An Abstract Machine for CLP($\mathcal{R}$)

JOXAN JAFFAR*
PETER J. STUCKEY‡
SPIRO MICHAYLOV†
ROLAND H.C. YAP*§

Abstract

An abstract machine is described for the CLP($\mathcal{R}$) programming language. It is intended as a first step towards enabling CLP($\mathcal{R}$) programs to be executed with efficiency approaching that of conventional languages. The core Constraint Logic Arithmetic Machine (CLAM) extends the Warren Abstract Machine (WAM) for compiling Prolog with facilities for handling rea' arithmetic constraints. The full CLAM includes facilities for taking advantage of information obtained from global program analysis.

1 Introduction

The CLP($\mathcal{R}$) language is an instance of the Constraint Logic Programming scheme (CLP) [3], a class of rule based constraint languages. CLP($\mathcal{R}$) embodies constraints over the domain of uninterpreted functors over real arithmetic terms. Appendix A contains an introduction to CLP($\mathcal{R}$). In this paper we present the design of the Constraint Logic Arithmetic Machine (CLAM) for the efficient execution of CLP($\mathcal{R}$) programs.

*IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA.
†School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA.
‡Dept. of Computer Science, Univ. of Melbourne, Parkville, Victoria 3052, Australia.
§On leave from Dept. of Computer Science, Monash University, Clayton, Victoria 3168, Australia.

ACM SIGPLAN '92 PLDI-6/92/CA
© 1992 ACM 0-89791-476-7/92/0006/0128...$1.50

Abstract machines have been used for implementing programming languages for many reasons. Portability is one: only an implementation of the abstract machine needs to be made available on each platform. Another is simply convenience: it is easier to write a native code compiler if the task is first reduced to compiling for an abstract machine that is semantically closer to the source language. The best abstract machines sit at just the right point on the spectrum between the conceptual clarity of the high-level source language and the details of the target machine. In doing so they can often be used to express programs in exactly the right form for tackling the efficiency issues of a source language. For example, the Warren Abstract Machine (WAM) [10] revolutionized the execution of PROLOG, since translating programs to the WAM exposed many opportunities for optimization that were not apparent at the source level. An example from further afield is the technique of optimizing functional programs by first converting them to Continuation Passing Style (CPS) [4]. CPS conversion has lead to highly optimizing compilers for Scheme and Standard ML. The benefit from designing an appropriate abstract machine for a given source language can be so great that even executing the abstract instruction code by interpretation can lead to surprisingly efficient implementations of a language. For example, many commercial PROLOG systems compile to WAM-like code. Certainly more efficiency can be obtained from native code compilation, but the step that made PROLOG usable was that of compiling to the WAM.

While the WAM made PROLOG practical, global analysis shows the potential of making another major leap. For example, Taylor [8] and Van Roy [9] used fairly efficient analyzers to generate high quality native code. Based on certain examples, they showed that the code quality was comparable to that obtained from a C compiler. In the case of CLP($\mathcal{R}$), the opportunities for obtaining valuable information from global analysis [7] are even greater than in PROLOG [1].

In what follows we describe the core CLAM, which includes a comprehensive set of basic instructions. The full CLAM is described next, the focus being on special instructions to take advantage of information obtained from analysis. The subsequent discussion of empirics compares the efficiency of interpretive code, core CLAM code, and optimized CLAM code.

2 CLP($\mathcal{R}$) Implementation Issues

The central implementation issue in CLP($\mathcal{R}$) is based on an observation about programming methodology: while the expressive power of the language comes from its considerable generality, much of the execution of typical programs involves solving simple constraints. These are often just PROLOG-like unifications, arithmetic tests, simple evaluations and assignments. This suggests that the generality of the language should not be allowed to compromise the efficiency of the more basic features. In particular, PROLOG programs should run as efficiently as in a PROLOG system. Furthermore, it is desired that simple arithmetic should be performed efficiently — without resorting to a general constraint solving mechanism. Another major implementation issue is incrementality. Throughout a computation, constraints are being added to or removed from a typically large set of collected constraints. For efficiency reasons this process should be performed without constantly having to resolve all of the collected constraints.

The original CLP($\mathcal{R}$) interpreter [5] satisfied these criteria in the sense that it executed PROLOG programs at least as efficiently as other PROLOG interpreters, and handled simple constraints directly without using the constraint solver. The interpreter was divided into a PROLOG-like inference engine, an interface, and a constraint solver. The inference engine, which was based on structure sharing, was responsible for search, control and unification. The job of the interface was to directly execute arithmetic tests, assignments and simple evaluations where possible. Otherwise it broke up complex constraints into a simpler form to be passed to the constraint solver. The constraint solver itself was partitioned into an equation solver based on Gaussian elimination, an inequality solver which used a highly modified incremental simplex algorithm, and a nonlinear handler which managed the delay and wakeup of nonlinear constraints.

Clearly the desirable features of the interpreter should be retained in a compiler-based system. Thus the objectives of the core CLAM are as follows.

- It should be based on the WAM, because of the latter's suitability for PROLOG.
- It should provide a set of instructions directly "interfacing" to the constraint solver. These being used to replace the functions of the interface in the interpreter.
- The granularity of instructions should be "right" in the sense that constraints can be broken up in a way corresponding to the structure of the constraint solver.

