Optimizing Compilation for CLP(\mathcal{R})

Kim Marriott* Harald Søndergaard†
Peter J. Stuckey† Roland H.C. Yap*†

Abstract

Constraint Logic Programming (CLP) is a recent innovation in programming language design. CLP languages extend logic programming by allowing constraints from different domains such as real numbers or Boolean functions. This gives considerable expressive power and flexibility and CLP programs have been proven to be a high-level programming paradigm for applications based on interactive mathematical modeling. These advantages, however, are not without cost. Implementations of CLP languages must include expensive constraint solving algorithms tailored to the specific domains. Indeed, performance of the current generation of CLP compilers and interpreters is one of the main obstacles to the widespread use of CLP. Here we outline the design of a highly optimizing compiler for CLP(\mathcal{R}) , a CLP language which extends Prolog by allowing linear arithmetic constraints. This compiler is intended to overcome the efficiency problems of the current implementation technology. The main innovation in the compiler is a comprehensive suite of program optimizations and associated global analyses which determine applicability of each optimization. We describe these optimizations and report very promising results from preliminary experiments.

1 Introduction

One of the most promising innovations in recent programming language design is the amalgamation of constraint programming and logic programming. Constraints provide a powerful and natural programming paradigm, in which the objects of computation are not explicitly constructed but rather they are implicitly defined using constraints. This allows for very high-level programming in which the program directly refers to objects and relationships from the application domain. Logic programming, on the other hand, offers a convenient modeling language for describing knowledge and rules which also has the attraction of a simple, declarative semantics. Furthermore, logic has obvious qualities as a database language, and it even serves as a compact formalism for concurrent programming. The combination, constraint logic programming, combines the best features of constraint programming and logic programming.

Constraint logic programs are ideal for solving problems that require interactive mathematical modeling. Specific applications have been in many diverse areas. One of the major application areas has been electrical circuit analysis, synthesis and diagnosis [5]. Civil engineering [10] and mechanical engineering have also attracted some attention. This is because engineering applications tend to combine hierarchical composition of complex systems, mathematical or boolean models, and—especially in the case of diagnosis and design—deep rule-based reasoning; hence they are particularly suited to constraint logic programming. Another major area has been options trading [6] and financial planning [2] where applications have taken the form of expert systems involving mathematical models. Other applications are in traditional operations research problems, such as cutting stock and scheduling. More exotic applications have included restriction site mapping in genetics [17] and generating test data for communications protocols.

CLP(\mathcal{R}) is an extension of Prolog incorporating real arithmetic constraints. Originally developed at Monash University, CLP(\mathcal{R}) is the prototypical constraint logic programming language. But many other CLP languages are now available. Some, such as CLP(\mathcal{R}) , are research systems, while others are commercial products, for example ‘CHIP’ from ECRC (Bull, ICL, and Siemens), and ‘Prolog-III’ from Prologia. While these systems are all very expressive and ideal for rapid prototyping, constraint solving is generally very expensive. For some applications, existing CLP systems do offer a performance comparable with solutions in conven-

*Dept. of Computer Science, Monash University, Clayton 3168, Australia.
†Dept. of Computer Science, University of Melbourne, Parkville 3052, Australia.
tional programming languages, but generally performance is the main obstacle to the widespread use of CLP. This is perfectly reasonable, as these CLP systems are all offshoots of first-generation research systems and cannot be expected to have reached a mature, industrial-strength level.

In this paper we outline the design of a highly optimizing compiler for CLP(R) which we are currently building. This compiler is a second generation implementation which is intended to overcome the efficiency problems of the current implementation technology. The main innovation in the compiler is a comprehensive suite of program optimizations. Our earlier studies [7, 9, 14] have shown that each of these optimization techniques can lead to orders of magnitude improvement in execution time or space for particular classes of programs. Not all optimizations are applicable to each program, but for most programs at least one of the optimizations is applicable. The design of the highly optimizing compiler is quite complex as determining applicability of each optimization technique requires sophisticated global analysis which determine information about possible and definite interaction between constraints. Another complication is that the different optimizations interact in non-trivial ways. Performing one optimization in one part of the program may preclude performing a different optimization in another part of the program. Our preliminary studies given here indicate the highly optimizing compiler will, for programs which only make use of simple constraints, be able to generate “industrial strength” code of a quality which is comparable to that generated for an equivalent program written in a conventional programming language. On the other hand, for programs which require true symbolic constraint manipulation, the compiler will generate code which is substantially more efficient than that produced by the current generation CLP(R) compiler.

