The CLP($\mathcal{R}$) Programmer's Manual: Version 1.1

Nevin Heintze[†]
Joxan Jaffar[‡]
Spiro Michaylov[†]
Peter Stuckey[§]
Roland 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 Science
University of Melbourne
Parkville, Victoria 3052, Australia

October 29, 1997

Contents

1 Introduction **3**

2 Syntax and Simple Examples **4**

3 Programming in CLP($\mathcal{R}$) **7**
3.1 Basic Operational Model . . . 7
3.2 Delay Mechanisms . . . 8
3.3 Meta–programming . . . 10
3.4 Additional Example Programs . . . 17
3.4.1 Crypto-arithmetic Puzzle . . . 17
3.4.2 Critical Path Analysis . . . 19

4 Using the System **25**
4.1 Command Line Arguments . . . 25
4.2 Queries . . . 26
4.3 Sample Session . . . 28
4.4 Organization of Consulted Files . . . 33
4.5 Debugging Support . . . 33
4.6 Arithmetic Precision . . . 34
4.7 Notes on Efficiency . . . 34

5 Built-In Facilities **35**
5.1 System Predicates . . . 35
5.1.1 Rulebase . . . 35
5.1.2 Control . . . 36
5.1.3 Meta Level . . . 37
5.1.4 Input/Output . . . 38
5.1.5 Switches . . . 39
5.1.6 Unix-Related Facilities . . . 40
5.1.7 Special Facilities . . . 40
5.1.8 Missing Predicates . . . 41
5.2 Interpreted Functors . . . 42
5.3 Pre-Defined Operators . . . 42

6 Differences from the Monash Interpreter 44

Chapter 1

Introduction

The CLP($\mathcal{R}$) language is an instance of the Constraint Logic Programming scheme defined by Jaffar and Lassez [?]. 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, although the book by Sterling and Shapiro [?] can serve as a suitable introductory text. Further technical information on CLP($\mathcal{R}$) is available on language design and implementation [?, ?], meta-programming [?] and delay mechanisms [?]. Additionally, much has been written about applications in electrical engineering [?], differential equations [?], options trading [?], music theory [?] etc.

This document is both an introductory tutorial and reference manual describing IBM's compiler-based implementation of CLP($\mathcal{R}$). 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 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.

Chapter 2

Syntax and Simple Examples

A program is a collection of rules. The definition of a rule is similar to that 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 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.

An arithmetic expression (also called an arithmetic term) can also be used in constructing terms, as parts of atoms. For example,

```
X
3.14159
X + Y
sin(X + 2.0)
( X + Y ) / 4
```

are all valid arithmetic terms. However,

```
f(3)
c + 5.0
tan(f(3))
```

are not. Furthermore,

```
X > 5.0
```

```
X + Y + Z = 3
X <= Y
X = V
3 = tan(X)
1.234 + X < Y
```

are all valid constraints, while

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

are not. The following terms, some of which contain arithmetic terms, are valid:

```
f(c)
f(X)
f(3.14159)
g(22, h(4))
f(X + 3)
d
```

but the following are not valid terms

```
f(X) + g(X)
a - 3 .
```

Now we will look at some simple example programs without considering how their execution differs from that of PROLOG programs. 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 represent the complex number `X + iY` as `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. Even the special functions `sin`, `cos`, `tan`, and others are designed with this in mind. For example the constraint

```
X = tan(Y)
```

serves both to compute the tangent of Y and to compute $\mathtt{Y} = \mathrm{Tan}^{-1}\mathtt{X}$ while the latter expression cannot be expressed directly as a constraint.

Similarly, the `pow` function can be used to compute powers, roots and logarithms of arbitrary base. For example, we may define the rules

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

which state that a non-negative number has a positive and negative square root.