Another objective of the full CLAM is to effectively make use of the kind of information that can be obtained from a global analyzer.

- A special datatype of mutable arithmetic variables should be available. These assume fixed but changeable values, as opposed to normal (logical) variables.
- Simple evaluation of expressions based on mutable variables, and assignment to such variables, should be supported. Mixed constraints, obtaining some coefficients and constants from the mutable variables, should be supported.
- Optimizations based on variables that will no longer be used, and constraints that will become redundant, should be expressible.

As is typical in PROLOG, we assume that optimizing compilation is performed on a program in the context of a *calling pattern* which defines the set of allowed queries. Thus, for one program, different code may be obtained for each different pattern. While the specific nature of such patterns need not concern us here, they typically describe which arguments of a predicate are ground[1], and what their types are.

Next we describe the CLAM, and as we shall see, the extension from the WAM to the CLAM has two essential aspects. One is the *generalization* to arithmetic constraints, and the other is *specialization* for low-level optimizations.

3 Core CLAM

Presented here is a set of basic instructions which are sufficient to execute CLP($\mathcal{R}$) programs in general. In order to understand the design of the instructions, it

[1] A variable is ground if it has one fixed value.

is necessary to discuss some key aspects of the constraint solver. Mainly, we will describe basic instructions which organize constraints for the solver. Some specialized versions of these instructions are also described. We conclude with a short discussion of runtime structures.

3.1 Constraints

A *linear parametric form* is $c_0+c_1V_1+\cdots+c_nV_n$, where $n \geq 1$, each c_i is a real number and each V_i is a distinct *solver variable*. A linear equation, say $V = c_0+c_1W_1+\cdots+c_nW_n$ equates a solver variable and a parametric form. The variables W_i are known as *parameters* and V is a non-parameter. Linear inequalities are stored in the form $lpf \geq 0$ or $lpf > 0$ where lpf is a linear parametric form. The solver ensures that every solver variable appearing in an equation either is a parameter, or appears as a non-parameter in at most one equation. It also ensures that every variable in the inequalities is parametric if it appears in an equation.

The basic instructions of the core CLAM build and manipulate parametric forms:

- `initpf`[2] `c0`
  initializes a parametric form with constant `c0`.
- `addpf_va{lr}`[3] `ci, Vi`
  adds the term `ci * Vi` to the parametric form[4]. The two versions of the instruction (`var` or `val`) correspond to whether V_i is a local variable appearing for the first time (a *new variable*). If so, we need to create storage for the variable, and we may also simplify constraint solving. The variable V_i may be a parameter, in which case c_i is added to the coefficient of V_i in the parametric form, or already have a parametric representation lpf, in which case $c_i * lpf$ is added.
- `solve_eq0`
  signifies the end of the construction of the linear form. The equation $c_0 + c_1V_1 + \cdots + c_nV_n = 0$ is solved by the equation solver.
- `solve_ge0` and `solve_gt0`
  Similar to the above, but forming an inequality or strict inequality instead of an equation.

For example, consider the constraint $5 + X - Y = 0$, where X is a new variable. It could be compiled as shown in Figure 1. We also indicate how the instructions are executed in the case where Y is non-parametric, say $Y = Z + 3.14$ is in the solver.

Recall that a constructed parametric form contains only parametric and new variables. The process of solving an equation built using such a parametric form roughly amounts to (a) finding a parameter V in the form to become non-parametric, (b) writing the equation into the normalized form $V = lpf$, (c) substituting out V using lpf in all other constraints, and finally (d) adding the new constraint $V = lpf$ to the solver.

Suppose there is a new variable in the equation to be compiled. Then it will always appear in the parametric form at runtime hence it can be chosen in step (a). Since this choice is made at compile time, much of the work of step (b) can also be compiled away. Step (c) is not needed since the variable is new. Hence all we need is a new instruction for step (d), one for which the solver always returns *true*. A similar simplification can be made for the compilation of inequalities which contain a new variable. The new instructions:

- `solve_no_fail_eq V`
- `solve_no_fail_ge V`
- `solve_no_fail_gt V`

For example, the above constraint $5+X-Y = 0$, where X is new, can be better compiled into the instructions in Figure 2.

Finally, there are a number of simple enhancements we can make to the instruction set to cater for commonly occurring cases. The cases where the constant is zero or the coefficient is 1 or -1 occur in the majority of instances, so special instructions can be expected both to keep down the code size and cut down on decode time. This can be done using

- `initpf_0`
  start a new linear form with constant 0.
- `addpf_va{lr}_{+-} Vi`
  add a term consisting of a variable with coefficient ± 1.

Nonlinear constraints are separated out at compile time so that they appear in one of a few particular forms: $X = Y \times Z$, $X = pow(Y, Z)$, $X = abs(Y)$, $X = sin(Y)$, $X = cos(Y)$. CLP($\mathcal{R}$) delays the satisfiability of nonlinear constraints until they become linear. To detect when a nonlinear constraint becomes linear, the nonlinear constraint handler associates with each constraint a number of *wakeup degrees* representing the information currently known about the variables appearing in the constraint. As variables become ground, a partic-

[2] The "pf" stands for parametric form.
[3] This brace notation indicates `addpf_val` or `addpf_var`
[4] V_i is either a register or a local variable in the stack; the distinction is not important for this paper.