This paper is a continuation of our earlier work on the implementation of CLP(R) [8, 7, 11] and on preliminary studies of each of the optimizations considered here. The main contributions of the present paper are to study the effects of the optimizations together and to sketch how they can be combined in highly optimizing compiler for CLP(R). The proposed compiler is also related to recent experimental compilers for Prolog, which also make use of global program analysis [16, 15]. These have given rise to dramatic performance improvements and we conjecture that the same will hold true for CLP(R). Indeed, we hope for even bigger performance improvements because linear arithmetic constraint solving is significantly more expensive than unification. A main difference with global Prolog compilers is that here we concentrate also on optimizations and consequent dataflow analyses which are specific to linear constraint solving.

In Section 2 we describe CLP(R) using an example. A reader with an elementary understanding of Prolog should recognize the basic ideas. In Section 3 we briefly discuss the existing compiler and abstract machine CLAM. Section 4 introduces a suite of code optimizations specific to arithmetic constraints and CLP(R) . Finally, in Section 5 we discuss the design of a highly optimizing compiler which uses global analysis to achieve these optimizations.

2 Constraint Logic Programming

CLP(R) allows the expression of real arithmetic constraint as well as Prolog-style constraints over uninterpreted terms. Figure 1 shows a simple program for computing mortgage repayments. The mortgage predicate relates the parameters of principal, life of the mortgage (in months), annual interest rate (%) which is compounded monthly, the monthly payment, and finally the outstanding balance. Notice that recursion is permitted. The program’s first rule expresses the fact that a mortgage for one month is given by a simple interest calculation: interest is found by multiplying monthly interest rate (annual interest rate/1200) by principal, and the balance is then principal plus interest minus payment. Mortgages for more than one month are defined by the second rule which effectively calculates the new “principal” by adding interest and subtracting payment, and then proceeds recursively to calculate a mortgage of one less month, given the new principal.

Execution proceeds in Prolog style from an initial query or goal by repeatedly rewriting the goal using the rules in the program. Each rule $H :- B$ may be used to replace the atom $H$ in the goal by the body of the rule $B$. As constraints are encountered in the rules they are added to the constraint store. The constraints in the store are always required to be consistent. If adding a constraint will lead to inconsistency, execution backtracks to the last point where there was a choice of rule for the rewriting step, and chooses a different rule.

For example, one may ask “how much will I have to pay per month for an $80,000$ mortgage of 30 years at 9.5%?” This is expressed by the
goal \text{mortgage}(80000, 360, 9.5, MP, 0) which gives the answer $MP = 672.68$. The evaluation can be seen to be similar to that of a logic program, except that tests for constraint satisfaction replace unification.

A key difference between a CLP($\mathcal{R}$) program and a corresponding program written in a conventional language is that the CLP($\mathcal{R}$) program specifies relationships between its arguments. This means that the same program can be used to answer many different types of goal. For example, instead of the goal above, we may ask a more complex question such as “what is the relationship between principal, monthly payment, and balance, given a 30 year mortgage at 15%?” The goal \text{mortgage}(P, 360, 15, MP, B) returns the appropriate relationship $P = 79.09 \times MP + 0.0114 \times B$. This also illustrates another important feature of constraint logic programs—the ability to return symbolic answers. An even more general goal is

\[
B \geq 0, \quad B \leq 1000, \quad MP \geq 900, \quad MP \leq 1100, \quad \text{mortgage}(80000, T, 9.5, MP, B)
\]

which asks how long a mortgage is needed if the monthly payment is kept between 900 and 1100 and the final balance should be under 1000. The answer constraint returned is

\[
T = 108 \wedge B = -169.702 \times MP + 187478 \wedge 1098.86 \leq MP \leq 1100
\]

A rather different and much more complex program would be required to answer this goal in a conventional language without constraints.

3 Compilation of CLP($\mathcal{R}$)

Execution of CLP($\mathcal{R}$) programs involves repeatedly adding a constraint to the constraint store and checking that the store remains satisfiable. Thus a successful execution path involves a growing number of constraints in the store. For example the mortgage goals each generate more than 1000 constraints. Hence the key issues in implementation are incremental constraint solving and dealing with a growing constraint store. While the constraint solvers in CLP($\mathcal{R}$) are specialized for incremental solving, in the worst case they may require processing virtually the entire constraint store to add a single new constraint. Thus the number of constraints in the constraint store can have a major impact on runtime speed.

