Some Global Compile-Time Optimizations for CLP($\mathcal{R}$)

Niels Jørgensen
Roskilde University Center
DK-4000 Roskilde, Denmark
`nielsj@dat.ruc.dk`

Kim Marriott
IBM T. J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598, U.S.A.
`kimbal@watson.ibm.com`

Spiro Michaylov
School of Computer Science, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213, U.S.A.
`spiro@cs.cmu.edu`

Abstract

Generation of efficient code for constraint logic programming (CLP) languages poses new problems. It is not yet clear which techniques from Prolog compilation will be important nor what sort of domain specific techniques can be developed. As a case study, we describe and evaluate four global optimizations for use in the compilation of the language CLP($\mathcal{R}$). The first three are variants of optimizations used in Prolog compilation. The fourth is novel, and is based on finding constraints which are "future redundant" in the sense that at some point in the future they will be implied by other constraints. Finally, we give evidence in the form of preliminary tests that these optimizations can significantly improve performance.

1 Introduction

Constraint logic programming (CLP) languages[7] extend the power of logic programming languages, such as Prolog, by allowing computation over non-symbolic domains such as the real numbers. Compilation of CLP languages poses new problems, requiring techniques used in the compilation of Prolog as well as techniques from operations research for efficient constraint solving. It is yet not clear which optimizations used in Prolog compilation will be important nor what sort of domain specific techniques can be developed.

As a case study we describe and evaluate some optimizations based on global analysis for use in the compilation of the CLP language CLP($\mathcal{R}$). Motivation for this comes from the impressive performance improvements given by similar global techniques in Prolog compilation [14, 16, 17]. We consider four optimizations. In the first, a variant of mode analysis is used to detect those constraints or parts of constraints that are trivial in the sense that simple arithmetic evaluation and assignment will suffice. The second optimization reduces choicepoint creation and backtracking by recognizing when clauses for a predicate are mutually exclusive. The third reduces the number of expression evaluations by moving recursion-invariant expressions out of recursive definitions. The fourth is the most interesting. It is a novel optimization relying on detection of constraints which are "future redundant" in the sense that at some point in the future they will be implied by other constraints. Future redundant constraints need not be added to the current constraint, improving the constraint solver's time and space performance. We describe preliminary tests of these optimizations that show them to be very promising.

To our knowledge, existing compilers for CLP languages use only local optimizations, and do not employ global analysis and optimizations similar to those given here. The techniques for detecting trivial constraints and elimination of unnecessary choicepoints are closely related to those studied for the compilation of Prolog. For example, see Mellish [12], Debray [4, 5], or Hickey and Mudambi [6]. The technique for removal of invariant expressions from recursion is in spirit similar to the "code motion" optimization used in compilation of conventional programming languages [13], and Debray [3] has suggested a related transformation for Prolog programs. However the actual analysis we suggest seems new. The optimization involving future redundant constraints is unlike any other optimization that we are aware of.

In Section 2 we give an introduction to CLP($\mathcal{R}$). The optimizations are described in Sections 3 to 6. Section 7 contains the test results and describes the optimized code, and Section 8 concludes.

2 Introduction to CLP($\mathcal{R}$)

In this section we sketch the language CLP($\mathcal{R}$), its operational semantics, and its implementation. The reader is referred to [7, 8, 9] for more details.

A CLP program consists of a number of *clauses* of the form

H :- π, B.

where H is an *atom*, π is a (possibly empty) conjunction of constraints, and B is a (possibly empty) sequence of atoms. The clause is read as "A is true if the constraints π hold and B is true". CLP($\mathcal{R}$) extends Prolog by adding constraint solving over the real numbers. In CLP($\mathcal{R}$) constraints are equalities and inequalities over arithmetic expressions and equalities over symbolic expressions.

The operational semantics of a CLP program are given by query evaluation. A *query* $\langle G, \Pi \rangle$ consists of two parts, a sequence of atoms G, called a *goal*, and a *constraint* Π, initially the empty or true constraint. A query is repeatedly rewritten as described below until an empty sequence of atoms is reached. The associated constraint is an *answer constraint* to the original query. It is simplified to be in terms of the original variables.

```
mortgage(P,T,I,R,B) :-
T > 1,
T1 = T - 1,
P >= 0,
P1 = P * (1 + I / 1200 ) - R,
mortgage(P1,T1,I,R,B).
mortgage(P,T,I,R,B) :-
T = 1,
B = P * ( 1 + I / 1200 ) - R.
```