`initpf`	`5`	*lpf*: 5
`addpf_var`	`1, X`	*lpf*: $5 + 1 * X$
`addpf_val`	`-1, Y`	*lpf*: $1.86 + 1 * X - 1 * Z$
`solve_eq0`		*solve*: $1.86 + 1 * X - 1 * Z = 0$

Figure 1: Core CLAM code for the constraint $5 + X - Y = 0$

`initpf`	`-5`	*lpf*: -5
`addpf_val`	`1, Y`	*lpf*: $-1.86 + 1 * Z$
`solve_no_fail_eq`	`X`	*add*: $X = -1.86 + 1 * Z$

Figure 2: Optimized CLAM code for the constraint $5 + X - Y = 0$

ular instance of a nonlinear constraint passes from one degree to another until it reaches a point where it can be awoken. For a full description see [6].

Instructions are provided for creating a nonlinear constraint in any one of its degrees, so for example there are five instructions for *pow* corresponding to the various wakeup degrees:

- `pow_vvv Vi, Vj, Vk` for $V_i = pow(V_j, V_k)$ with variables V_i, V_j, V_k
- `pow_cvv Vj, Vk` for $c = pow(V_j, V_k)$
- `pow_vcv Vi, c, Vk` for $V_i = pow(c, V_k)$
- `pow_vvc Vi, Vj, c` for $V_i = pow(V_j, c)$
- `pow_cvc c0, Vj, c2` for $c_0 = pow(V_j, c_2)$

For example, $X = pow(3, Y)$ is compiled into the instruction `pow_vcv X, 3, Y`. The remaining forms of a *pow* constraint, e.g. $8 = pow(2, X)$, are equivalent to linear equations and hence the exponentiation is evaluated at compile time, and replaced with a linear equation, e.g. $X = 3$. There is also a class of variants of these instructions, such as `pow_Vcv`, which allow a variable (the one indicated by `V`) to be initialized. The other nonlinears are handled similarly.

3.2 Data Structures

Some of the data structures needed to support the CLAM are a routine extension of those for the WAM — the usual register, stack, heap and trail organization. The main new structures pertain to the solver. Their description is beyond the scope of this paper; see [5]. All that is needed here is that variables involved in arithmetic constraints have a *solver identifier*, which is used to refer to that variable's location in the solver data structures.

The routine modifications to the basic WAM architecture are:

- *Solver identifiers*
  Arithmetic variables need to be represented differently from PROLOG variables. In addition to the usual WAM cell data types, one more is required. Cells of this type contain a solver identifier. Note that the basic unification algorithm needs to be augmented to deal with this new type.

- *Tagged trail*
  In the WAM, the trail merely consists of a stack of addresses to be reset on backtracking. For CLP($\mathcal{R}$), the trail is also used to store changes to constraints. Hence a tagged value trail is required.

- *Choice points*
  Choice points are expanded slightly so as to save the high water mark for solver identifiers and inequalities.

- *Linear form accumulator*
  A linear constraint is built up using one instruction for the constant term, and one for each linear component. During this process, the partially constructed constraint is represented in an accumulator. One of the `solve` instructions then passes the constraint to the solver.

4 CLAM

Above we described the core CLAM and associated runtime structures. Here we present specialized as well as new instructions which implement some key optimization steps in arithmetic constraint solving. Specific kinds of global analysis are required in order to utilize these instructions. A compiler for the core CLAM, in contrast, requires no global analysis.

4.1 Modes

One of the principle mechanisms for enhancing efficiency is to avoid invoking the fully general solver in cases where the constraints are simple. For example when they are equivalent to tests or assignments. Given some fixed calling pattern for a predicate, including information about which variables are ground or unconstrained at call time, we can determine which constraints can be executed as tests or assignments.

Consider the following simple program, whose structure is typical of recursive definitions in CLP($\mathcal{R}$):

```
sum(0, 0).
sum(N, X) :-
      N >= 1, N' = N - 1,
      X' = X - N, sum(N', X').
```

and consider executing the query `sum(7, X)`. The first constraint encountered, $N \geq 1$, can be implemented simply as a test since the value of N is known. The second constraint, $N' = N - 1$, can be implemented simply as an evaluation and assignment to N' because N' is a new variable and the value of N is known. These observations continue to hold when executing the resulting subgoal `sum(6, X')` and for each subsequent subgoal. Hence for any query `sum(c, X)`, where c is a number, the constraint $N \geq 1$ is *always* a test, and $N' = N - 1$ is always an evaluation/assignment.

To take advantage of such constraints, a new data type, *fp_val*, is provided to represent arithmetic variables whose value is known to be ground. (In CLP($\mathcal{R}$) fp_vals are in fact stored in adjacent pairs of registers, or in adjacent stack locations.) The new instructions using fp_vals are given in Figure 3.

