Optimizing Compilation of Constraint Logic Programming Languages

Andrew D. Kelly[*] Andrew Macdonald[†] Kim Marriott[*]
Peter J. Stuckey[†] Roland H.C. Yap[‡]

Abstract

Constraint Logic Programming (CLP) is an innovation in programming language design. CLP languages extend logic programming by allowing constraints from different domains such as real numbers or Boolean functions. Building efficient compilers and interpreters for CLP languages has proved difficult due to the expensive constraint solving algorithms required for the domains. We present a highly optimizing compiler for CLP($\mathcal{R}$) a CLP language which extends Prolog by allowing linear arithmetic constraints. This compiler overcomes many of the efficiency problems of current implementations. The main innovation in the compiler is the incorporation of powerful program optimizations and associated sophisticated global analyses which determine information about various kinds of interaction among constraints. The optimizations presented in this paper are: reordering of constraints, bypass of the constraint solver, splitting and dead code elimination, future redundancy, removal of redundant variables, and no fail constraints. The six program optimizations are designed to remove the overhead of constraint solving when possible and keep the number of constraints in the store as small as possible. We present the performance of our compiler on a number of benchmark programs. Our results indicate that the highly optimizing compiler leads to a threefold increase in execution speed and one third decrease in space requirements when compared against the released version of the CLP($\mathcal{R}$) compiler (v1.2).

1 Introduction

One of the most promising innovations in recent programming language design are constraint programming languages, and in particular *constraint logic programming* (CLP) languages which combine constraints with logic programming. The key characteristic of these languages is a global constraint solver which is queried to direct execution and to which constraints are monotonically added. CLP languages have proved to be ideal for expressing problems that require interactive mathematical modeling and for expressing complex combinatorial optimization problems. However, CLP languages have mainly been considered as research systems, useful for rapid prototyping, but not really competitive with more conventional programming languages when performance is crucial. In general, performance is probably the main current obstacle to the widespread use of CLP. This situation is not surprising, as current CLP systems are off-shoots of first-generation research systems and general purpose constraint solving is expensive. Slowness of constraint solving is exacerbated by the addition without deletion of constraints which means that in current CLP systems the number of constraints in the global constraint solver grows rapidly as evaluation proceeds.

We present a highly optimizing compiler for CLP($\mathcal{R}$) which overcomes many of the efficiency problems of the current implementation technology. CLP($\mathcal{R}$) is the prototypical constraint logic programming language. It extends Prolog by incorporating real arithmetic constraints. The main innovation in the compiler is the incorporation of powerful program optimizations and associated sophisticated global analyses which determine information about various kinds of interaction among constraints. The six program optimizations are designed to remove the overhead of constraint solving when possible and keep the number of constraints in the store as small as possible. They have been carefully chosen so

[*]Dept. of Computer Science, Monash University, Clayton 3168, Australia.
[†]Dept. of Computer Science, University of Melbourne, Parkville 3052, Australia.
[‡]Dept. of Information Systems and Computer Science, National University of Singapore, Singapore 119260.

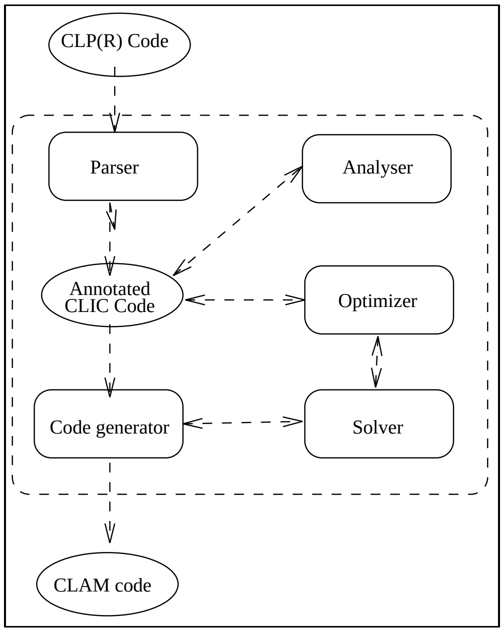

Figure 1: The optimizing compiler

that they span the program space in the sense that some optimization can be performed on almost any program.