Figure 1: Program for reasoning about mortgage repayments

A query $\langle G, \Pi \rangle$ is rewritten as follows. Let A be the first atom in the goal G and G' the remaining sequence. A clause H:-π, B from the program is selected to *resolve* against A and the variables in the clause are renamed apart from those in the query. If more than one clause is a candidate for selection, a *choicepoint* is established. The constraint $\Pi' = (\Pi \wedge (A = H) \wedge \pi)$ is then tested for *satisfiability*. If it is satisfiable then the query is rewritten to $\langle B :: G', \Pi' \rangle$ where :: indicates sequence concatenation. If Π' is unsatisfiable, failure occurs and execution backtracks to the most recent choicepoint, and an alternative clause is chosen to resolve with.

Some further terminology is needed. When the computation has reached the query $\langle G, \Pi \rangle$, the *current constraint* is Π. Given the query $\langle A :: G, \Pi \rangle$ the *calling constraint* associated with A is the projection of Π onto the variables that occur in A.

As an example consider the *mortgage* program in Figure 1. In the arguments to *mortgage*, P is the payment, T is the time in months, I is the annual interest rate which is compounded monthly, R is the monthly repayment, and B is the balance. This program can be queried in many different ways. For example, consider the queries:

$$\begin{array}{lcl} Q_1 & : & \mathit{mortgage}(100000, 360, 12, 1025, B) \\ Q_2 & : & \mathit{mortgage}(P, 360, 12, 1025, 12625.9) \\ Q_3 & : & R > 0 \wedge \mathit{mortgage}(P, 360, 12, R, B) \\ Q_4 & : & 0 \leq B \wedge B \leq 1030 \wedge \mathit{mortgage}(100000, T, 12, 1030, B) \end{array}$$

for which the answer constraints are

$$\begin{array}{lcl} Q_1 & : & B = 12625.9 \\ Q_2 & : & P = 100000 \\ Q_3 & : & R > 0 \wedge P = 0.0278 * B + 97.22 * R \\ Q_4 & : & T = 355 \wedge B = 385.449. \end{array}$$

The query Q_1 could be processed correctly by a Prolog system, while the others require the extra power of CLP($\mathcal{R}$). In particular the answer to Q_3 shows the ability to produce "symbolic output".

The CLP($\mathcal{R}$) interpreter [8, 9] is implemented as an *engine*, *interface*, and *constraint solver*. The engine performs traditional Prolog-like operations such as clause search and backtracking. The interface classifies and solves trivial constraints, while the constraint solver handles arithmetic constraints.

Constraints are sent by the engine to the interface, which performs an initial analysis of the terms in the constraints, so as to determine their type and, hence, how they should be processed. In particular, the interface seeks to avoid invoking the time consuming constraint solver. The essential idea is that some constraints are of a trivial nature. For instance, in case of the constraint $T1 = T - 1$, if $T1$ is free and T is a number, we need only assign the value of $T - 1$ to $T1$, and this should be done without involving the constraint solver.

Consider what happens in the CLP($\mathcal{R}$) interpreter when an equality constraint, $T_1 = T_2$, is first encountered. The interface first classifies each term into one of five categories: *Var*, *Function*, *ArVar*, *Number*, or *ArFn*; and then performs the action specified in Table 1.

Table 1: CLP($\mathcal{R}$) equality solving

T_1 / T_2	Var	Function	ArVar	Number	ArFn
Var	$T_1 \leftrightarrow T_2$	$T_1 \rightarrow T_2$	$T_1 \rightarrow T_2$	$T_1 \rightarrow T_2$	interface
Function		$T_1 = T_2$?	fail	fail	fail
ArVar			solver	solver	solver
Number				$T_1 = T_2$?	interface
ArFn					interface

The category *Var* stands for variables that are uninstantiated and are not constrained by arithmetic constraints. *Function* stands for non-arithmetic terms such as *nil*, and *ArFn* stands for complex arithmetic expressions. *ArVar* represents the variables that are uninstantiated but constrained by arithmetic constraints. We assume that if a variable is constrained by arithmetic constraints and the variable is uniquely determined, then it occurs as a *Number* in the above table. The entry $T_1 \rightarrow T_2$ means that T_1 is bound to T_2. $T_1 \leftrightarrow T_2$ means that T_1 can be bound to T_2 or vice versa. The meaning of the entry *interface* is that further examination of the equality is necessary to determine if a call to the solver can be avoided. For instance, if in *Var* = *ArFn* all variables in the arithmetic expression are bound to numbers, the solver need not be involved. The meaning of $T_1 = T_2$? in the table's entry for *Function* $\times$ *Function* is that if the functors are identical, the arguments are equated recursively.