Chapter 3

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

3.1 Basic Operational Model

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 flavour 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. 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, backtrack takes place in the usual way. On the other hand, when an atom is selected, te set of rules is searched in the usual top-down fashion, each time matching that atom with the head of some rule. The atom and head are considered to be terms, and an equation between them is added to the set of collected constraints. As before, it is required that the system of constraints collected so far has a solution. In general, deciding 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 compuation 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 the 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. So the answer to the above query is

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

Now consider a more general program, which includes both arithmetic terms and explicit constraints:

```
p(10, 0).

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 the 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).`
$\{\cdots, p(U, V) = p(10, 10)\}$ `?- .`

The answer for this derivation is

```
X = 5, Y = 5, W = 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 Mechanisms

A knowledge of how CLP($\mathcal{R}$) programs are executed, and especially how and when variables are instantiated, is useful for a number of reasons. As is the case for PROLOG, some procedures of a program may have infinite derivations unless certain variables are instantiated.

Furthermore, in the CLP($\mathcal{R}$) system we need to be aware of the kinds of constraints that can and cannot be solved. To understand these aspects of CLP($\mathcal{R}$) fully, it is necessary to understand how the solution of constraints may be delayed by the system.

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 the requirement that an efficient interpreter be available, 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 which makes the set so complicated that it is not practical to decide whether it has remained solvable. One approach to dealing with this problem might be to disallow expressions that can result in such complexity. However, in CLP($\mathcal{R}$), because of the requirement for generality, such expressions are kept in a *delayed constraint set*. At each operational step, instead of blindly adding each constraint to the collected constraint set, we remove any constraints that would make the set too complicated, and keep them 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 the notion of which expressions are "too complicated" is dependent on the implementation. The particular distinction for CLP($\mathcal{R}$) will be discussed later in this section.

First let us consider the example where the collected constraint set is empty, and the constraint obtained is

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

Then 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 — we are assuming that the best we can do is solve simultaneous linear equations. 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 place this delayed constraint into 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.

In the CLP($\mathcal{R}$) system, a linear equation or inequality is always sufficiently simple to be solved immediately, as a solution in parametric form is held internally.

Finally, the functions `sin`, `cos`, `tan`, `pow`, `max`, `min` and `abs` are delayed until they become simple evaluations in one direction or another. This means that sin, cos and tan require either the input or output to be ground, and `pow` requires 2 out of 3 to be ground, except in cases such as

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

where Z = 0. The reason is that Y^0 is 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.

3.3 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 arrise in CLP($\mathcal{R}$), but the special nature of arithmetic terms and arithmetic constraints results in some extra facilities being needed, and requires some of the remaining ones to be modified. The 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

First we introduce the macro-like operator `quote`. This is expanded in an outer-most 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 is an *uninterpreted* function symbol. 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

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

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

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

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 and surrounded by single quotes. For example, the query `?- q(X, 5).` to the above rule would yield the answer `X = f(g(5), 2 '^*' 5)`.

Additionally, to facilititate manipulating programs which themselves use meta-programming facilities, we need coded forms of the `quote` operator itself, and the new `eval` interpreted function symbol, which will be described below. This is why `quote` is expanded outer-most first. For example,

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

Quote-expansion occurs in outer-most first order so that an occurrence of `quote` which appears within the scope of another `quote` will be translated to $\widehat{\texttt{quote}}$, and will not be quote-expanded.

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. However, it is delayed until its argument is constructed. So, 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` is that with `eval` pushed inwards as far as possible. Now consider a more useful goal

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

here after the first constraint X is equal to `U` $\hat{+}$ `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`. Of course, if the goal is permuted to

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

the final answer is the same. However, the first and second constrints 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 woken — 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 copmputation. 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.

As we shall see later, the CLP($\mathcal{R}$) system incorporates a system predicate `dump/1` which prints the constraints on the list of variables provided as its argument. More generally, in meta-prograaming it can be useful to obtain the coded form of the constraints on a number of variables. For this we need another 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. There are two reasons for this. 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 counterinuitive 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.