The first CLP($\mathcal{R}$) implementation was based on an interpreter which consisted of a PROLOG-like interpreter rewriting engine and a set of constraint solvers: a unification solver, linear equation solver, linear inequality solver and a non-linear solver, together with an interface which translated constraints into a canonical form suitable for the constraint solvers. The constraint solvers are organized in a hierarchy: unification solver, direct evaluation/testing, linear equation solver and linear inequality solver, where the later solvers are more expensive to invoke than earlier ones. The interface sends constraints to the earliest solver in the hierarchy that can deal with the constraint. Thus the more expensive solver is only invoked when the previous one is not applicable. For example, when solving a linear equation, the linear equation solver is often sufficient and only when all the variables in the equation are involved in inequalities is it necessary to also use the inequality solver.

The current implementation of CLP($\mathcal{R}$) is a compiler. It compiles down into an abstract machine, the CLAM. The CLAM is an extension of the Prolog WAM architecture to deal with arithmetic constraints. Because the constraint solvers deal principally with linear constraints, the main arithmetic instructions in the CLAM construct the data structures representing linear arithmetic forms. These data structures are in a form which
can be used directly by the constraint solvers. The constraint solving hierarchy used in the interpreter is retained but is more effective since some run-time decisions in the interpreter can now be shifted to compile-time. Figure 2 gives a more detailed overview of the organization of the constraint solvers.

To give a flavor of the CLAM (see [7] for details), we now describe the compilation of a simple constraint \( 5 + X = Y \). We will assume that \( X \) is a new variable and the equation store contains 
\( Y = Z + 3.14 \). The following CLAM code could be produced by the compiler:

```clam
initf 5  /:5
addf_var 1 X  /:5 + X
addf_val -1 Y  /:1.86 + X - Z
solve_eq0  solve :1.86 + X - Z = 0
```

On the left hand side we give the CLAM instructions executed and the right shows the operation of the CLAM instructions to construct the linear form \( (lfl) \). The original constraint is rewritten into a linear canonical form, \( 5 + X - Y = 0 \) to compile. This CLAM code executes as follows: firstly a new linear form is initialized to \( 5 \), and \( X \), being a new variable, is added directly. Then \( Y \) is added which entails adding it’s linear form, \( Z + 3.14 \). After the first three instructions have constructed a linear form, the last `solve_eq0` instruction is used to represent the equation \( lfl = 0 \). In general the `solve_eq0` may reduce to an assignment, a test or a call to the equation solver.

CLAM instructions operate below the level of a single constraint and can be optimized and combined in various ways. The current implementation of the compiler consists of a core CLAM instruction set which is sufficient to execute a CLP(R) program without requiring global analysis. At present, the compiler does not have any integrated global analyzer and utilizes the core CLAM instruction set together with some peephole optimizations and some rule level optimizations. For example, when we know that a constraint is always satisfiable, it may be possible to decide at compile time how the constraint is to be represented in the constraint store at runtime and simply add that data structure.

### 4 Optimizations

In this section we introduce a suite of optimizations for CLP(R) programs using a worked example. Though we only informally justify the correctness of each of the optimization methods, global analysis methods (see section 5) can be used to determine the applicability of each method. Consider the following CLP(R) program defining the relation `fib` where `fib(n, f)` if \( f \) is the \( n \)th Fibonacci number. The program is for illustrative purposes only. It is a direct translation of the usual (computationally inefficient) mathematical definition of Fibonacci numbers.

**CLAM**

```
fib(N, F) :- N = 0, F = 1.
fib(N, F) :- N > 2, F = F1 + F2, N1 = N - 2, N2 = N - 1.
fib(N1, F1), fib(N2, F2).
```

**Reordering of Constraints.** Constraint addition can be moved to a later point in execution if the constraint involved does not affect the control flow in the intervening computation. Reordering reduces the size of the constraint store. Reordering requires determining possible interaction between constraints.