For example, the constraints $N \geq 1, N' = N - 1$ can be compiled into

```
              subcf     N, 1, Tmp
              jgef      Tmp, cont
              fail
cont:         mvf       Tmp, N'
```

Even when compilation into the above special instructions is not possible, some simplifications can be made. For example, consider the constraint $V = I * R + V_0$ where the values of I and V_0 will be known when the constraint is encountered. Since I will be known, it is clear at compile time that the constraint will be linear and hence there is no need to separate out the multiplication. Furthermore, the constraint becomes a linear equation on two variables since V_0 is also known.

We add new instructions for creating parametric forms using fp_val variables:

- `initpf_fp FPi`
  begin a parametric form obtaining constant from an fp_val.
- `addpf_fp_va{rl} FPi, Vi`
  add a linear component obtaining the coefficient from an fp_val.

The constraint $V = I * R + V_0$ can be compiled to

```
initpf_fp       V0
addpf_fp_val    I, R
addpf_val_-     V
solve_eq0
```

We finally remark that the instructions for nonlinears are also augmented to make use of variables stored in fp_vals.

4.2 Redundancy

An obvious way to enhance efficiency is to minimize the number of constraints in the solver. Toward this aim we want to eliminate redundant constraints, and an obvious starting point is to eliminate newly encountered constraints which are redundant. Unfortunately such constraints are rare. What is more common however is that new constraints make some *earlier* collected constraints redundant. We concentrate on linear constraints in this subsection since detecting redundancy involving nonlinear constraints is intractable.

We now discuss two forms of redundancy: one associated with variables and the other with constraints.

Redundancy of Variables

We say that a variable is redundant at a certain point in the computation if it will never appear again. The elimination of these variables from the solver produces a more compact representation of the constraints on the variables of interest. Eliminating these variables not only saves space, but also reduces solver activity associated with these variables when adding new constraints. Every variable becomes redundant eventually; thus to gain the most compact representation we may project the constraints onto the remaining variables. (For example, if the only variables of interest are S and T, the constraints $T = T_1, T_1 = T_2 + T_3 - T_4, T_4 - T_3 = 1, T_2 = S + 2$ can be better represented by $T = S + 1$.) However, projection can be expensive, especially when inequalities are involved. It is only worthwhile in certain cases, one of which is identified below. We note that identifying redundant variables could be achieved at runtime

• litf	c, FPi	load a numeric constant into an fp_val;
• getf	Vi, FPj	convert a solver variable to an fp_val;
• putf	FPi, Vj	convert an fp_val to a solver variable;
• stf	FPi, S	put an fp_val on the stack frame (offset S);
• ldf	S, FPi	read an fp_val from stack frame (offset S);
• mvf	FPi, FPj	copy one fp_val to another;
• addf	FPi, FPj, FPk	add fp_vals; similarly for mulf, subf, divf;
• addcf	FPi, c, FPk	add a constant to an fp_val; similarly for mulcf, subcf, divcf;
• jeqf	FPi, L	jump to label L if FPi is zero;
• jgtf	FPi, L	jump to label L if FPi is positive.
• jgef	FPi, L	jump to label L if FPi is nonnegative.

Figure 3: Full CLAM instructions using fp_vals

as part of general garbage collection. However, greater benefit can be obtained by utilizing CLAM instructions to remove these variables in a timely fashion.

Consider the sum() program above. After compiling away the simple constraints as described above, the following sequence of constraints, among others, arise from executing the goal sum(7, X) :

$$\begin{array}{lrcl} (1) & X' & = & X-7 \\ (2) & X'' & = & (X-7)-6 \\ (3) & X''' & = & ((X-7)-6)-5 \\ \ldots & & \ldots & \end{array}$$

Upon encountering the second equation $X'' = X' - 6$ and simplifying into (2), note that the variable X' will never occur in future. Hence equation (1) can be deleted. Similarly upon encountering the third equation $X''' = X'' - 5$ and simplifying into (3), the variable X'' is redundant and so (2) can be removed. In short, only one equation involving X need be stored at any point in the computation. We hence add the following instructions to the CLAM:

• addpf_va{lr}e	ci, Vi
• addpf_va{lr}e_{+-}	Vi
• addpf_fp_va{lr}e	FPi, Vi

As before, these augment the current parametric form with an entry involving V_i, but now they indicate that the variable V_i is redundant and can be eliminated (hence the "e"). Returning to the example above, a possible sequence of

(1)	init_pf	-7
	addpf_val_+	X
	solve_no_fail_eq0	X'
(2)	init_pf	-6
	addpf_vale_+	X'
	solve_no_fail_eq0	X''
(3)	init_pf	-5
	addpf_vale_+	X''
	solve_no_fail_eq0	X'''
...	...	

Notice that a different set of instructions is required for the first equation from that required for the remaining equations. Hence the first iteration needs to be unrolled to produce the most efficient code.

We conclude this subsection with a brief discussion about implementation. Eliminating a variable X from the constraint solver is quite simple: if X is a non-parametric variable, then we just remove the linear form associated with X (as in the above example). If, however, X is a parameter, then to eliminate X we must (i) find an equation containing X, (ii) rewrite the equation with X as the subject, (iii) substitute out X everywhere else using this equation, and finally, (iv) remove the equation. Thus when X is a parameter, there is a trade-off between having one less equation and performing the work toward this aim (because this work is essentially equivalent to adding one new equation). For example, removing an equation is not worthwhile if execution immediately backtracks after its removal.