Let us consider an 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),
        /*** Collects a function Y(X) ***/
    dump([Y, X], [V, U], Z),
        /*** Accesses the coded form of Y(X) ***/
    Z =.. ['=', V, RHS],
        /*** Assumes Z is one equation V = f(U) ***/
    diff(RHS, DVDU),
        /*** Symbolic differentiation ***/
    solve(DVDU, U),
        /*** Finds extremum ***/
    printf("Turning point: X = %, Y = % \n",[U,V]).
```

Next we consider how these basic facilities may be used for reasoning about programs. The canonical application for such reasoning is the meta-circular interpreter. We begin by discussing the simple ("vanilla") meta interpreter and then consider extensions to this interpreter that provide (in varying degrees) their own constraint solving algorithms.

Like the `clause/2` predicate of Prolog, we need the 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`.) Then the basic meta-interpreter may be written:

```
goal(true).
goal((A, B)) :-
        goal(A),
        goal(B).
goal(X) :-
        constraint(X).
goal(X) :-
        rule(X,Z),
        goal(Z).

constraint(A = B):-
        A = B.
constraint(A > B):-
        A > B.
/* similarly for <, <=, >= */
```

The goal `?- g.` is solved by running `?- goal(g).` There are two important points here. First, this interpreter utilizes the constraint solver of the underlying CLP($\mathcal{R}$) system – it takes no control over how the constraints are solved. Second, the interpreter deals correctly with programs that contain occurrences of `quote` and `eval`.

Since `rule` returns terms rather than coded terms, it cannot be used to examine the rulebase structurally. This essentially prevents us from exercising any control over the constraint solving process. To overcome this restriction we need to be able to obtain coded forms of terms in the rule base. For this reason we need the system predicate `quoted_rule/2` which behaves as if there were facts `quoted_rule(quote(E),quote(F))` for each rule `E :- F` in the rulebase (and `quoted_rule(quote(A),true)` for each fact `A`). We note that `rule/2` can be written in terms of `quoted_rule/2` :

```
rule(eval(H),eval(B)) :-
        quoted_rule(H,B).
```

We now modify the basic meta-interpreter to use `quoted_rule` instead of `rule` as follows: (We note that for pragmatic reasons an implementation of `quoted_rule` would require that its first argument be constructed, requiring minor alterations to this meta-interpreter.)

```
goal(true).
goal((A, B)) :-
        goal(A),
        goal(B).
goal(X) :-
        constraint(X).
goal(X) :-
        constraint(X = Y),
        quoted_rule(Y,Z),
        goal(Z).

constraint(A = B):-
        eval(A) = eval(B).
constraint(A > B):-
        eval(A) > eval(B).
/* similarly for <, <=, >= */
```

In this case the goal `?- g.` is solved by running `?- goal(quote(g)).` Again, this interpreter may be used to execute programs which use the `quote` and `eval` facilities, including itself. (We would, of course, require an extra rule to execute the system predicate `quoted_rule` .) It is now possible to extend this interpreter to give more control over the solving of constraints by replacing the rules for `=` , `>=` , `<=` , `>` , `<` by new constraint solving algorithms.

It is desirable to be able to selectively delete rules from the database on the basis of the coded form of the database as well as the actual database. For this reason we need the additional predicate `quoted_retract/1` to complement `retract/1`. For example consider the following 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(p(X, 2*X))` removes rules (c) and (d). However the goal `?- quoted_retract(quote(X, 2*X))` removes only the rule (c). Rule (b) could be removed with the goal `?- retract(p(X, Y) :- W)` . The relationship between `retract` and `quoted_retract` is identical to that between `rule` and `quoted_rule` . In particular, `retract/1` can be implemented in terms of `quoted_retract/1` :

```
retract(eval(R)) :- quoted_retract(R).
```

In the presence of constraints, the definition of `assert/1` is somewhat unclear. The essential question is how to treat arithmetic constraints on the variables in the term to be asserted. For example, consider the goal

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

To motivate our definition of `assert` we examine the behavior of `assert` in Prolog in terms of constraints. Consider the following Prolog program and query:

```
p(f(U, d), e).
q(f(c, Z)).