The *constraint solver* proper processes arithmetic constraints. It uses Gaussian elimination to process equalities and a version of the Simplex algorithm to process inequalities. In particular, the Simplex algorithm is expensive and the cost is proportional to the number of constraints and the number of variables. In some of the optimizations we will discuss, it will be necessary to consider redundancy of constraints in the current constraint. Redundant constraints are constraints that are implied by other constraints in the conjunction. For instance the constraint $T > 1$ is redundant in the conjunction $T > 1 \wedge T1 > 1 \wedge T1 = T - 1$. Gaussian elimination, like unification, has the property that redundant equations are removed automatically as a byproduct of keeping equalities in solved form. However, the Simplex algorithm does not have this property. As runtime elimination of redundant inequalities is expensive it is not employed in the CLP($\mathcal{R}$) system. Finally, the constraint solver cannot solve simultaneous nonlinear constraints. Hence, these are simply assumed to be satisfiable when they are encountered, and their further consideration is delayed until they become linear through the grounding of certain variables.

The CLP($\mathcal{R}$) compiler [10], which presently does not employ any global code optimization techniques, is based on the Constraint Logic Abstract Machine (CLAM), a superset of the Warren Abstract Machine (WAM). For our purposes, the main difference between the compiler and the interpreter described above, is that the compiler "compiles out" much of the interface code. For example, at compile time complex expressions may be broken into more basic ones and set up for the appropriate parts of the constraint solver.

3 Bypassing the constraint solver

In many cases the constraints generated when executing a program are trivial in the sense that the constraint solver is not required to process them. For example, if T is a number in a query to *mortgage* then the second argument will always be a number. This means that the constraint $T1 = T - 1$ will always be a simple "assignment" and so never needs to be passed to the constraint solver. Also there are cases where a constraint will still have to be passed to the constraint solver, but can be simplified at compile time. For example, consider the constraint $P1 = P * (1 + I/1200) - R$ in the *mortgage* program. If it is known at compile time that I will always be a number, this constraint will always be linear, so the overhead of dealing with nonlinear expressions can be avoided at run time. The expression $1 + I/1200$ can simply be evaluated, and used as the coefficient of the variable P in the linear equation.

In CLP($\mathcal{R}$) such trivial constraints do not incur the full cost of the constraint solver since they will be recognized by the interface. However, much of the cost of setting them up for the constraint solving algorithms is still incurred. If it was known at compile time that the constraints were guaranteed to be trivial, the generated code would not need to call the interface, and the code for variable lookup, order of evaluation and assignment or testing could be hardwired in. In this section we sketch how a variant of "mode analysis" [4, 12] used in Prolog compilation can be used to provide such information to the compiler. The reader is referred to [11] for a formal framework in which to express this analysis.

A mode analysis finds "call patterns" to describe the form of runtime calls to predicates. In a simple Prolog mode analysis the call pattern for an n argument predicate p is of the form $p(d_1, \ldots, d_n)$, where $d_i \in \{Gnd, Var, Any\}$. If d_i is *Gnd* this indicates that in all calls to p the ith argument is "ground", that is contains no variables. If d_i is *Var* this indicates that in all calls the ith argument is a variable, and if d_i is *Any* then nothing is known.

For our purposes, the extended set of modes $\{Gnd, Var, Any, Num, Arith\}$ is more useful. The mode *Var* corresponds to the category *Var* used in the initial testing, the mode *Num* to the category *Number*, and a term is *Arith* if it is either a *ArVar* or a *Number*. Of course, if a term is described by em Num, it is also described by *Gnd*.

Consider the queries Q_1 to Q_4 to the *mortgage* program. For each of the four queries it can be inferred by a simple mode analyzer that all calls to the *mortgage* predicate are described by the modes:

$$\begin{array}{lcl} Q_1 & : & mortgage(Num, Num, Num, Num, Var) \\ Q_2 & : & mortgage(Any, Num, Num, Num, Num) \\ Q_3 & : & mortgage(Any, Num, Num, Arith, Any) \\ Q_4 & : & mortgage(Num, Any, Num, Num, Arith). \end{array}$$

Using this information the compiler can generate simplified arithmetic code for the constraints. In the simplified code, many of the arithmetic variables will behave like arithmetic variables in an imperative language, and many constraints will either become simple evaluations, tests and assignments, or be substantially reduced in complexity.