The following illustrates a typical execution sequence. Suppose the solver contained $X = Y + 1, T = U + Y + 1$ and a new constraint $Y + X - T = 0$ is encountered. Suppose further that it is known that Y does not appear henceforth and so can be eliminated. A straightforward implementation would (a) write the new constraint into parametric form $U - Y = 0$, (b) substitute out U everywhere by Y, (c) add the new constraint $U = Y$, and finally (d) using the information that Y is redundant, process the three resulting equations $X = Y + 1, T = 2 * Y + 1, U = Y$ in order to eliminate Y in the manner described above. A much better implementation will (a) write the new constraint into parametric form $U - Y = 0$, and (b) substitute out Y everywhere by U (instead of vice versa).

program	*Quintus 3.0b*	*CLP($\mathcal{R}$) interpreter*	*CLP($\mathcal{R}$) compiler*
nrev	0.66	1.91	0.79
dnf	0.61	1.89	0.78
zebra	1.33	2.58	1.57

Figure 4: *Prolog benchmarks*

Redundancy of Constraints

Of concern here are constraints which become redundant as a result of new constraints. One class of this redundancy that is easy to detect is *future redundancy* defined in [7], where a constraint added now will be made redundant in the future, but it does not effect execution in between these points.

Consider the `sum()` program once again, and executing the goal `sum(N, X)` using the second rule. We obtain the subgoal

```
N >= 1, N' = N - 1, sum(N', X')
```

and note that we have omitted writing the constraint involving X. Continuing the execution we have two choices: choosing the first rule we obtain the new constraint $N' = 0$, and choosing the second rule we obtain the constraint $N' \geq 1$ (among others). In each case the original constraint $N \geq 1$ is made redundant. The main point of this example is that the constraint $N \geq 1$ in the second rule should be implemented simply as a test[5], and not added to the constraint store. We hence add the new instructions

- `solve_no_add_eq0`
- `solve_no_add_ge0`
- `solve_no_add_gt0`

that behave like the `solve` class of instructions, but which do not add the new constraint. In general this task involves significantly less work than the usual constraint satisfiability check and addition since we do not have to detect implicit equalities and may avoid substitutions.

We finally remark that, in our experiments, implementing future redundancy has lead to the most substantial efficiency gains compared to the other optimizations discussed here. The main reason is that inequalities are prone to redundancy, and the cost of maintaining inequalities in general is already relatively high. Equations in contrast are maintained in a form that is essentially free of this kind of redundancy.

[5]In general, these tests are not just simple evaluations.

4.3 Optimizing Compilation

The kinds of program analysis required to utilize the specialized CLAM instructions include those familiar from PROLOG — most prominently, detecting special cases of unification and deterministic predicates. Algorithms for such analysis have become familiar; see [1, 2] for example. The extension to constraints involves no more than a straightforward extension to these algorithms.

Detecting redundant variables and future redundant constraints can in fact be done without dataflow analysis. One simple method involves unfolding the predicate definition (and typically once is enough), and then, in the case of detecting redundant variables, simply inspecting where variables occur last in the unfolded definitions. For detecting a future redundant constraint, the essential step is determining whether the constraints in an unfolded predicate definition imply the constraint being analyzed. Some further discussions appear in [7].

While we do not have a fully engineered analyzer, our experimental analyzer indicates that many CLP($\mathcal{R}$) programs can be analyzed quite effectively.

5 Some Empirics

Throughout this section, all timings (in seconds) were obtained on an IBM RS 6000/530 workstation running AIX Version 3.0. The C compiler used throughout was the standard AIX C compiler with '−O' level optimization. The systems tested are

- The CLP($\mathcal{R}$) interpreter, written entirely in C, whose inference engine uses standard PROLOG structure sharing techniques.
- The CLP($\mathcal{R}$) compiler system, executing core CLAM code. The CLAM code is interpreted, using an emulator written in C.
- An emulator, as above, for the full CLAM, executing handwritten code.
- Quintus PROLOG version 3.0b, a widely-used commercial system.

```
mortgage(P, T, I, R, B) :-
    T > 1,
    T1 = T - 1,
    P >= 0,
    P1 = P * I - R,
    mortgage(P1, T1, I, R, B).
mortgage(P, T, I, R, B) :-
    T = 1,
    B = P * I - R.
```

Figure 5: *Program for reasoning about mortgage repayments*

Q_1 : $mortgage(100000,\ 360,\ 1.01,\ 1025,\ B)$
Q_2 : $mortgage(P,\ 360,\ 1.01,\ 1025,\ 12625.9)$
Q_3 : $R > 0 \wedge B \geq 0 \wedge mortgage(P,\ 360,\ 1.01,\ R,\ B)$
Q_4 : $0 \leq B \wedge B \leq 1030 \wedge mortgage(100000,\ T,\ 1.01,\ 1030,\ B)$

