow(X, Y)`
Delays until (a) `X` and `Y` are ground, or (b) `X` and `Z` are ground, or (c) `X` $= 1$, or (d) `Y` $= 0$, or (e) `Y` $= 1$.

`Z = abs(X)`
Delays until (a) `X` is ground, or (b) `Z` $= 0$, or (c) `Z` is ground and negative.

`Z = min(X, Y)`
Delays until `X` and `Y` are ground.
(A proper implementation, delaying until `X` $\leq$ `Y` or `X` $\geq$ `Y`, may come later.)

`Z = max(X, Y)`
Similar to the above.

`Z = eval(X)`
Delays until `X` is constructed.

5.3 Pre-Defined Operators

```
::- op(21, fy, '-').
::- op(21, yfx, *).
::- op(21, yfx, /).
::- op(31, yfx, (-)).
::- op(31, yfy, +).
::- op(37, xfx, <).
::- op(37, xfx, <=).
::- op(37, xfx, >).
::- op(37, xfx, >=).
::- op(40, xfx, =).
::- op(40, xfx, =..).
::- op(40, xfx, is).
```

```
::- op(50, fx, `).
::- op(51, xfy, (.)).
::- op(60, fx, alisting).
::- op(60, fx, als).
::- op(60, fx, h).
::- op(60, fx, history).
::- op(60, fx, lib).
::- op(60, fx, libdir).
::- op(60, fx, listing).
::- op(60, fx, ls).
::- op(60, fx, not).
::- op(60, fx, once).
::- op(252, xfy, ',').
::- op(253, xfy, ;).
::- op(254, xfy, (->)).
::- op(255, fx, (:-)).
::- op(255, fx, (::-)).
::- op(255, fx, (?-)).
::- op(255, xfx, (:-)).
```

Chapter 6

Installation Guide

Here we discuss how CLP($\mathcal{R}$) can be made to run on a particular computer system. For installation details on MS/DOS or OS/2, please refer to the appropriate README in the DOS directory.

6.1 Portability

This version of compiled CLP($\mathcal{R}$) should be easily portable to 32-bit computers running some reasonable variant of the UNIX operating system. In most cases, all that will be necessary is for the installer to edit the `Makefile` to specify the machine and operating system, choose the C compiler, optimization level and name of the CLP($\mathcal{R}$) executable file, and run `make`.

6.1.1 Pre-defined Installation Options

The `Makefile` for CLP($\mathcal{R}$) contains definitions of the environment variables `CC`, `CFLAGS`, `EXEC` and `OPTIONS`. They should be checked before installation and adjusted as follows.

- `CC` is just the name of the C compiler to be used to compile the CLP($\mathcal{R}$) system. It is almost always reasonable to leave this as `cc`, although many machines now have more efficient (and more correct) C compilers available. Information about these can be obtained from your system administrator.
- `CFLAGS` specifies switches of the above C compiler that need to be used. While various C compilers have their own range of switches that might have to be used to make

such a large program compile and run, in most cases only the optimization level will be needed here. This will almost always be `-O` but higher optimization levels may be available. Also special flags may sometimes be necessary to utilize the full performance of the native floating point hardware. However, it is important to realize that *many* optimizing C compilers have bugs that are only triggered by compiling a large program at a high optimization level. For this reason, the first attempt to install CLP($\mathcal{R}$) should be made without invoking the C compiler's global optimizer. This usually involves just leaving the `CFLAGS` field blank.

- `EXEC` specifies the name of the CLP($\mathcal{R}$) binary to be generated. We recommend `clpr`.
- `LIBPATH` specifies the default directory for the startup file `init.clpr`. It should be set to the directory in which CLP($\mathcal{R}$) is installed.
- `OPTIONS` is used to specify the hardware and operating system. A number of predefined options are available, which often need to be used in combinations.

  `BSD` Always set if system is running Berkeley Unix, or MACH, or Ultrix. This is also to be set for NEXT machines.

  `AIX` Always set if system is running IBM's AIX operating system.

  `SYS5` This broadly indicates that some version of System V Unix is being used. Should also be used in combination with `AIX` flag and if the operating system is Hewlett-Packard's HP/UX.

  `IBMRT` This indicates that the system is an IBM RT/PC.

  `RS6000` This indicates that the system is an IBM RS/6000.