?- p(X, Y), q(X), assert( r(X) :- s(Y) ).
```

The expected result in Prolog is that the rule

```
r(f(c,d)) :- s(e).
```

is added to the database, since this corresponds to the variable bindings when `assert` is called. Equivalently, the call to `assert` may be viewed as the call `assert(r(X) :- s(Y))` in the presence of the constraints

```
X = f(U,d)  &  Y = e  &  X = f(c,Z).
```

Simplifying these constraints, and projecting onto the variables occurring in the argument to `assert` , we have

```
X = f(c,d)  &  Y = e.
```

If we combine the rule `r(X) :- s(Y)` with a representation of these constraints we obtain:

```
r(X) :- X = f(c,d), Y = e, s(Y).
```

which is semantically equivalent to the expected result in Prolog. In Prolog, constraints may always be represented as a substitution, and so the explicit appearance of constraints in an asserted rule is unnecessary. For example we may represent the constraints { `X = f(c,d), Y = e.` } as the substitution { `X/f(c,d), Y/e` }, and on applying this to the rule, obtain the expected Prolog result. However this is not the case for arithmetic constraints, and so the explicit appearance of constraints in an asserted rule is necessary.
We therefore propose that $\texttt{assert(h :- b}_1, \cdots, \texttt{b}_n\texttt{)}$ adds the rule

$$\texttt{h :- c}_1, \cdots, \texttt{c}_m, \texttt{b}_1, \cdots, \texttt{b}_n\,.$$

to the database, where $\texttt{c}_1, \cdots, \texttt{c}_m$ are constraints representing the projection of the current collected constraint set with respect to the variables in $\texttt{h :- b}_1, \cdots, \texttt{b}_n$.
For example, the goal `?- X + Y > 2, assert( p(X, Y) )` , given earlier, asserts the rule:

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

As another example, consider the goal:

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

which asserts the rule:

```
p(X,Y) :- X + Y = 2, Y >= 1.5, Y <= 2.
```

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 voltage-current relationship of a pair of resistors connected in parallel:

```
parallel_resistors(V, I, R1, R2) :-
        I = V / R1 + V / R2.
```

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 constrast the goal `?- assert( p(X, X + X) :- X - 3 > 0 )` could, for example, add the (semantically equivalent) rule:

```
p(X, Y) :- Y = 2*X,  Y > 6.
```

3.4 Additional Example Programs

Here we collect a number of medium-sized programs that serve to illustrate some interesting program techniques.

3.4.1 Crypto-arithmetic Puzzle

Consider one of the standard crypto-arithmetic puzzles. We require an assignment of digits 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 axioms are used a generators of possible values. The problem is so combinatorially explosive that a naive generate and test solution could not be contemplated, while the CLP($\mathcal{R}$)program runs quite quickly, although the speed depends on the ordering of the generating axioms, `dig` and `carry`.

```
solve(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,
        D + E = Y + 10*C1,
        C1 + N + R = E + 10*C2,
        C2 + E + O = N + 10*C3,
```

```
        C3 + S + M = O + 10*M,
        carry(C1, C2, C3),
        dig(S), dig(E), dig(N), dig(D), dig(M), dig(O), dig(R), dig(Y),
        difflist([S, E, N, D, M, O, R, Y]).

carry(0, 0, 0).
carry(0, 0, 1).
carry(0, 1, 0).
carry(0, 1, 1).
carry(1, 0, 0).
carry(1, 0, 1).
carry(1, 1, 0).
carry(1, 1, 1).

dig(0).
dig(1).
dig(2).
dig(3).
dig(4).
dig(5).
dig(6).
dig(7).
dig(8).
dig(9).

difflist([X | T]) :-
        notmem(X, T),
        difflist(T).
difflist([]).

notmem(X, [Y | Z]) :-
        X < Y,
        notmem(X, Z).
notmem(X, [Y | Z]) :-
        X > Y,
        notmem(X, Z).
notmem(X, []).