Although some details of the optimizations and compiler design are specific to CLP($\mathcal{R}$) , the general ideas behind the optimizations and compiler design are applicable to any CLP language and also to other constraint programming paradigms such as concurrent constraint programming languages [19], constraint databases [7], constraint imperative languages [5] and constraint functional languages [4]. The main contributions of this paper are:

- A general design for optimizing compilers of CLP languages. The key idea is to view optimizations as source-to-source transformations on a language which extends CLP by providing assignments and tests and removal of constraints from the constraint solver.
- Design of a suite of transformations which remove much of the overhead of arithmetic constraint solving.
- Experimental verification that global program analysis together with program optimizations leads to large performance improvement in the execution of CLP languages, making them competitive with traditional languages.
- Additional experimental confirmation that sophisticated global program analysis and optimizations can be performed in reasonable time and is practical in real compilers.

The optimizing compiler has four main components (see Figure 1): the *optimizer* which performs the optimizations; a *global analyzer* and a *constraint solver* which provide information to guide the optimizer; and a *code generator* which produces CLAM abstract machine code. CLAM is an extension of the Prolog WAM [1] architecture which supports calls to constraint solvers.

The main complication in the design of the optimizer is the number of different optimizations. This is exacerbated by non-trivial interaction between the optimizations. Performing one transformation in one part of the program may preclude performing a different transformation in another part of the program. Another complication is the need for multi-variant specialization. Our solution to these potential difficulties is to uniformly view all optimizations as source-to-source transformations on an intermediate language called CLIC, standing for Constraint Logic Intermediate Code. Optimizations consisting of transformations and multi-variant specialization are applied to the CLIC code in multiple passes. This allows for a simple treatment of the optimizations and facilitates experimentation as it is easy to apply optimizations in different orders and to add new optimizations. Both the optimizer and code generator make use of the CLP($\mathcal{R}$) runtime solver to reason about constraints in the CLIC program.

CLIC is an extension of CLP($\mathcal{R}$) which also provides the usual imperative arithmetic commands and various specialized solver instructions. Indeed CLIC can be thought of as a hybrid imperative-logic constraint programming language. One important feature of CLIC is that it provides commands to remove constraints and variables from the global constraint solver. Thus execution of a CLIC program does not necessarily lead to more and more constraints in the constraint solver. A major role of the transformations in the compiler is to transform a CLP($\mathcal{R}$) program with monotonic constraint addition into an equivalent CLIC program in which the number of constraints in the global solver is bounded.

In fact the optimizer does not directly interact with the CLIC code, instead it works with an "annotated" CLIC program in which each program point in the program has an annotation which describes the constraints which will be encountered at this point during runtime. It is the role of the global analyzer to keep the annotation up to date whenever the optimizer changes the underlying CLIC program. For efficiency therefore, the analyzer is incremental and only re-analyzes those parts of the program which have been changed.

The global analyzer consists of a generic abstract interpretation engine that provides efficient fixpoint computation. It is used together with seven different but interacting analysis domains. New analysis domains can be easily added to the generic analyzer.

Empirical evaluation of the optimizing compiler has been extremely promising. Benchmark evaluation indicates that the optimizing compiler leads to a threefold increase in execution speed and one third decrease in space requirements. The speed of the optimizing compiler is also impressive, taking less than a minute to compile typical CLP($\mathcal{R}$) programs. The current compiler consists of about 54,000 lines of C++ code. The reason for the success of the optimizing compiler is, at least partly, due to the simple and declarative CLP semantics. This allows efficient and accurate global analysis as well as the use of many simple yet powerful optimizations. Indeed, in some sense the optimizing compiler can be viewed as an efficient program synthesizer as it takes an executable logical specification of the program and transforms this into specialized CLIC programs which handle different types of goals.