  `HP835` This is needed for the Hewlett-Packard RISC workstations – especially the 9000 series model 835. Note that it is not appropriate for those HP workstations based on Motorola processors.

  `MIPS` Needed for machines with MIPS CPU, such as SGI machines and the DECStation 3100.

  `MSDOS` Set for 386 or 486 PC's running MS/DOS. See the file "DOS/README.DOS" for details.

  `OS2V2` Set for 386 or 486 PC's running OS/2 2.0 (IMPORTANT NOTE: CLP($\mathcal{R}$) version 1.2 will not work on OS/2 1.x because that only supports 16-bit addressing). See the file "DOS/README.OS2" for details.

  So, for example an IBM RS/6000 running AIX would need the definition

  ```
  OPTIONS = -DAIX -DSYS5 -DRS6000
  ```

One parameter which may have to be changed to ensure that the CPU timing is correct on machines running System V Unix is the Hertz rate which determines the unit of time measured. Typically this value of `HZ` is either 60 or 100. The default value is 60 and otherwise it should be added to the OPTIONS line in the Makefile, eg. -DHZ=100, (on the RS/6000 the default is 100hz).

6.1.2 Customized Installation

When CLP($\mathcal{R}$) starts up it performs some consistency checks on some of the default values in the startup. In particular, on failure to startup it may recommend that the definition of `PMASK` be changed in `emul.h`. If that does not work or if a fatal installation error was reported then you may have an unusual operating system problem which cannot be easily fixed by the installer, and it may be best to contact the authors.

While there are various system limits, these are mostly parameterized and can be changed either directly on the command line or by recompiling with new values for the limits. Most of the limits are contained in the file `config.h`, and some of them will be described below. The parameters which are not listed below may be more dangerous to change arbitrarily.

Pre-defined constant	Meaning
DEF_CLP_SUFFIX	default suffix for CLP($\mathcal{R}$) program files
INITFNAME	default bootstrap file
DOS_TMP_FILE	name of temporary file used only under MSDOS or OS2
DEFAULT_EPS	default value of for zero
DEF_CODE_SZ	default maximum size of code space
MAX_GOAL_CODE	default maximum size of code for a top-level goal
DEF_LSTACK_SZ	default maximum stack size
DEF_HEAP_SZ	default maximum heap size
DEF_TRAIL_SZ	default maximum trail size
DEF_SOLVER_SZ	default maximum number of solver variables
MAX_DUMP_VAR	maximum number of variables for dump
MAX_PROJ_NUM	maximum number of real constants in dump
MAX_IMPLICIT	maximum number of implicit equations detected by a constraint

The stack, code, heap and trail sizes; value of zero; and the number of solver variables can all be changed from the command line (see Section 4.1).

6.2 Basic Configuration

The only file CLP($\mathcal{R}$) system needs to read on startup is `init.clpr`. It always looks for this file in the directory specified at runtime by the environment variable `CLPRLIB`, defaulting to either the current working directory or what `LIBPATH` has been specified as in the `Makefile`.

The only other environment variable which one may want to change is to add your own list of file suffixes with the environment variable `CLPRSUFFIX`. The format is in the style of the UNIX `sh` `PATH` variable.

Chapter 7

Bug Reports and Other Comments

Please address all correspondence to

Joxan Jaffar, H1-D48
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
U.S.A.
(joxan@watson.ibm.com, joxan@yktvmh.bitnet)

Bibliography

[1] Tod Amon and Gaetano Borriello. An approach to symbolic timing verification. In *Tau '92: 2nd International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems*, Princeton, NJ, March 1992.

[2] Tod Amon and Gaetano Borriello. An approach to symbolic timing verification. In *Proc. 29th ACM/IEEE Design Automation Conference*, pages 410–413, Anaheim, CA, USA, June 1992.

[3] Christoph Brzoska. Temporal logic programming and its relation to constraint logic programming. In *Logic Programming: Proceedings of the 1991 International Symposium*, pages 661–677, October 1991.

[4] Michael M. Gorlick, Carl F. Kesselman, and Daniel A. Marottaand D. Stott Parker. Mockingbird: A logical methodology for testing. *Journal of Logic Programming*, 8(1 & 2):95–119, January/March 1990.