Consider the execution of the goal `fib(n, F)` where \( n \) is some number. The last clause leads to two new calls to `fib(n - 2, F1)` and `fib(n - 1, F2)`. Each subsequent call follows the same pattern where the first argument is constrained to a unique value. The second argument \( F \) can take any value, and is in that sense unconstrained. In subsequent calls \( F1 \) and \( F2 \) are also unconstrained in this sense. As \( F \) is unconstrained, the constraint 
\( F = F1 + F2 \) cannot cause failure, regardless of the values of \( F1 \) and \( F2 \). Hence moving it...
later does not change the execution of the goals \( \text{fib}(N1, F1) \) and \( \text{fib}(N2, F2) \). Similarly the constraint \( N2 = N - 1 \) does not affect the evaluation of the goal \( \text{fib}(N1, F1) \), so it can be placed after that goal. The resulting program is

\[
\begin{align*}
\text{(REO)} \\
\text{fib}(N, F) &: - N = 0, F = 1. \\
\text{fib}(N, F) &: - N = 1, F = 1. \\
\text{fib}(N, F) &: - N > 2, \ \\
& \quad N1 = N - 2, \text{fib}(N1, F1), \\
& \quad N2 = N - 1, \text{fib}(N2, F2), \\
& \quad F = F1 + F2 .
\end{align*}
\]

**Bypass of the constraint solver.** In many cases, by the time a constraint is encountered, it is a simple Boolean test or assignment. In this case a call to the solver can be replaced by the appropriate test or assignment. This both decreases the size of the constraint store and also removes calls to the solver. Application of this optimization requires determining when variables are constrained to a unique value and when they are unconstrained.

Consider the execution of the goal \( \text{fib}(n, F) \) in program \text{REO}. By replacing constraints with tests and assignments we can in fact remove all access to the constraint solver. This results in the following program:

\[
\begin{align*}
\text{(BYP)} \\
\text{fib}(N, F) &: - N = test \ 0, F = assign \ 1. \\
\text{fib}(N, F) &: - N = test \ 1, F = assign \ 1. \\
\text{fib}(N, F) &: - N > test \ 2, \ \\
& \quad N1 = assign \ N - 2, \\
& \quad N2 = assign \ N - 1, \\
& \quad F = assign \ F1 + F2 .
\end{align*}
\]

Here \( > test \) simply evaluates both sides and tests whether the \( \geq \) relationship holds, while \( assign \) evaluates the right hand side and sets the value of the left hand side to this value. These are primitives in an internal language suitable for better code generation.

**Deterministic rules.** By the time an atom is resolved, it often happens that only one of the clauses defining the atom has constraints consistent with the current store. In this case there is no need to set up a choicepoint as subsequent backtracking will lead to failure.

This behavior occurs in the above example goal. Because \( n \) is fixed only one of the clauses will be applicable since they are mutually exclusive on \( n \). Rather than producing expensive choicepoints we can replace the multiple clauses of (BYP) by a single deterministic clause:

\[
\begin{align*}
\text{(DET)} \\
\text{fib}(N, F) &: - \text{if} (N = test \ 0) \text{ then} \\
& \quad F = assign \ 1. \\
& \quad \text{else if} (N = test \ 1) \text{ then} \\
& \quad F = assign \ 1. \\
& \quad \text{else if} (N >= test \ 2) \text{ then} \\
& \quad N1 = assign \ N - 2, \\
& \quad N2 = assign \ N - 1, \\
& \quad \text{fib}(N1, F1), \\
& \quad \text{fib}(N2, F2), \\
& \quad F = assign \ F1 + F2, \\
& \quad \text{else fail.}
\end{align*}
\]

The resulting program is in this case completely procedural, but only applicable to goals of the form \( \text{fib}(n, F) \).

**Refinement.** In this optimization, constraints which will eventually become redundant are added to the start of a clause. The advantage is that these new constraints make information available earlier in the computation, and so may improve the operational behavior by guiding subsequent execution away from unprofitable choices.

Consider the original program \text{FIB} and the goal \( \text{fib}(N, 6) \). There are no answers to this goal but evaluation will not terminate, since there is an infinite rewriting sequence. **Refinement** can be used to overcome this problem. For each call to \( \text{fib}(N, F) \) every answer must satisfy the constraints \( N \geq 0 \land F \geq 1 \). We refine the program by adding the redundant constraints to every clause body where the calls are made. Hence the refined program is:

\[
\begin{align*}
\text{(REF)} \\
\text{fib}(N, F) &: - N = 0, F = 1. \\
\text{fib}(N, F) &: - N = 1, F = 1. \\
\text{fib}(N, F) &: - N >= 2, F = F1 + F2, \\
& \quad N1 = N - 2, N2 = N - 1, \\
& \quad N1 >= 0, F1 >= 1, \\
& \quad N2 >= 0, F2 >= 1, \\
& \quad \text{fib}(N1, F1), \text{fib}(N2, F2). \\
\end{align*}
\]

The chief advantage of the refined program is that it will work with a wider class of calls than the original. For instance, it terminates with answer no for goal \( \text{fib}(N, 6) \). As refinement may change the operational behavior of the program, albeit for the better, it is different in nature to the other steps in the optimization. Thus refinement might be performed in an optional pre-compilation stage, in which the programmer can intervene.