This paper continues our work on the implementation of CLP($\mathcal{R}$) [10, 9, 13, 8] and on preliminary studies of each of the optimizations considered here [11, 9, 17, 13, 16]. The only other paper which describes an optimizing compiler for a CLP language, that we are aware of, is [8] which describes a preliminary version of this compiler which only provided three of the six optimizations used in the current compiler. The main difference from this earlier work is that analysis and optimization were each performed in a single static phase. This design precluded many of the more powerful optimizations such as reordering and future redundancy which are included in the current compiler. Apart from being much more powerful, the current compiler and analyzer have a simple, extendible design and are also an order of magnitude faster.

In Section 2 we describe CLP($\mathcal{R}$) using an example. A reader with an elementary understanding of Prolog should recognize the basic ideas. We also briefly discuss the existing compiler. Section 3 describes the suite of transformations by means of a simple example. In Section 4 we describe the analyzer and optimizer in more detail. Section 5 describes our experimental evaluation of the compiler and Section 6 concludes.

2 CLP($\mathcal{R}$) and Existing Implementation Technology

In this section we describe CLP($\mathcal{R}$) using an example. Figure 2 shows a simple program for computing mortgage repayments. The `mortgage` predicate relates the principal (`Prin`), time of the mortgage (in months) (`Time`), annual interest rate (%) (`Rate`) which is compounded monthly, the monthly payment (`MP`), and finally the outstanding balance (`Bal`). Notice that recursion is permitted. The program's first rule expresses the fact that for mortgages of zero months the balance is the principal. Mortgages for more than one month are defined by the second rule which in effect 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.

```
mortgage(Prin, Time, Rate, MP, Bal) :-
    Time = 0,
    Prin = Bal.
mortgage(Prin, Time, Rate, MP, Bal) :-
    Time > 0,
    NTime = Time - 1,
    Int = Prin * Rate/1200,
    NPrin = Prin + Int - MP,
    mortgage(NPrin, NTime, Rate, MP, Bal).
```

Figure 2: Mortgage program

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 `mortgage(80000, 360, 9.5, MP, 0)` which gives the answer `MP = 672.68`. The evaluation can be seen as 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 `mortgage(P, 360, 15, MP, B)` returns the appropriate relationship `P = 79.09*MP + 0.0114*B`. This also illustrates another important feature of constraint logic programs—the ability to return answers in the form of relationships between variables.

Constraint logic programs, therefore, allow very simple and compact solutions to programming problems as the same program can be used to answer many different types of goals. This contrasts with a conventional language in which a separate program is required for each type of goal. Thus the above program really corresponds to a suite of conventional programs.

However, there is a price to be paid for this flexibility. General purpose constraint solving is expensive. This is especially true when the number of constraints in the store grows rapidly as evaluation proceeds. For example, in the above program, both goals generate more than 1000 constraints. For this reason efficient execution of constraint logic programs requires that the overhead of constraint solving be kept as small as possible.

The released version of the CLP($\mathcal{R}$) compiler (v1.2) is designed to do this, and is the result of nine years of development. It translates CLP($\mathcal{R}$) code into a core set of CLAM instructions and makes use of state of the art Prolog implementation technology, and highly specialized incremental constraint solving algorithms. Considerable effort has gone into the design and implementation of this compiler and core CLAM instruction set, so that the compiler can perform many local optimizations. The constraint solver has been specialized to take advantage of simple cases and employs a heirarchy of constraint solvers to reduce the cost of constraint solving.

The main factor limiting the quality of CLAM code generated by the existing compiler is the lack of global analysis information and subsequent inability to perform multi-variant code specialization. This consideration has lead to the development of the optimizing compiler described herein.