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

The search space is pruned even more if linear inequalities are decided rather than delayed, but in this particular program this results in no further speed improvement because of overheads.

3.4.2 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 threre passes that would be needed using a conventional programming language.

```
%
% Network is an input project network of the form
%       [ [node1 , node2, time ] .... ]
%       Graph is the critical path graph produced
%       Latest is the latest possible completion time is specified
% cpm/3 is used if the latest time is specified
% otherwise use cpm/2
%

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 an adjacency graph out of the network
build([],Graph) :-
        buildv([],_,Graph).
build([[I,J,C]|T],Graph) :-
        buildv(ed(I,J,C),to,Graph),
        buildv(ed(I,J,C),from,Graph),
        build(T,Graph).

buildv([],_,[]) :- !.
buildv([],_,[ad(_,_,_,To,From)|T]) :-
        !,addedg([],_,To),
        addedg([],_,From),
        buildv([],_,T).
buildv(ed(I,J,C),to,[ad(I,Es,Lc,To,From)|T]) :-
        !,addedg(J,C,To).
```

```
buildv(Edge,to,[H|T]) :-
        !,buildv(Edge,to,T).
buildv(ed(I,J,C),from,[ad(J,Es,Lc,To,From)|T]) :-
        !,addedg(I,C,From).
buildv(Edge,from,[H|T]) :-
        !,buildv(Edge,from,T).

addedg([],_,[]) :- !.
addedg([],_,[H|T]) :- !,
        addedg([],_,T).
addedg(V,C,[ed(V,C,_,_,_,_)|T]) :- !.
addedg(V,C,[H|T]) :-
        addedg(V,C,T).

% 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),
        compute(Ls,Ec,Tf,Ff,Esj,Lcj,Esi,Lci,C),
        analyse_times(T,Esi,Lci,G).

% Indirect way of doing the calculation just to speed things up
% can be removed and inserted directly in analyse_times
compute(Ls,Ec,Tf,Ff,Esj,Lcj,Esi,Lci,C) :-
        X = Esi+C,
        Ls = Lcj-C,
        Ec = Esi+C,
        Tf = Lcj-X,
        Ff = Esj-X.
```

```
% display routines
print_analysis(G) :-
        printf("\t\tNode\tEs\tLc\n",[]),
        printf("Node1\tNode2\tT\tLs\tEc\tTf\tFf\n",[]),
        print_analysis1(G).
print_analysis1([]).
print_analysis1([H|T]) :-
        print_node(H),
        print_analysis1(T).
print_node(ad(I,Es,Lc,[],From)) :-
        !,
        printf("--------------------------------------------------\n",[]),
        printf("END NODE\t%\t%\t%\n",[I,Es,Lc]).
print_node(ad(I,Es,Lc,To,[])) :-
        !,
        printf("--------------------------------------------------\n",[]),
        printf("START NODE\t%\t%\t%\n",[I,Es,Lc]),
        printf("--------------------------------------------------\n",[]),
        print_times(To,I).
print_node(ad(I,Es,Lc,To,From)) :-
        printf("--------------------------------------------------\n",[]),
        printf("\t\t%\t%\t%\n",[I,Es,Lc]),
        printf("--------------------------------------------------\n",[]),
        print_times(To,I).

print_times([],_).
print_times([ed(V,C,Ls,Ec,Tf,Ff)|T],I) :-
        printf("%\t%\t%\t%\t%\t%\t%",[I,V,C,Ls,Ec,Tf,Ff]),
        is_critical(Tf),
        print_times(T,I).

is_critical(0) :-
        printf(" *\n",[]).
is_critical(Tf) :-
        Tf > 0,
        printf("\n",[]).

% Explanation of output
/*
                Node    Es      Lc (
Gives the Earliest Start and Latest Completion timed for the node)
Node1   Node2   T       Ls      Ec      Tf      Ff
```

```
(Details the times relating to the activity between Node1 & Node2
        T is the time required for the activity
        Ls the Latest Start time
        Ec the Earliest Completion time
        Tf the Total Float
        Ff the Free Float)
 Activities on the critical path are marked with an asterisk
 The start node and end node are computed automatically and distinguished
*/
```