4 Mutual exclusion

The optimization described in this section relies on finding when the clauses defining a predicate are "mutually exclusive" for a particular call pattern which grounds some of the arguments. The engine can use this information to reduce the number of choicepoints for the call, giving both space and time improvements. The optimization is a modification of those suggested for Prolog compilation [5, 6].

As an example, note that in the *mortgage* program, the constraint $T > 1$ in the first clause and the constraint $T = 1$ in the second clause cannot both be satisfiable if T is bound to a specific value. Furthermore, when T is ground, each of these constraints is just a test. Thus, for calling constraints in which T is a number, the *mortgage* predicate can be compiled into the following deterministic pseudo target code.

$$\begin{array}{lcl} mortgage(\ldots) & \text{:-} & \text{if } T > 1 \text{ then } \ldots\ mortgage(\ldots) \\ & & \text{else if } T = 1 \text{ then } \ldots \text{ else } fail. \end{array}$$

We simplify the following exposition by assuming that predicate definitions only contain two clauses with disjoint variables. However it is straightforward to extend the definitions. Consider a call $p(\bar{t})$ and the predicate definition:

$$\begin{array}{l} p(\bar{x}_1) \text{:- } \pi_1,\ B_1. \\ p(\bar{x}_2) \text{:- } \pi_2,\ B_2. \end{array}$$

The clauses are *mutually exclusive* for $p(\bar{t})$ if the constraints $\bar{t} = \bar{x}_1 \wedge \pi_1$ and $\bar{t} = \bar{x}_2 \wedge \pi_2$ are not both satisfiable. A set of clauses are said to be mutually exclusive for a call pattern if they are mutually exclusive for any atom described by the call pattern. For example, *mortgage* is mutually exclusive for the call pattern $mortgage(Any, Gnd, Any, Any, Any)$.

It is simple to test whether clauses are mutually exclusive for a particular call pattern. Let CP be a call pattern for p, and let $\overline{g}_1$ and $\overline{g}_2$ be the arguments of $p(\overline{x}_1)$ and $p(\overline{x}_2)$ that CP specifies to be ground. Then the clauses in the definition of p are mutually exclusive for CP if the constraint $\overline{g}_1 = \overline{g}_2 \wedge \pi_1 \wedge \pi_2$ is unsatisfiable.

Given that the compiler knows that predicate p is mutually exclusive for the call patterns $CP_1, \ldots, CP_n$, it can produce optimized code corresponding to the definition:

$$p(\overline{x}) \text{ :- if } CP_i \text{ describes } \overline{x} \text{ then } p_i(\overline{x}) \text{ else } p_0(\overline{x}).$$

Where $p_0(\overline{x})$ has the original (non-deterministic) definition of $p(\overline{x})$ and each $p_i(\overline{x})$ is defined by the pseudo-code:

$$\begin{array}{lll} p_i(\overline{x}) & \text{:-} & \text{if } test(CP_i, \pi_1) \text{ then } (rem(CP_i, \pi_1), B_1) \\ & & \text{else if } test(CP_i, \pi_2) \text{ then } (rem(CP_i, \pi_2), B_2) \text{ else } fail. \end{array}$$

The test $test(CP, \pi)$ is the projection (or restriction) of constraint π onto the variables constrained to be ground by the call pattern CP and the constraints $rem(CP, \pi)$ are the remaining constraints in π. For example, if π is $T > 1, T1 = T - 1, P >= 0, P1 = P * (1 + I/1200) - R$ and CP constrains T and P to be ground, $test(CP, \pi)$ is $T > 1, P >= 0$ and $rem(CP, \pi)$ is $T1 = T - 1, P1 = P * (1 + I/1200) - R$.

The advantage of this code is that, if p is called with a call described by one of the call patterns, no choicepoints need to be established and no backtracking will occur. The disadvantage is an increase in code size and runtime tests. As a further optimization, global analysis similar to that used in the last section can be used to remove runtime tests for groundness. Such eliminable tests arise in two ways. The first way is because in all calls to the predicate the test will be true. The second way is that earlier tests imply the result of subsequent tests. Thus, for example, if in a call to the *mortgage* program the second argument is ground then in subsequent calls to *mortgage* the second argument will always be ground.