3 Optimization

In this section we describe the operation of the optimizer and detail the suite of six optimizations it uses by means of a worked example.

Consider the following CLP($\mathcal{R}$) program defining the relation `fib` where `fib(N,F)` holds if, and only if, `F` is the `N`'th Fibonacci number. The program is for illustrative purposes only and is a direct translation of the usual (computationally inefficient) mathematical definition of Fibonacci numbers.

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

Imagine that the compiler has been given this program in a file together with the declarations

```
:- export fib(N, F) where ground(N), free(F).
:- export fib(N, F) where true.
```

which tell the compiler to compile `fib(N,F)` for two modes of usage. In the first mode `fib` is called with `N` a fixed value, that is `N` is *ground*, and `F` is a variable which is unconstrained except that it may be aliased to other unconstrained variables, that is `F` is *free*. This mode of usage is intended to compute the `N`'th Fibonacci number. In the second mode, there are no restrictions on how it can be called. This mode is useful for computing the first `N` Fibonacci numbers, or checking if a number is in the Fibonacci sequence. We now detail in turn how the compiler will optimize `fib` for each of these modes.

Reordering of constraints. The addition of a constraint to the store can be moved to a later point in execution if the constraint does not affect the control flow in the intervening computation. Control flow is only affected if the constraint causes failure and hence backtracking. The advantage of reordering is that it reduces the size of the constraint store and also facilitates the other optimizations. Reordering requires determining possible interaction between constraints and hence possible unsatisfiability.

Consider the first mode of usage and the third rule in the definition. Initially `F` is free. This means that the constraint `F = F1+F2` is satisfiable no matter what values `F1` and `F2` take. Thus the constraint `F = F1+F2` cannot cause failure in the rule body and so can be moved to the end of the rule. Because `N1` and `N2` are new, free variables, the constraints `N1 = N-2` and `N2 = N-1` can also be moved to just before the first point in which constraints are placed on `N1` and `N2` respectively. The resulting program specialized for this usage is

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

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 program **(REO)** in the first mode of usage. By replacing constraints with tests and assignments we can in fact remove all calls to the constraint solver. This results in the following CLIC program. (Note that the CLIC instruction`>=`^{test} simply evaluates both sides and tests whether the `>=` relationship holds, while `=`^{assign} evaluates the right hand side and sets the value of the left hand side to this value. It is only because of reordering that it is possible to make `F = F1+F2` into an assignment.)

(BYP)

```
fib^{gf}(N, F) :- N =^{test} 0, F =^{assign} 1.
fib^{gf}(N, F) :- N =^{test} 1, F =^{assign} 1.
fib^{gf}(N, F) :- N >=^{test} 2,
                  N1 =^{assign} N - 2,
                  fib^{gf}(N1, F1),
                  N2 =^{assign} N - 1,
                  fib^{gf}(N2, F2),
                  F =^{assign} F1 + F2.
```

Splitting and dead code elimination.

In general the compiler will perform *multi-variant specialization* by copying rule definitions for different patterns of usage if this enables more optimizations to be performed. This is called "splitting" and will continue until some upper limit of the number of versions of a rule is reached. Multi-variant specialization is the process of applying optimizations on each split copy, or variant, in a different way from any other split variant. Whenever a defintion is split, bounds information can be used to eliminate those rules that cannot succeed. This is a form of dead code elimination.

The compiler will first apply the reordering and solver bypass strategies. In this case as `F` is not free, `F = F1+F2` can only be moved before the second recursive call to `fib`. However this movement means that `F1` in the first recursive call to `fib` is now free which gives rise to a new pattern of usage for `fib` in which the first argument is unrestricted and the second is free. Also, after returning from the first call to `fib`, `N1` is ground, and hence `N` is ground. Thus `N2 = N-1` is an assignment and on the second call to `fib` the first argument is ground, which is again a new pattern of usage. As both patterns of usage allow additional solver bypass optimizations, the compiler will "split" the definition of `fib` and perform multi-variant specialization.