**Removal of redundant constraints.** Another common source of redundancy in the solver are constraints which have become redundant in
the sense that their information is implied by other constraints in the current store. Execution can be optimized by adding instructions which remove these constraints from the store, as this decreases the size of the constraint store. This optimization requires determining definite interaction between constraints.

Refinement in general adds redundant constraints. Clearly for the program (REF) \( N \geq 2 \land N1 = N - 2 \land N2 = N - 1 \rightarrow N1 \geq 0 \land N2 \geq 0 \). Hence the last two constraints can be removed without affecting computation. More importantly some constraints are initially non-redundant but later become redundant. Consider the constraint \( N \geq 2 \). In the call to \( fib(N1, F1) \) the first constraint encountered is either \( N1 = 0 \), \( N1 = 1 \) or \( N1 \geq 2 \). But \( N1 = N - 2 \), so in each case the constraint \( N \geq 2 \) is made redundant. Hence it can be removed before the call to \( fib(N1, F1) \) without affecting execution. Similar reasoning applies to the constraints \( F1 \geq 1 \) and \( F2 \geq 1 \) which can be removed just before their respective calls.

Removal commands can also be reordered, just like constraint additions. The removal of constraint \( N \geq 2 \) can be moved to just after its addition, because it cannot cause failure over the intervening period. In this case the constraint is tested for satisfiability with respect to the current constraints, and then not added to the solver. This is the best case for removal, and is handled specially by the solver. The resulting program is (REM) below. The removal of constraints \( F1 \geq 1 \) and \( F2 \geq 1 \) cannot be made earlier without more knowledge about the form of the original goal.

(REM)

\[
\begin{align*}
\text{fib}(N, F) & :\quad N = 0, F = 1. \\
\text{fib}(N, F) & :\quad N = 1, F = 1. \\
\text{fib}(N, F) & :\quad \text{add/removed}(N \geq 2), \quad F = F1 + F2, \\
& \quad N1 = N - 2, N2 = N - 1, \\
& \quad F1 \geq 1, F2 \geq 1, \\
& \quad \text{remove}(F1 \geq 1), \\
& \quad \text{fib}(N1, F1), \\
& \quad \text{remove}(F2 \geq 1), \\
& \quad \text{fib}(N2, F2).
\end{align*}
\]

Removal of redundant variables. A common source of redundancy in the constraint solver are variables which will never be referred to again. Execution can be improved by adding instructions which project out variables from the current constraints in the store as this helps keep down the size of the constraint store. Clearly this optimization requires determining which variables are still alive.

Consider the original program (FIB) and the variable \( F \) in the last clause. After the constraint \( F = F1 + F2 \) the variable is never again referred to within the clause. If it is not required after the call, it can be removed at this point. Examining the recursive calls \( fib(N1, F1) \) and \( fib(N2, F2) \) in the body we note that the last arguments \( F1 \) and \( F2 \) are never referred to again. Hence we can optimize the program by giving two versions of the clauses— one for when the last argument will be referred to later, and one when it is not (fib^ad). For the second set of clauses we can remove the second argument within the call. This reduces the number of variables and constraints in the solver.

(REV)

\[
\begin{align*}
\text{fib}(N, F) & :\quad N = 0, F = 1. \\
\text{fib}(N, F) & :\quad N = 1, F = 1. \\
\text{fib}(N, F) & :\quad N \geq 2, F = F1 + F2, \\
& \quad N1 = N - 2, N2 = N - 1, \\
& \quad F1 \geq 1, F2 \geq 1, \\
& \quad \text{fib}(N1, F1), \\
& \quad \text{fib}(N2, F2).
\end{align*}
\]

Empirical Results

To illustrate the effect of the optimizations, we show the effect on a number of different programs and goals. Consider the fib program, with the goal \( \text{fib}(16, F) \) where we have specified the value of \( N \). We give below the execution times in CPU seconds using a modified version of CLP(R) v1.1 on a SparcStation 2 with the speedup ratio to the original unoptimized goal given in brackets. The maximum amount of space utilized by the constraint solvers (measured in nodes) is also given together with the ratio of the solver space saved compared with the original goal.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Time</th>
<th>Solver Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIB</td>
<td>14.2 (1.0)</td>
<td>37267 (1.0)</td>
</tr>
<tr>
<td>REO</td>
<td>6.5 (2.2)</td>
<td>12774 (2.9)</td>
</tr>
<tr>
<td>BYP+DET</td>
<td>8.1 (1.8)</td>
<td>6417 (5.8)</td>
</tr>
<tr>
<td>ALL</td>
<td>0.8 (17.8)</td>
<td>64 (∞)</td>
</tr>
</tbody>
</table>