Figure 6: *Four queries for the mortgage program*

```
double mg1(p, t, i, r)
double p, t, i, r;
{
  if (t == 1.0) return (p*i - r);
  else if (t > 1.0 && p >= 0.0)
    return mg1(p*i - r, t - 1.0, i, r);
  else exit(1);
}
```

```
double mg2(t, i, r, b)
double t, i, r, b;
{
double p;
  if (t == 1.0) return ((b + r)/i);
  else if (t > 1.0) {
    p = (mg2(t - 1.0, i, r, b) + r)/i;
    if (p >= 0.0) return p; else exit(1);
  } else exit(1);
}
```

Figure 7: *C functions for Q_1 and Q_2*

query	CLP($\mathcal{R}$) *interpreter*	CLP($\mathcal{R}$) *core CLAM*	CLP($\mathcal{R}$) *full CLAM but no redundancy*	CLP($\mathcal{R}$) *full CLAM*	*C program*
Q_1	0.10	0.05	0.0042	0.0042	0.0010
Q_2	2.08	1.78	0.0054	0.0054	0.0016
Q_3	2.35	1.84	1.05	0.0700	n/a
Q_4	4.20	2.05	1.78	0.0640	n/a

Figure 8: *Timings for Mortgage program*

- Two C programs, for comparison with a CLP($\mathcal{R}$) program compiled into CLAM code.

In Figure 4 we compare CLP($\mathcal{R}$) with Quintus. The programs chosen are a naive reverse benchmark (six times on a 150 element list, built using a normal functor rather than the list constructor), a program for converting boolean formulae into disjunctive normal form, and a program which solves a standard logic puzzle by combinatorial search. The programs in themselves are not interesting, and hence the code is omitted. The important point is that they test major aspects of a PROLOG inference engine. The purpose of this comparison is first to indicate the relative speeds of the inference engines of the two CLP($\mathcal{R}$) systems, and more importantly, to give evidence that the CLAM can be implemented without significantly compromising PROLOG execution speed.

The main part of this section deals with the constraint aspects of the CLAM. We use the program in Figure 5 in four different ways, as shown in Figure 6, so as to utilize different constraint solving instructions[6]. Appendix C contains the core CLAM code for the mortgage program, and also the (full) CLAM code obtained for this program given the calling patterns of the first and third queries. The code for the second (fourth) query is similar to that for the first (third).

Given the calling pattern associated with the first query, the program can be compiled as though it were a simple recursive definition. Similarly for the second query, though a different recursive definition is obtained. Figure 7 contains C functions which implement these two definitions. The third query essentially propagates constraints, and hence it cannot be compiled so simply. Similarly for the fourth query which essentially carries out a search.

In Figure 8, timings are tabulated for the interpreter, core CLAM and full CLAM. To separate the effects of the mode-based optimizations from those of the redundancy-based optimizations, Figure 8 also contains timings for full CLAM code that does not take advantage of redundancy. The first two timing columns illustrate the benefit of compilation over interpretation. For the next two columns, the program is specialized w.r.t. the modes corresponding to the four queries.

While we have not presented results for a comprehensive set of benchmarks, the mortgage program is somewhat typical in structure for recursive definitions. It was chosen because the four different classes of queries require most of the main components of the constraint solver. Similar tests on a number of other programs, not presented here for space reasons, yielded similarly favorable results.

[6] Similar measurements were carried out in [7] for this program and queries. Their focus was evaluating specific analyses; here it is the efficiency of specific CLAM code.

6 Conclusion

We have described an abstract machine for the efficient execution of compiled CLP($\mathcal{R}$) programs. In addition to providing a basis for executing CLP($\mathcal{R}$) programs with an efficiency comparable to that of commercial Prolog compilers, it is suitable for taking advantage of global optimization techniques that are currently on the leading edge of logic programming implementation techniques.

References

[1] S. K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs", *ACM Transactions on Programming Languages and Systems* **11** (3), 1989, pp 418–450.

[2] S. K. Debray and D.S. Warren, "Functional Computations in Logic Programs", *ACM Transactions on Programming Languages and Systems* **11** (3), 1989, pp 451–481.

[3] J. Jaffar and J-L. Lassez, "Constraint Logic Programming", *Proceedings* 14^{th} *ACM Symposium on Principles of Programming Languages*, January 1987, pp 111–119.

[4] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, "ORBIT: an optimizing compiler for Scheme", *SIGPLAN'86 Symposium on Compiler Construction*, June 1986, pp 219–233.

[5] J. Jaffar, S. Michaylov, P.J. Stuckey and R.H.C. Yap", "The CLP($\mathcal{R}$) Language and System", *ACM Transactions on Programming Languages and Systems*, to appear, July 1992.

[6] J. Jaffar, S. Michaylov and R.H.C. Yap, "A Methodology for Managing Hard Constraints in CLP Systems", *Proceedings ACM-SIGPLAN Conference on Programming Language Design and Implementation*, June 1991, pp 306–316.

[7] N. Jørgensen, K. Marriott and S. Michaylov, "Some Global Compile-time Optimizations for CLP($\mathcal{R}$)", *Proceedings 1991 International Logic Programming Symposium*", October 1991, pp 420–434.