(SPLIT)

```
fib(N, F) :- N = 0, F = 1.
fib(N, F) :- N = 1, F = 1.
fib(N, F) :- N >= 2,
             N1 = N - 2,
             fib^{af}(N1, F1),
             N2 =^{assign} N - 1,
             F = F1 + F2,
             fib^{ga}(N2, F2).
fib^{af}(N, F) :- N = 0, F =^{assign} 1.
fib^{af}(N, F) :- N = 1, F =^{assign} 1.
fib^{af}(N, F) :- N >= 2,
                  N1 = N - 2,
                  fib^{af}(N1, F1),
                  N2 =^{assign} N - 1,
                  fib^{gf}(N2, F2),
                  F =^{assign} F1 + F2.
fib^{ga}(N, F) :- N =^{test} 0, F = 1.
fib^{ga}(N, F) :- N =^{test} 1, F = 1.
fib^{ga}(N, F) :- N >=^{test} 2,
                  N1 =^{assign} N - 2,
                  fib^{gf}(N1, F1),
                  N2 =^{assign} N - 1,
                  F = F1 + F2,
                  fib^{ga}(N2, F2).
```

This example did not demonstrate dead code elimination. However, if there was a call to `fib` of the form `N >= 2, fib(N,F)` where `N` and `F` are unrestricted, then splitting would occur as above except that the first two rules for `fib` will be eliminated as they cannot ever succeed.

Future redundancy. A major source of inefficiency in the solver is caused by constraints which, after addition of other constraints to the solver, have become redundant in the sense that their information is implied by these other constraints in the current solver. Execution can be optimized by adding instructions which remove these constraints from the store, as this decreases the size of the constraint store but cannot change the behavior of the program. In many cases removal can occur even before the constraint becomes redundant as the constraint cannot cause failure between the time of removal and when it becomes redundant. An important case is when the constraint can be removed immediately after it is added to the solver. 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. This is called *future redundancy*.

Consider the constraint `N >= 2` in the recursive rule of $\texttt{fib}^{af}$ in **(SPLIT)**. In the call to $\texttt{fib}^{af}$`(N1,F1)` the first constraint encountered is either `N1 = 0`, `N1 = 1` or `N1 >= 2`. But `N1 = N-2`, so in each case the constraint `N >= 2` is made redundant. Hence it can be removed before the call to $\texttt{fib}^{af}$`(N1,F1)` without affecting execution. Thus `N >= 2` is future redundant. Thus the compiler will transform $\texttt{fib}^{af}$ in **(SPLIT)** into **(REM)** below.

(REM)

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

Application of future redundancy requires knowledge about possible interaction between constraints and also tests for unsatisfiability of constraints. In the above example the call to $\texttt{fib}^{af}$ is unfolded and the compiler checks that `N >= 2` is made redundant before subsequent atoms. More exactly the compiler checks that: `F = F1+F2` $\wedge$ `N1 = N-2` $\wedge$ `N1 = 0` implies `N >= 2`, `F = F1+F2` $\wedge$ `N1 = N-2` $\wedge$ `N1 = 1` implies `N >= 2`, and `F = F1+F2` $\wedge$ `N1 = N-2` $\wedge$ `N1 >= 2` implies `N >= 2`. This is tested by checking that the constraints `F = F1+F2` $\wedge$ `N1 = N-2` $\wedge$ `N1 = 0` $\wedge$ `N < 2`, `F = F1+F2` $\wedge$ `N1 = N-2` $\wedge$ `N1 = 1` $\wedge$ `N < 2`, and `F = F1+F2` $\wedge$ `N1 = N-2` $\wedge$ `N1 >= 2` $\wedge$ `N < 2` are unsatisfiable. This optimization is only applied to arithmetic inequalities.

In future work we plan to add arbitrary removal of constraints. However, our experience has been that future redundancy captures most cases for removal.