[5] James A. Harland and Spiro Michaylov. Implementing an ODE solver: a CLP approach. Technical Report 87/92, Department of Computer Science, Monash University, Victoria, Australia, June 1987.

[6] Nevin Heintze, Spiro Michaylov, and Peter Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the 4th International Conference*, pages 675–703, Melbourne, Victoria, Australia, May 1987. MIT Press. Also to appear in Journal of Automated Reasoning.

[7] Nevin Heintze, Spiro Michaylov, Peter Stuckey, and Roland Yap. On meta-programming in CLP($\mathcal{R}$). In Ewing Lusk and Ross Overbeek, editors, *Logic Programming: Proceedings of the North American Conference, 1989*, pages 52–68. MIT Press, October 1989.

[8] D. S. Homiak. A constraint logic programming system for solving partial differential equations with applications in options valuation. Master's project, DePaul University, 1991.

[9] Joseph C. Tobias II. Knowledge representation in the Harmony intelligent tutoringsystem. Master's thesis, Department of Computer Science, University of California at Los Angeles, 1988.

[10] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.

[11] Joxan Jaffar, Michael Maher, Peter Stuckey, and Roland Yap. Output in CLP($\mathcal{R}$). In *Proceedings of the 1992 Conference on Fifth Generation Computer Systems, Tokyo*, 1992.

[12] Joxan Jaffar and Spiro Michaylov. Methodology and implementation of a CLP system. In Jean-Louis Lassez, editor, *Logic Programming: Proceedings of the 4th International Conference*, pages 196–218, Melbourne, Australia, May 1987. MIT Press. Revised version of Monash University technical report number 86/75, November 1986.

[13] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, July 1992.

[14] Joxan Jaffar, Spiro Michaylov, and Roland Yap. A methodology for managing hard constraints in CLP systems. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation*, pages 306–316, Toronto, Canada, June 1991.

[15] Sivand Lakmazaheri and William J. Rasdorf. Constraint logic programming for the analysis and partial synthesis of truss structures. *Artificial Intelligence for Engineering Design, Analysis, and Manufacturing*, 3(3):157–173, 1989.

[16] Catherine Lassez, Ken McAloon, and Roland Yap. Constraint logic programming and options trading. *IEEE Expert, Special Issue on Financial Software*, 2(3):42–50, August 1987.

[17] T.G. Lindholm and R. A. O'Keefe. Efficient implementation of a defensible semantics for dynamic prolog code. In *Logic Programming: Proceedings of the 4th International Conference*, pages 21–39. MIT Press, May 1987.

[18] Igor Mozetič and Christian Holzbaur. Integrating numerical and qualitative models within constraint logic programming. In *Logic Programming: Proceedings of the 1991 International Symposium*, pages 678–693, October 1991.

[19] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley and Sons, 1986.

[20] L. Sterling and E. Y. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[21] T. Sthanusubramonian. A transformational approach to configuration design. Master's thesis, Engineering Design Research Center, Carnegie Mellon University, 1991.

[22] Roland Yap Hock Chuan. Restriction site mapping in CLP($\mathcal{R}$). In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 521–534, Paris, France, June 1991. MIT Press.

[23] Ricky Yeung. Mpl - a graphical programming environment for matrix processing based on logic and constraints. In *IEEE Workshop of Visual Languages*, pages 137–143. IEEE Computer Society Press, October 1988.

[24] Edward K. Yu. MODIC: A program for model-based diagnosis that uses constraint logic programming. Master's thesis, Department of Computer Science, University of South Carolina (Columbia), 1991.

Index

Appendix A

Differences from the Monash Interpreter

Here we only list those facilities from the Monash interpreter that are not supported.

- The issue of string handling has not yet been settled.
- Predicate definitions cannot be indiscriminately spread over a number of files.
- There is no automatic variable generation for answer projection; there is no `dump/0` predicate.
- Goals result in a prompt for alternate solutions whenever there is a choice point, regardless of whether there are variables in the goal.
- No profiling; the predicates `prof/0` and `noprof/0` are not available.
- Sytem warnings are controlled differently using `warning/1`.
- Linear inequalities are always decided immediately rather than delayed; the predicates `ineq/0` and `noineq/0` are not available.
- Statistics are now available through special system predicates, so the `stats/0` system predicate, while may exist, is not supported.
- There is no `is/2` predicate.