5 Code motion

A standard optimization technique in conventional programming languages [13] is to move loop invariant code from inside a loop to its beginning. This is called *code motion*. A similar optimization may be applied to CLP($\mathcal{R}$) programs by moving recursion-invariant expressions out of recursive clauses. A related transformation was studied by Debray [3] for Prolog programs.

The expression $(1 + I/1200)$ in the *mortgage* program is recursion invariant because I is invariant. Thus the mortgage program can be rewritten to that shown in Figure 2. This program is more efficient because the expression $(1 + I/1200)$ is not repeatedly evaluated.

We now consider how to automate this optimization. We will only consider simple recursion. The idea is to first find which arguments are "invariant" for each atom and then to optimize the expressions containing these variables. Define the

```
mortgage(P,T,I,R,B) :-
mortgage1(P,T,(1 + I / 1200),R,B).

mortgage1(P,T,E,R,B) :-
T > 1, T1 = T - 1, P >= 0, P1 = P * E - R, mortgage1(P1,T1,E,R,B).
mortgage1(P,T,E,R,B) :-
T = 1, B = P * E - R.
```

Figure 2: Result of applying code motion optimization to *mortgage* program

derived constraint of A and clause H :- π, B to be $A = H \wedge \pi$. An argument *Arg* of p is defined to be *invariant* as follows. Consider every two clauses (H :- π, B) and C which define p, and every recursive call, say A, to p in B. Let π_{der} be the derived constraint of A and $\rho(C)$ where ρ is a renaming. Then *Arg* is invariant if the implication $\pi \wedge \pi_{der} \Rightarrow Arg = \rho(Arg)$ *always* holds. That implication can be tested by having CLP($\mathcal{R}$) test the (un)satisfiability of the constraint $\pi \wedge \pi_{der} \wedge Arg \neq \rho(Arg)$.

For the *mortgage* predicate we must consider two pairs of clauses, one which consists of two instances of the recursive clause, and one which is the recursive clause and the nonrecursive clause. For the first pair we compute the derived constraint

$$\pi_{der} = (P1 = P' \wedge T1 = T' \wedge I = I' \wedge R = R' \wedge B = B') \wedge \\ (T' > 1 \wedge T1' = T' - 1 \wedge P' \geq 0 \wedge P1' = P' * (1 + I'/1200) - R').$$

Letting $\pi = T > 1 \wedge T1 = T - 1 \wedge P \geq 0 \wedge P1 = P * (1 + I/1200) - R$ we then have to test that the constraint $\pi \wedge \pi_{der} \wedge I \neq I'$ is unsatisfiable. Similarly for the other pair of clauses. Each test will show that the relevant constraint is unsatisfiable, so the optimization applies.

6 Future redundant constraints

Consider the *mortgage* program. The constraint $T > 1$ is "future redundant". By this we mean that after testing the constraint for satisfiability, adding or not adding the constraint to the current constraint will not affect subsequent operational behavior. Intuitively this is because at the next step in any reduction a stronger constraint will be encountered, either $T1 = T - 1 \wedge T1 > 1$ or $T1 = T - 1 \wedge T1 = 1$, so the constraint $T > 1$ will be redundant in any future constraint. If T is a variable, then not adding the future redundant constraint reduces the size of the current constraint. Initial test results given in the next section suggest that this may give a dramatic speed up. There seems to be no analogue to this optimization for standard Prolog.

Constraint π is *future redundant* in clause H :- $\pi_{other} \wedge \pi, B$ if for every clause C, we have that $\pi_{der} \wedge \pi_{other} \Rightarrow \pi$ where π_{der} is the derived constraint of the first atom in B and some renaming of C.

The optimization associated with a future redundant constraint is to modify the operational semantics so that future redundant constraints are tested for satisfiability, but are not added to the current constraint. More exactly, consider a query $\langle A : G, \Pi\rangle$ which we are rewriting by resolving with the clause H :- $\pi_{other} \wedge \pi, B$ in which π is future redundant. The rewrite step in the operational semantics is modified to:

1. check that $\Pi \wedge A = H \wedge \pi_{other} \wedge \pi$ is satisfiable and if so
2. rewrite the query to $\langle B :: G, \Pi \wedge A = H \wedge \pi_{other}\rangle$.

Correctness of this transformation is captured in the following Proposition. Note that proving preservation of answer substitutions is more subtle than it first seems, as a constraint may be future redundant with respect to another future redundant constraint and so on. The key point is that in a successful derivation every future redundant constraint must be implied by constraints in the unit clauses, and constraints in the unit clause cannot, by definition, be future redundant.