Now the goal

```
?-      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).
```

results in the output

```
                Node    Es      Lc
  Node1   Node2 T       Ls      Ec      Tf      Ff
  -------------------------------------------------
  START NODE    n1      0       0
  -------------------------------------------------
  n1      n2    4       0       4       0       0 *
  n1      n3    3       4       3       4       2
  n1      n4    4       6       4       6       0
  -------------------------------------------------
                n2      4       4
  -------------------------------------------------
  n2      n5    7       4       11      0       0 *
  n2      n3    1       6       5       2       0
  n2      n7    8       8       12      4       4
  -------------------------------------------------
                n3      5       7
  -------------------------------------------------
  n3      n5    4       7       9       2       2
  -------------------------------------------------
                n4      4       10
  -------------------------------------------------
  n4      n6    2       10      6       6       6
  -------------------------------------------------
                n5      11      11
```

```
--------------------------------------------------
n5      n6      1       11      12      0       0 *
n5      n7      3       13      14      2       2
--------------------------------------------------
END NODE        n7      16      16
--------------------------------------------------
                n6      12      12
--------------------------------------------------
n6      n7      4       12      16      0       0 *
```

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. No distinction is made between compiled and interpreted code. All goals are compiled (quickly) before being executed, and any consulted file is immediately compiled. Normally the rulebase is available for inspection 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.

Note: Compilation into CLAM files has not yet been implemented.

4.1 Command Line Arguments

In addition to the name of a file to be consulted on startup, the following command line options are 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 <real>`
Set internal notion of zero to this.

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

4.2 Queries

After the system has been initialized, it will prompt the user for a query. This again is similar to the style of most PROLOG systems. Once a query has been solved, the constraints on variables in the query are printed and the user is prompted to either press carriage return or enter "." (or "n") to accept the answers, or ";" (or "y") to cause backtracking. Each successive set of answer constraints is printed in this way. If there is no chance of backtracking (ie: no choice points), the user is simply informed whether the goal succeeded. Note that if delayed constraints remain at the end of the execution, the answer `Maybe` is returned. A buffer of the last 50 goals is kept, and may be examined by the `h` predicate. An old query may be re-executed by just entering its history number as a goal (eg: `?- 5.`).

The concept of answer constraints for a query, rather than simply an answer substitution, needs to be discussed in more detail. In PROLOG, all the substitutions obtained throughout the execution of a query are "projected" onto the goal variables, and the result is printed. If all variables in a CLP($\mathcal{R}$) goal have either ground values or strictly syntactic bindings, the result is the same. However, when some goal variables are involved in arithmetic constraints such that the variables do not become ground, the situation is a generalisation of the answer substitution case. Consider the example:

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

?- p(V).
```

V is not grounded, but involved in two arithmetic constraints. However, there is only one answer constraint:

```
        0 > V.
```

This is because the constraint relating `X` to `Y` in the rule is not of interest, as `Y` does not relate `X` back to a goal variable in any other way. As another example, consider the two rules

```
p(X,Y):-
        X + Y = 3.

q(X,Y):-
        X + Y + W = 3.
```

and the two goals

```
?- p(A,B).