[8] A. Taylor, "LIPS on a MIPS: Results from a Prolog Compiler for a RISC", *Proceedings 7th International Conference on Logic Programming*, June 1990, pp 174–185 [Also Ph.D. thesis, CS, Univ. of Sydney, 1991].

[9] P. van Roy and A.M. Despain, "The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler", *Proceedings 1990 North American Conference on Logic Programming*, October 1990, pp 501–515 [Also Ph.D. thesis by Van Roy, CS/EE, UC Berkeley, 1990].

[10] D.H.D. Warren, "An Abstract PROLOG Instruction Set", Technical note 309, AI Center, SRI International, Menlo Park, October 1983.

Appendix

A An Overview of CLP($\mathcal{R}$)

Real constants and real variables are both *arithmetic terms*. If t, t_1 and t_2 are arithmetic terms, then so are (t_1+t_2), (t_1-t_2), (t_1*t_2), (t_1/t_2), $abs(t)$, $sin(t)$, $cos(t)$ and $pow(t_1,t_2)$. Uninterpreted constants and functors are like those in PROLOG. Uninterpreted constants and arithmetic terms are *terms*, and so is any variable. If f is an n-ary uninterpreted functor, $n \geq 0$, and $t_1,...,t_n$ are terms, then $f(t_1,...,t_n)$ is a term. If t_1 and t_2 are arithmetic terms, then $t_1 = t_2$, $t_1 < t_2$ and $t_1 \leq t_2$ are all arithmetic *constraints*. If at least one of t_1 and t_2 is not an arithmetic term, then only the expression $t_1 = t_2$ is a constraint.

An *atom* is of the form $p(t_1,t_2,\ldots,t_n)$ where p is a predicate symbol distinct from $=$, $<$, and $\leq$, and $t_1,...,t_n$ are terms. A CLP($\mathcal{R}$) program is defined to be a finite collection of *rules*: $A_0 :- \alpha_1,\alpha_2,\ldots,\alpha_k$ where each α_i, $0 \leq i \leq k$, is either a constraint or an atom. A CLP($\mathcal{R}$) *goal* is of the form $?- C \mid \alpha_1,\alpha_2,\ldots,\alpha_k$ where C is a conjunction of constraints and each $\alpha_i, 1 \leq i \leq k$, is either a constraint or an atom.

A *derivation step* for the above CLP($\mathcal{R}$) goal selects the first atom or constraints α_1. If α_1 is a constraint c, then if $c \wedge C$ is satisfiable the new goal is $?- C \wedge c \mid \alpha_2,\ldots,\alpha_k$. If the constraints are unsatisfiable then execution "backtracks" to a point where an alternate choice of matching rule is available and proceeds using this rule. If α_1 is an atom $p(s_1,\ldots,s_n)$ and a rule in the program of the form $p(t_1,\ldots,t_n) :- \beta_1,\beta_2,\ldots,\beta_l$. The resulting new goal is of the form
$?- C \mid s_1 = t_1,\ldots,s_n = t_n,\beta_1,\ldots,\beta_l,\alpha_1,\ldots,\alpha_k$.

Ideally the constraint solver would detect whenever the constraints are unsatisfiable. In practice, because of the intractability of detecting unsatisfiability of nonlinear constraints, our implementation of CLP($\mathcal{R}$) only detects the satisfiability/unsatisfiability of all the non-arithmetic and linear arithmetic constraints in the goal, including nonlinear constraints that are effectively linear in the context of the non-arithmetic and linear constraints.

A *derivation sequence* is a possibly infinite sequence of goals, starting with an initial goal, wherein there is a derivation step to each goal from the preceding goal. A derivation sequence is *successful* if it is finite and its last goal contains only constraints. The constraints in the last goal of successful derivation sequences are called *answer constraints*.

Consider the program in Figure 5. and the goal `?- | mg(P, 2, 1.1, MP, B).` Execution proceeds by first matching with the first rule to obtain the new goal

```
?- | 2 > 1, T1 = 2 - 1, P >= 0,
     P1 = P*1.1 - MP,
     mortgage(P1, T1, 1.1, MP, B).
```

After four steps in which all the constraints are satisfiable, we obtain (ignoring trivial constraints)

```
?- P >= 0, P1 = P*1.1 - MP |
     mortgage(P1, 1, 1.1, MP, B).
```

The next derivation step matches the first rule again,

```
?- P >= 0, P1 = P*1.1 - MP | 1 > 1,
     T1' = 1 - 1, P1 >= 0, P1' = P1*1.1 - MP,
     mortgage(P1', T1', 1.1, MP, B).
```

In the next step, the constraint is unsatisfiable (`1 > 1`) and hence execution backtracks and the next matching rule is attempted, resulting in

```
?- P >= 0, P1 = P*1.1 - MP | 1 = 1,
     B = P1*1.1 - MP.
```

After two more steps the constraints have been added to the solver and found to be satisfiable. Now as only satisfiable constraints remain, the derivation is successful. The answer constraint after simplification is