Proposition There is a one-to-one correspondence between (possibly partial) derivations for the original operational semantics and the "future redundant" operational semantics in which derivation length is preserved. Furthermore, in the case of successful derivations, the answer constraints are equivalent, modulo variable renaming. ■

The definition of future redundant constraints suggests a simple analysis to find them. As an example of the analysis consider the *mortgage* program. Assume that we wish to test if $T > 1$ is future redundant in the first clause of mortgage. We first compute the derived constraints of *mortgage*$(P1, T1, I, R, B)$ for each clause in the program. These are

$$\pi_1 = (P1 = P' \wedge T1 = T' \wedge I = I' \wedge R = R' \wedge B = B') \wedge$$
$$(T' > 1 \wedge T1' = T' - 1 \wedge P' \geq 0 \wedge P1' = P' \times (1 + I'/1200) - R')$$

and

$$\pi_2 = (P1 = P' \wedge T1 = T' \wedge I = I' \wedge R = R' \wedge B = B') \wedge$$
$$(T' = 1 \wedge B' = P' \times (1 + I'/1200) - R').$$

We then verify that $\pi_{other} \wedge \pi_1 \Rightarrow T > 1$ and $\pi_{other} \wedge \pi_2 \Rightarrow T > 1$ where

$$\pi_{other} = (T1 = T - 1 \wedge P \geq 0 \wedge P1 = P \times (1 + I/1200) - R).$$

It follows that $T > 1$ is future redundant.

Note that Gaussian elimination deals with redundant equalities directly, so here we are only concerned with redundant inequalities. Implications of the form $\pi \Rightarrow E < E'$ can be tested by simply checking that $\pi \wedge E' \leq E$ is satisfiable.

The analyses for code motion and for future redundancy are local in nature. However, both analyses can be extended to take calling constraints into account. We shall only look at future redundancy, but the case for code motion is analogous.

Constraint π is *future redundant* in clause H :- $\pi_{other} \wedge \pi, B$ in the context of calling constraint π_{call}, if for every clause C, we have that $\pi_{call} \wedge \pi_{der} \wedge \pi_{other} \Rightarrow \pi$ where π_{der} is the derived constraint of the first atom in B and some renaming of C.

Given that a constraint is future redundant in a clause in the context of all of the clause's calling constraints, we can safely apply the future redundancy optimization. To prove future redundancy in the context of all calling constraints, we can use abstract interpretation techniques [11]. These can find descriptions of the inequalities that *always* hold between the variables in the clause head whenever the clause is called.

A simple domain for this purpose is the set of constraints which are conjunctions of strict and non-strict inequalities between variables and the constants 0, 1 and -1. Call this domain *Ineq*. An example element is $x \geq 0 \wedge x > y$. An element of *Ineq* describes any constraint which implies it. Similar description domains, based on linear constraints, have been used for static array bound checking [1], determination of invariant assertions [2], and relationships between argument sizes in definite clause programs [15].

An analysis based on *Ineq* will show that: for queries Q_1 and Q_2 the calling constraints for *mortgage* are described by $T > 0 \wedge I > 1 \wedge R > 1$; for Q_3 they are described by $T > 0 \wedge I > 1 \wedge R > 0$; and for Q_4 they are described by $I > 1 \wedge R > 1 \wedge B \geq 0$. From this, one can show that the constraint $P \geq 0$ in the first clause of *mortgage* is future redundant in the context of the calling patterns generated by all of these queries.

7 Test results

In this section we describe some preliminary test results to evaluate the usefulness of the optimizations, and we describe the optimized code in the form of high-level pseudo code. We again use the *mortgage* program of Figure 1, and the optimized code is shown in Figure 3. This shows code for the four versions of the program specialized for the respective queries Q_1 to Q_4. The variables shown in italics are those that can be treated like variables in an imperative language, and $\leftarrow$ denotes that the right hand side is directly evaluated and assigned to the variable on the left. The symbol `fr_ineq` denotes a future redundant inequality constraint, while `ineq` and `eqn` denote normal constraints. Constraints not enclosed by one of these symbols are simple boolean tests.