The optimizations applied here are FIB for the original program, REO for reordering, BYP+DET for bypass and determinism applied together and ALL combines REO+BYP+DET together. Both REO and BYP+DET individu-
ally give time and space improvements but **ALL** illustrates how the individual optimizations can combine even better together. The end result achieves a large speedup because it does not use the solver at all and hence also no solver space.¹

We can apply more optimizations with goals where the value of $F$ is given but not $N$. The goal used in the table below is $\text{fib}(N, 2854)$ with the following optimizations applied individually to $\text{FIB}$: $\text{BYP+DET, REM}$ (constraint removal) and $\text{REV}$ (variable removal). The $\text{BYP+DET}$ optimization arises because once the first recursive call to $\text{fib}$ returns, the value of $N$ becomes determined and the second recursive call to $\text{fib}$ uses a fixed $N$.

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Time</th>
<th>Solver Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIB</td>
<td>9.3</td>
<td>(1.0)</td>
</tr>
<tr>
<td>REF</td>
<td>29.7</td>
<td>(0.3)</td>
</tr>
<tr>
<td>BYP</td>
<td>4.9</td>
<td>(1.9)</td>
</tr>
<tr>
<td>+ DET</td>
<td>9.2</td>
<td>(1.0)</td>
</tr>
<tr>
<td>REM</td>
<td>8.3</td>
<td>(1.1)</td>
</tr>
<tr>
<td>REV</td>
<td>4.0</td>
<td>(2.3)</td>
</tr>
<tr>
<td>ALL</td>
<td>60874</td>
<td>(1.9)</td>
</tr>
<tr>
<td></td>
<td>294337</td>
<td>(0.2)</td>
</tr>
<tr>
<td></td>
<td>10817</td>
<td>(5.6)</td>
</tr>
<tr>
<td></td>
<td>60684</td>
<td>(1.0)</td>
</tr>
<tr>
<td></td>
<td>39612</td>
<td>(1.5)</td>
</tr>
<tr>
<td></td>
<td>2332</td>
<td>(26.1)</td>
</tr>
</tbody>
</table>

The original $\text{FIB}$ program has the drawback that it may run forever for some goals, e.g. supplying an incorrect $F$. Applying $\text{REF}$ (refinement) fixes this by ensuring that all such goals will terminate. However the results show that the refined program is about three times slower with five times increased space. Because refinement adds redundant constraints, applying the constraint optimizations $\text{REM, REV, REO}$ can remove the extra constraints and overhead introduced by refinement. The final row in the table above, **ALL** consists of the refined program $\text{REF}$ with $\text{BYP+DET+REM+REV}$ applied. The resulting program is twice as fast as $\text{FIB}$ without any of $\text{FIB}$'s drawbacks.

As a final illustration, the following table shows the results of applying the optimizations to the mortgage program using the goal $\text{mortgage(P, 720, 8, B, 800)}$. The dominant optimization with goal patterns of this form is constraint removal which gives 40 times speedup over the original. Individually both $\text{BYP+DET}$ and $\text{REV}$ don't show any speedup as their effect is swamped by other constraint solving. However combining all three optimizations (**ALL**) shows the advantages of using them together.

¹The 64 nodes in the table represents a small constant overhead required in the system.

![Diagram of the optimizing compiler](image)

**Figure 3: The optimizing compiler**

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Time</th>
<th>Solver Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>MG</td>
<td>21.5</td>
<td>(1.0)</td>
</tr>
<tr>
<td>BYP</td>
<td>21.4</td>
<td>(1.0)</td>
</tr>
<tr>
<td>+ DET</td>
<td>0.5</td>
<td>(43.0)</td>
</tr>
<tr>
<td>REM</td>
<td>21.2</td>
<td>(1.0)</td>
</tr>
<tr>
<td>REV</td>
<td>0.3</td>
<td>(71.7)</td>
</tr>
<tr>
<td>ALL</td>
<td>14401</td>
<td>(1.0)</td>
</tr>
<tr>
<td></td>
<td>12963</td>
<td>(1.1)</td>
</tr>
<tr>
<td></td>
<td>7211</td>
<td>(2.0)</td>
</tr>
<tr>
<td></td>
<td>10090</td>
<td>(1.4)</td>
</tr>
<tr>
<td></td>
<td>1465</td>
<td>(9.8)</td>
</tr>
</tbody>
</table>

## 5 Design of the Compiler

In the previous section we indicated how the key to faster execution of constraint logic languages lies with sophisticated compile-time optimizations. In this section we sketch the design of a CLP(R) compiler which will perform these optimizations.