?- q(A,B).
```

The first goal gives the answer constraint

```
        A = 3 - B
```

while there is no answer constraint for the second. This is easily understood if we realise that the first rule defines a correspondence between its two arguments, while the second one does not. In particular, the second rule could be called with any two arithmetic values as arguments and would succeed. Consequently, it is only the first rule that imposes a "constraining" relationship among the two variables.

The main reason why it is important to understand this behaviour is that a `dump` predicate, as described below, is provided. This allows the programmer to request a set of "answer" constraints at any point of the computation, specifying with respect to which variables the constraints collected so far are to be projected. Consider two rules which show the importance of the choice of variables in the projection:

```
        g:- X + Y + Z = 3, dump([X, Y, Z]).

        g:- X + Y + Z = 3, dump([X, Z]).
```

As expected, the first rule prints the answer constraint

```
        X = 3 - Y - Z,
```

but the second produces no answer constraint at all: again there is no "constraining" relationship between the variables `X` and `Z`, or for that matter, any other pair of variables in the rule.

4.3 Sample Session

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

```
% clpr                          % Sometimes need switches and arguments.

CLP(R) Version 1.1
(c) Copyright International Business Machines Corporation 1989, 1991
All Rights Reserved

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

Y = B
X = g(A)

*** Yes

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

X = 3
Y = -1
Z = 2

*** Yes

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

X = 0.666667
Y = -0.5
Z > 0.166667

*** Yes

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

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

*** 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.

X = 3
Y = -1
Z = 2

*** 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

F = 8
N = 5

*** Retry?

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

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

*** Yes

12 ?- ls.               % look at the entire rulebase

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'].

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

*** Yes

14 ?- ls.