The test results were obtained by hand coding the CLAM instructions for a number of optimized variants of the *mortgage* program, and comparing the run time with that of the unoptimized program given in Section 2. To express the optimizations in low level code we have augmented the CLAM instruction set with about a dozen instructions of a “von Neumann” flavor, for simple one-directional arithmetic and control flow, as well as some extra instructions for linear equations and inequalities.

We show the empirical results in the form of the five histograms in Figures 4 and 5. Figure 4 shows the effect of the future redundancy optimizations when applied to the four queries, in before/after pairs. Note that there is no change for Q_1 since the inequalities are ground. This optimization is shown separately because its effect is so much more dramatic than that of the others. Then Figure 5 shows, for each of the queries, one histogram describing the effects of all the other optimizations that apply to the query. These begin with the version that included the future redundancy optimization, and show the *additional* effects of the other optimizations. The bar F represents the basic program with future redundancy optimization only. The other

```
mortgage(P,T,I,R) =
  I' ← 1 + I/1200
  mortgage1(P,T,I',R)

mortgage1(P,T,I,R) =
  if T > 1
    T_1 ← T − 1
    if P ≥ 0
      P_1 ← P ∗ I − R
      return mortgage1(P_1,T_1,I,R)
    else fail
  else if T = 1
    return P ∗ I − R
  else fail

?-P ← 100000
  T ← 360
  I ← 12
  R ← 1025
  B ← mortgage(P,T,I,R)
```

Q_1

```
mortgage(P,T,I,R,B) :-
  I' ← 1 + I/1200
  mortgage1(P,T,I',R,B)

mortgage1(P,T,I,R,B) :-
  if T > 1
    T_1 ← T − 1
    fr_ineq( P >= 0 )
    eqn( P_1 = P * I - R )
    mortgage1(P_1,T_1,I,R,B).
  else if T = 1
    V_1 ← (B + R)/I
    eqn( P = V_1 )
  else fail

?-T ← 360
  I ← 12
  R ← 1025
  B ← 12625.9
  mortgage(P,T,I,R,B)
```

Q_2

```
mortgage(P,T,I,R,B) :-
  I' ← 1 + I/1200
  mortgage1(P,T,I',R,B)

mortgage1(P,T,I,R,B) :-
  if T > 1
    T_1 ← T − 1
    fr_ineq( P >= 0 )
    eqn( P_1 = P * I - R )
    mortgage1(P_1,T_1,I,R,B)
  else if T = 1
    eqn( B = P * I' - R )
  else fail

?-T ← 360
  I ← 12
  mortgage(P,T,I,R,B)
```

Q_3

```
mortgage(P,T,I,R,B) :-
  I' ← 1 + I/1200
  mortgage1(P,T,I',R,B)

mortgage1(P,T,I,R,B) :-
  fr_ineq( T > 1 )
  eqn( T_1 = T - 1 )
  if P ≥ 0
    P_1 ← P ∗ I − R
    mortgage1(P_1,T_1,I,R,B)
  else fail
mortgage1(P,T,I,R,B) :-
  eqn( T = 1 )
  V_1 ← P ∗ I − R
  eqn( B = V_1 )

?-ineq( 0 <= B )
  ineq( B <= 1030 )
  P ← 100000
  I ← 12
  R ← 1030
  mortgage(P,T,I,R,B)
```

Q_4

Figure 3: Optimized programs corresponding to the four queries

Figure 4: Run time improvement from future redundancy optimization

Figure 5: Additional run time improvement from remaining optimizations

bars represents the additional optimizations of code motion (M), mutual exclusion (D), and solver bypass (S), which are first shown separately, and then all together (A). Note however that since the mutual exclusion optimization does not apply to Q_4, the A bar does not include that optimization.

All times are in CPU seconds. All measurements were obtained using a Sun 3/60 workstation with a 68881 floating point accelerator. To help factor out the cost of floating point arithmetic, measurements were taken using both the accelerator and software-emulated floating point. The times using a floating point accelerator are shown as the darkly shaded part of each bar, since they are uniformly shorter as expected.

For comparison, we include the measurement for a *mortgage* program written in C. This measurement is for input data that correspond to the query Q_1, and is shown by the bar labeled C. The C program is modeled on the CLP($\mathcal{R}$) program, i.e. it uses a recursive procedure call and the four input variables are arguments to the procedure. The program uses double precision floating point and is compiled using the portable C compiler with –O option.