The optimizing compiler has three main components (see Figure 3): the **optimizer** which takes a program as input and performs high-level optimizations; the **code generator**, which takes the output of the optimizer and translates it into CLAM code; and an **analyzer** which is used by the optimizer to determine applicability of the various optimizations. One reason for this architecture is that it allows us to leverage from existing technologies and software. In particular the CLAM emulator will be an extension of that used in the existing CLP(R) compiler.
The Optimizer

Input to the highly optimizing compiler essentially consists of standard CLP(\(R\)) programs. However, because of the need to support global analysis with incremental compilation the language must be extended in two ways. The first extension is to include module declarations which allow the programmer to list predicates to be exported or imported by the module. This allows predicates local to a module to be heavily optimized as they can only be called from within the module. Of course module declarations are also important as they support programming in the large. The second extension is to allow declarations which give advice to the compiler about the type of goals particular predicates will be typically used in. This allows the compiler to target the optimizations for these cases. An example declaration would be to declare that fib will be primarily called with either its first or its second argument bound to a number.

The optimizer takes the original program and repeatedly performs optimizations to it. Applicability of the different optimizations is dictated by the analyzer. The optimizer will introduce different versions of the same predicate for different types of calls to the predicate when this warranted by user declarations that these are typical types of call to the predicate and the different versions allow different optimizations.

Which optimizations are performed by the compiler depends on the type of constraints manipulated by the program. For those parts of the program in which constraint solving is simple: bypass of the constraint solver, and deterministic rule optimizations are applicable. For the other parts of the program in which more complex constraint solving is required: the removal of redundant variables is used to remove redundant equality constraints and the removal of redundant constraints is used to remove redundant inequality constraints. The reordering of constraint addition and removal optimization facilitates and amplifies the effect of these other optimizations. In particular, reordering can be used to allow the solver bypass optimization as a constraint may be moved to a point where it becomes a simple assignment. This was illustrated in the Fibonacci example. The compiler will also perform traditional optimizations such as code motion and tail recursion optimization.

In addition there will be a number of optional source-to-source transformations, which do not change the declarative semantics of the program, that is, its answers, but may change its operational behavior. These include clause refinement, introduced in the previous section, and partial evaluation. In partial evaluation, clauses are unfolded which reduces some of the overhead of rewriting.

Analyzing and determining the applicability of optimizations is a complex task. Different optimizations may interfere - reordering a clause may invalidate a bypass of the constraint solver optimization. Even if it is possible to apply an optimization, optimizations such as variable and constraint removal are not guaranteed to produce speedups or space savings because they affect future constraint solving as well as possibly changing a considerable portion of the constraint store. Furthermore the optimizations interact with each other (in fib we saw it was synergistic) and in the worst case can give much worse performance. It is impractical to consider all possible combinations of interactions. Instead we plan to use a “greedy” heuristic strategy. We will first compute the call graph for the predicates and then optimize the predicates in order of their distance from the bottom of the graph. This corresponds to the heuristic that code in inner loops should be optimized in preference to code in outer loops, as it will be executed more often.

The Analyzer

Analysis is formalized in terms of abstract interpretation [3]. Consider the idea of a constraint logic program interpreter which answers goals by returning not only a set of answer constraints, but also a thoroughly annotated version of the program: For each program point it lists the current constraint store for each time that point was reached during evaluation of the given query. Since control may return to a program point many times during evaluation, each annotation is naturally a (possibly infinite) set of constraints. Properly formalized, this idea leads to the notion of a collecting semantics, a semantics which gives very precise dataflow information, but which is of course not finitely computable in general. However, if we replace the possibly infinite sets of constraints by more crude “approximations” or “descriptions” then we may obtain a dataflow analysis which terminates in finite time. This is the idea behind abstract interpretation of logic programs and constraint logic programs [4].