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 = 0.24

P = 0.000773768*B + 99.9226*M

*** Retry?

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

17 ?- p(hello).
hello

*** Yes
```

4.4 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. If a relation is defined statically in more than one chunk, the second definition hides the first for all future bindings. However, if the earlier definition has been protected (using the `prot/2` predicate) 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 to be compiled and the `.clam` file is to be consulted. In the latter case, 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. 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.

4.5 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.

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

`nospy`
Makes all relations invisible to the debugger.

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

`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.

4.6 Arithmetic Precision

Because this implementation of CLP($\mathcal{R}$) makes use of double precision floating point arithmetic, some problems may be caused by artefacts 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 compartsisons, using the `-z` command line option.

4.7 Notes on Efficiency

Chapter 5

Built-In Facilities

5.1 System Predicates

5.1.1 Rulebase

`op(+P,+F,+X)`
: Operator declaration: Precedence `P` (1..1200), fixity `F` for symbol `X`.

`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.

`clisting +P`
`cls +P`
: List the CLAM code for the predicate P.

`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 above. 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. Note that `[-F]`, which is a common synonym for `reconsult` in PROLOG systems, cannot be used.

`retractall`
Delete entire unprotected portion of the rulebase.

`retractall(+H)`
Delete all currently visible rules with heads matching `H`.

`asserta(+R)`
Add rule `R` to the rulebase before all others defining the same predicate.

`assertz(+R)`
`assert(+R)`
Add rule `R` to the rulebase after all others defining the same predicate.

`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.

`quoted_rule(+H,?B)`
This is like `rule/2`, except that it has a "coded" view of the rulebase.

`deny(+H,?B)`
Delete rule matching `H :- B` from the currently visible part of the rulebase. Also tries again on backtracking.

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

`quoted_retract(+R)`
Like `retract/1`, but has a "coded" view of the rulebase.

`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.

`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.

`+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. Note that this form must be used – it is not permissible to simply put a variable in the body of a rule. A cut inside one of these will behave very strangely. In this version, `printf/2` and `dump/1` cannot be used inside `call`.

`not(+X)`

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

`once(+X)`

This is equivalent to `call(X), !` and unfortunately right now it's 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.

`real(?X)`

True if `X` has been bound to a real number.

`atom(?X)`
True if `X` is an atom.

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

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

`?T =.. ?L`
`T` is a term and `L` is the term expanded as a list.

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

`arithmetic(?X)`
True if `X` has a parametric form.

`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

Non-ground variables 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 facilies are as follows.

`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` function in C. The "empty list is needed if no variables are to 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 obtaind 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 Switches

`ctrace`
Start CLAM level tracing of goal execution. This has no effect if the system has not been compiled with the UNIX `-DCTRACE` argument.

`noctrace`
Stop CLAM level tracing of goal execution.

5.1.6 Unix-Related Facilities

`fork`

Split the current process. Fails in one child and succeeds in the other.

`pipe(+X)`

Create a pipe named `X`. For use with `see`, `tell`, etc.

`edit(+F)`

Invoke the default editor on file `F`, and then reconsult the file.

`more(+F)`

Run the file `F` through the UNIX [1] `more` utility.

`halt`

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

`sh`

Invoke an image of `sh` on UNIX systems.

`csh`

Invoke an image of `csh` under UNIX systems.

`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.

5.1.7 Special Facilities

`history`

Print last 20 command line goals.

`history +N`

Print last `N` command line goals.

`N`

Run the command line goal at position `N` in the history list.

`dump(+L)`

List all constraints currently known to the system on the current output stream, projected with respect to the variables in the list `L`.

[1] UNIX is a trademark of Bell Laboratories

`dump(+L1,+L2,-N)`

As above, except that `L1` is the list of variables to project on, `L2` is a list of terms to replace them in the projection (possibly variables) and `V` is a variable to which the coded form of the list of constraints will be bound.

`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 guranateed.

`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.

5.1.8 Missing Predicates

`compile(+F1,+F2)`

Invoke the compiler on file `F1` producing the `clam` file `F2`.

`display(?T)`

Print `T` in prefix format on current output.

`printrule(+R)`

Print the rule `R` in a suitable format on the current output.

`printgoal(+G)`

Print the goal `G` in a suitable format on the current output.

`eof(+X)`

True when `X` is the term signifying the end-of-file condition. Actually, true when `X = ?-(end)` but the predicate should be used for portability.

`libdir ?D`

Binds `D` to the name of the directory that library files live in.

`lib +F`

Consults the file `F` from the library directory.

`true`

Always succeeds.

`abort`
: Stop executing the current query and return to the command line.

`int(+X)`
: True if `X` is an integer.

`string(+X)`
: True if X is a string. Not appropriate now that strings don't exist.

`funstr(?F,?S)`
: Originally converted between a constant `F` and a string `S`.

`shell(+X)`
: Invoke the default shell with the string `X` as input.

`clpr`
: Check if we're in heaven.

5.2 Interpreted Functors

`pow(X, Y)`
: Delays until (a) Y is ground, or (b) X is zero, or (c) X is one.

`abs(X)`
: Delays until X is ground (or fails if the term this is equated to is negative).

`min(X, Y)`
: Delays until X and Y is ground, then returns the obvious answer. (A proper implementation, delaying until $X \leq Y$ or $X \geq Y$ is known, will come later.)

`max(X, Y)`
: Ditto.

`sin(X)`
`cos(X)`
`tan(X)`
: All of these delay until either the argument or result is ground. If only the result is ground, principle values are computed.

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

Differences from the Monash Interpreter

Here we only list those facilities from the Monash interpreter that are no longer supported.

- There are no pre-defined constants.
- The issue of string handling has not yet been settled, so they are not available except as the format (1st) argument to `printf/2`.
- Predicate definitions can no longer be indiscriminantly spread over a number of files.
- There is no more automatic variable generation for answer projection, and hence no `dump/0`.
- Goals result in a prompt for alternate solutions whenever there is a choice point, regardless of whether there are variables in the goal.
- Profiling, and hence the predicates `prof/0` and `noprof/0` are no longer available.
- There is no user control over system warnings, and hence the predicates `warn/0` and `nowarn/0` are no longer available.
- Linear inequalities are always decided immediately rather than delayed, so the predicates `ineq/0` and `noineq/0` are no longer available.
- Statistics are now available through special system predicates, so the `stats/0` system predicate, while may exist, is not supported.