Many aspects of these results are interesting. First we note that the biggest improvement comes from the future redundancy optimization. When the computation involves 360 recursive steps, 359 redundant inequalities are accumulated if the optimization is not applied, and this clearly must induce a very large overhead. Second, there is also a tremendous improvement gained from a special treatment of trivial constraints. This is especially the case where the program essentially becomes functional (i.e: for Q_1) and when the floating point hardware is used, as the floating point emulation overhead otherwise tends to mask the overhead of the constraint solver. However, even those cases where constraints do not become completely trivial benefit substantially form this treatment. While the space saving from the mutual exclusion optimization is very important, the result for speed is disappointing. However, it turns out to be more useful where choice points are large, and when more backtracking is required. For example, a version of the *mortgage* program with the base case first shows the future redundancy optimization to be less important, and mutual exclusion more important. This is because the nondeterministic base-case-first version of the program essentially doubles the number of substantial equations that need to be solved. (Due to space limitations, the results of the base-case-first version are not included here). Finally, the comparison with the C program is somewhat encouraging. We should stress that the C program was not written for speed, but nonetheless it suggests that even with an abstract machine emulator it should be possible in many cases to execute specialized versions of CLP($\mathcal{R}$) programs as efficiently as the corresponding program written in a conventional language.

8 Conclusion

A number of simple global analysis techniques and optimizations for CLP($\mathcal{R}$) compilation have been described and tested. We have shown that these techniques can lead to significant performance improvements.

We emphasize that although the test programs were hand coded, there was nothing in the code that could not be produced by a CLP($\mathcal{R}$) compiler using the result of global optimization together with the usual peephole optimizations. However it remains to be demonstrated in practice that the optimizations can be automated.

With a compiler that automates the optimizations, it will also be possible to test the optimizations on a large set of benchmarks, rather than just on a single program as we have done. We believe however that the *mortgage* program is appropriate for a preliminary evaluation of the optimizations, and that the results show that the construction of such a compiler is worthwhile. It can be usefully queried in many different ways, leading to a variety of calling patterns and thus to the constraints having a variety of different properties.

In general we believe that in the compilation of CLP languages the required analyses in an optimizing compiler will partly be quite straightforward modifications of those used with Prolog, for example mode analysis. Others, however, such as future redundancy detection, will be novel and specific to the particular CLP language being considered. It seems likely that such optimizations will be necessary to make the performance of CLP systems competitive with conventional programming languages.

References

[1] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proc. of the Fourth ACM Symposium on Principles of Programming Languages*, 238–252, 1977.

[2] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables in a Program. *Proc. of the Fifth ACM Symposium on Principles of Programming Languages*, 84–97, 1978.

[3] S. K. Debray. Unfold/Fold Transformations and Loop Optimization of Logic Programs. *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, SIGPLAN Notices 23, No. 7, 297–307, 1988.

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

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

[6] T. Hickey and S. Mudambi. Global Compilation of Prolog. *Journal of Logic Programming*, **7**, 193–230, 1989.

[7] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. *Proc. of the Fourteenth ACM Symposium on Principles of Programming Languages*, 111-119, 1987.

[8] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. *Proc. of the 4th International Conference on Logic Programming*, 196–218, 1987.

[9] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap. *The CLP($\mathcal{R}$) Language and System*. IBM Research Report RC 16292 (#72336), 1990.

[10] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap. *An Abstract Instruction Set for Compiling CLP($\mathcal{R}$)*. Forthcoming IBM technical report, 1991.

[11] K. Marriott and H. Søndergaard. Analysis of Constraint Logic Programs. *Proc. of the 1990 North American Conference on Logic Programming*, 521–540, 1990.

[12] C. S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming*, **2**, 43–66, 1985.

[13] S.S. Muchnick and N.D. Jones (Eds.). *Program Flow Analysis: Theory and Applications*, Prentice-Hall Inc., 1981.

[14] A. Taylor. LIPS on a MIPS: Results from a Prolog Compiler for a RISC. *Proc. of the 7th International Conference on Logic Programming*, 174–185, 1990.

[15] J. D. Ullman and A. Van Gelder. Efficient Test for Top-Down Termination of Logical Rules. *Journal of the ACM* **35** (2), 345–373, 1988.

[16] P. Van Roy and A.M. Despain. The Benefits of Global Dataflow Analysis for an Optimizing Prolog Compiler. *Proc. of the 1990 North American Conference on Logic Programming*, 501–515, 1990.

[17] R. Warren, M. Hermenegildo and S.K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. *Proc. of the 5th International Conference and Symposium on Logic Programming*, 684–699, 1988.