As an example consider the reordered Fibonacci program, REO, from Section 4. If this program is analyzed for the class of calls in which the first
argument is bound to a number then the following annotated program results. The constraint
description \{X, Y, Z, \ldots\} is read as: at this point the constraint store constrains the variables \(X, Y, Z, \ldots\),
to take a unique value, that is, they are bound to a number.

\[
\begin{align*}
\text{fib}(&N, F) := \\
\text{fib}((N, F)) := &\begin{cases} 
N = 1, & (N, F) = 1, \\
N = 2, & (N, F) = 1, \\
N > 3, & (N, F) = 1,
\end{cases} \\
N1 = & N - 2, (N, N1) \\
N2 = & N - 1, (N, N1, F1, N2) \\
F = & F1 + F2.
\end{align*}
\]

To facilitate the rich variety of analyses required in the compiler, the analyzer will be a generic tool —
different analyses will result by providing different parametric functions. The analyzer itself
will be similar to one already built for analyzing logic programs. However, this tool will need to
be extended to handle arithmetic constraints and more sophisticated analysis domains. Some of the
individual analyses, such as derivability analysis, will be similar to those used in compilation of logic
programs, while others, such as bounds or interval analysis, require new techniques.

We will be primarily interested in finding out about possible and definite interaction between
constraints. We have designed four description domains which capture this information.

- **Pos** is a description domain consisting of propositional formula capturing groundness
  information about variables and definite dependence [1]. For example the propositional
  formula \(X \land (Y \rightarrow Z)\) indicates that the variable \(X\) is constrained to take a unique value,
  and that if \(Y\) is ever constrained to a unique value, then so is \(Z\).

- **CHull** is a description domain consisting of sets of linear constraints. It is used to de-
  scribe the definite values and constraints between variables which have arithmetic type.

- **LSign** is a description domain consisting of linear constraints in which coefficients are ab-
  stracted by their sign. It is used to describe the possible interaction between variables which
  have arithmetic type.

- **Free** is a description domain which captures information about “freeness” of variables and
  possible dependence between non-arithmetic variables.

There are two points to note. First, the analyzer
needs to handle delayed non-linear constraints. We
plan to do this using “continuations” which have
been developed as a means of handling delayed negative
literals in Prolog [13]. Second, the analyzer
would be much more efficient if it was incremental.
This is because a new analysis must be performed
each optimization that modifies the program.
Thus it would be better if the analyzer only reid
those parts of the analysis which the modifications
may have invalidated.

**Extended CLAM**

The CLAM emulator and abstract machine consists of two components. One is the core CLAM
instruction set which is sufficient to execute all programs without requiring global optimization. The
other is the extended CLAM which adds more instructions specifically to deal with optimizations.
The CLAM architecture is useful for this as it operates below the level of a constraint and the optimi-
izations described all require modifying the operation of constraint solving to be effective and hence
slot neatly with the rest of the CLAM architecture.

We will give an overview of what the extended CLAM instructions look like rather than describe
them in detail. **BYP + DET** adds instructions which are like traditional arithmetic and
conditional instructions on conventional architectures. These operate directly on “unboxed” floating
point point values whereas most of the values in the CLAM (like the WAM) are tagged and may repre-
sent bindings. **REM** adds two classes of instructions: **solve_no_add** which are like the regular
solve instructions except that the constraint is not added to the constraint store (this corresponds to
**add_remove** in the **REM** example). More general constraint removal instructions are used to remove
other previous constraints. **REV** adds a variety of instructions to remove dead variables. For example
instead of an **addpf_val c**, X instruction in the Section 3 example, there are **addpf_val_elim c**, X instructions
which add X to the linear form being constructed as well as removing any linear form corresponding to X.
Instead of an **initpf c** initialization instruction, using **REV** allows us to
directly reuse storage and use “in-place” modifications to data structures with an **initpf_dead X**, c instruction
which uses for the current linear form being constructed, the old linear form for \(X\) and adds the constant \(c\). There are also other instructions such as **remove_var X** which projects away the variable \(X\) thus removing it as well as in-
structions to handle variable removal in non-linear constraints.

6 Summary

We have introduced a comprehensive suite of program optimizations for CLP(R) and related constraint logic programming languages. Experiments show that each optimization offers spectacular improvement in runtime behavior for some programs. In the examples used, many optimizations are applicable and they interact in a synergistic fashion to produce large space and time improvements. We anticipate that bigger “real-world” programs will also benefit significantly because at least one of the optimizations in the suite will be applicable.

This has led us to design a compiler which utilizes the suite to produce highly optimized code for an abstract machine, and we are now in the process of implementing this compiler. We expect this compiler to produce programs that are at least an order of magnitude faster and more space efficient than existing implementations.

References