`P = 0.826446*B + 1.73554*MP, P ≥ 0.`

B Summary of main CLAM instructions

`litf`	`c, FPi`	load numeric constant into fp_val
`getf`	`Vi, FPj`	convert solver variable to fp_val
`putf`	`FPi, Vj`	convert fp_val to solver variable
`stf`	`FPi, S`	put fp_val on stack frame (offset S)
`ldf`	`S, FPi`	read fp_val from stack frame (offset S)
`mvf`	`FPi, FPj`	copy one fp_val to another
`addf`	`FPi, FPj, FPk`	add fp_vals; similarly for `mulf`, `subf`, `divf`
`addcf`	`FPi, c, FPk`	add a constant to an fp_val; similarly for `mulcf`, `subcf`, `divcf`, `subfc`, `divfc`
`jeqf`	`FPi, L`	jump to label `L` if `FPi` is zero
`jgtf`	`FPi, L`	jump to label `L` if `FPi` is positive
`jgef`	`FPi, L`	jump to label `L` if `FPi` is nonnegative
`initpf`	`c0`	initialize accumulator lpf with constant `c0`
`initpf_0`		initialize accumulator lpf with constant 0
`initpf_fp`	`FPi`	initialize lpf with constant from fp_val
`addpf_va{lr}`	`ci, Vi`	add `ci * Vi` to lpf in accumulator
`addpf_va{lr}_{+-}`	`Vi`	add linear component, with 1 or -1 coefficient
`addpf_fp_va{rl}`	`FPi, Vi`	add linear component, with coefficient from fp_val
`addpf_va{lr}e`	`ci, Vi`	like `addpf_va{lr}`, eliminating V_i if possible
`addpf_va{lr}e_{+-}`	`Vi`	like `addpf_va{lr}_{+-}` , eliminating V_i if possible
`addpf_fp_va{lr}e`	`FPi, Vi`	like `addpf_fp_va{lr}` , eliminating V_i if possible
`solve_eq0`		invoke equation solver on $lpf = 0$
`solve_ge0`		invoke inequality solver on $lpf \geq 0$
`solve_gt0`		invoke inequality solver on $lpf > 0$
`solve_no_fail_eq`	`V`	simply add $V = lpf$ to solver
`solve_no_fail_ge`	`V`	simply add $V \geq lpf$ to solver
`solve_no_fail_gt`	`V`	simply add $V > lpf$ to solver
`solve_no_add_eq0`		check $lpf = 0$, do not add to solver
`solve_no_add_ge0`		check $lpf \geq 0$, do not add to solver
`solve_no_add_gt0`		check $lpf > 0$, do not add to solver
`pow_vvv`	`Vi, Vj, Vk`	for $V_i = pow(V_j, V_k)$ where V_i, V_j, V_k are variables
`pow_cvv`	`Vj, Vk`	for $c = pow(V_j, V_k)$
`pow_vcv`	`Vi, c, Vk`	for $V_i = pow(c, V_k)$
`pow_vvc`	`Vi, Vj, c`	for $V_i = pow(V_j, c)$
`pow_cvc`	`c0, Vj, c2`	for $c_0 = pow(V_j, c_2)$
		similar groups of instructions for *cos*, *sin*, *abs* and multiplication

C Example CLAM Code

General Program

```
mg    try          mg1, 5
      trust        mg2
mg1   initpf       -1
      addpf_val_+  T
      solve_gt0
      initpf       1
      addpf_val_-  T
      addpf_var_+  Tmp1
      solve_eq0
      initpf_0
      addpf_val_+  P
      solve_ge0
      mult_Vvv     Tmp3, P, I
      initpf_0
      addpf_val_+  R
      addpf_val_-  Tmp3
      addpf_var_+  Tmp2
      solve_eq0
      getvar       P, Tmp2
      getvar       T, Tmp1
      jump         mg
mg2   gettnum      1, T
      mult_Vvv     Tmp1, P, I
      initpf_0
      addpf_val_+  R
      addpf_val_-  Tmp1
      addpf_val_+  B
      solve_eq0
      proceed
```

Specialized for Q_1

```
mg    subcf        T, 1, Tmp
      jgtf         Tmp, mg1
      jeqf         Tmp, mg2
      fail
mg1   mvf          Tmp, T
      jgef         P, cont
      fail
cont  mulf         P, I, Tmp
      subf         Tmp, R, P
      jump         mg
mg2   mulf         P, I, Tmp
      subf         Tmp, R, B
      proceed
```

Specialized for Q_3

```
mg    subcf        T, 1, Tmp
      jgtf         Tmp, mg1
      jeqf         Tmp, mg2
      fail
mg1   subcf        T, 1, T
      jgtf         T, cont1
      fail
cont1 initpf_0
      addpf_val_+  P
      solve_no_add_ge0
      initpf_0
      addpf_fp_val I, P
      addpf_val_-  R
      solve_no_fail_eq Tmp3
      getvar       P, Tmp3
      jump         mg
mg2   subcf        T, 1, Tmp4
      jeqf         Tmp4, cont2
      fail
cont2 subfc        0, I, Tmp2
      initpf_0
      addpf_val_+  R
      addpf_fp_val Tmp2, P
      addpf_val_+  B
      solve_eq0
      proceed
```