Removal of redundant variables. Another common source of redundancy in the constraint solver is caused by variables which will never be referred to again. Execution can be improved by adding instructions which eliminate 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 and is useful because it reduces the number of variables and constraints in the solver.

Consider the calls to $\texttt{fib}^{ga}$`(N2,F2)` in **(SPLIT)**. After each call the variable `F2` is never again referred to and so is dead on return from the call. Thus, in the call the variable `F2` which is locally referenced by `F` inside the call can be removed from the constraint solver. This gives:

(REV)

```
fib^ga(N, F) :- N =^test 0, F^rem = 1.
fib^ga(N, F) :- N =^test 1, F^rem = 1.
fib^ga(N, F) :- N >=^test 2,
                N1 =^assign N - 2,
                fib^gf(N1, F1),
                N2 =^assign N - 1,
                F^rem = F1 + F2,
                fib^ga(N2, F2).
```

No fail constraints. Sometimes, when a constraint is encountered, it can be guaranteed not to fail because of the presence of new variables. The existing solver detects this at runtime and uses this information to solve the constraint quickly, but there is still an overhead in detecting the possibility and manipulating the constraint into the required form. If the information about new variables is collected at compile-time, we can produce specialized CLIC instructions that reduce the overhead. This optimization speeds up constraint solving. For example `fib(N,F)` can be transformed to:

(NOF)

```
fib(N, F) :- N = 0, F = 1.
fib(N, F) :- N = 1, F = 1.
fib(N, F) :- add_remove(N >= 2),
             nofail(N1 = N - 2),
             fib^af(N1, F1),
             N2 =^assign N - 1,
             nofail(F2 = F - F1),
             fib^ga(N2, F2).
```

Thus we have seen that information about call patterns and multi-variant specialization has allowed us to dramatically improve the definition of `fib` even in the case that we make no assumptions about the mode of usage.

4 The Compiler

In the previous section we described the transformations and the basic operations of the optimizing compiler. In this section we describe in more detail the global analyzer, optimizer and code generator.

The Analyzer

To facilitate the rich variety of analyses required in the compiler, the analyzer is a generic tool like other analysis engines, such as PLAI [18] and GAIA [12] developed for Prolog. Generic analyses engines are not only designed to do many different analyses but also to allow for the easy addition of new analyses. The core of the analyzer is an algorithm for efficient fixpoint computation. Efficiency is obtained by keeping track of which parts of a program must be reexamined when a success pattern is updated. The analyses currently performed by our analyzer are: *Pos*, *CallAlive*, *Shar*, *Free*, *NonLin*, *Type* and *Bounds*. These analyses will be outlined below.

The analyzer performs global analysis of programs, that is, program information (descriptions) is inferred about the calls "between" different rules of the program as well as inferring the local information "within" each rule. The global analysis is based on abstract interpretation of logic programs and constraint logic programs [3, 14, 15] in which operations in the execution of the goal are mimicked by abstract operations on the domain of descriptions, (the analyses).

The analyzer has two novel features. First it handles "widening" which means that the analysis will terminate for description domains such as arithmetic intervals which have infinite ascending chains of descriptions. Second, the analyzer is incremental. Whenever the underlying program or goal is modified it is not reanalyzed from scratch. For example when a definition is split no analysis takes place as information about annotations on the new rules is already in the old annotated program. When rules are modified, then from correctness of the transformations, the analyzer knows that all success pattern information is still correct, meaning that in practice little additional analysis is needed. Details about the algorithms used can be found in [6].

Details of the description domains are deliberately kept insulated from the optimizer so as to make it easier to change these. The most difficult condition to analyze for is deadness. This is because not being textually alive may not mean a variable is dead, for two reasons. The first reason is "structure sharing" between terms. The second reason is that "hard" constraints such as non-linear equations are delayed by the constraint solver until they become simple enough to solve. This means that in the analyzer variables in hard constraints must be assumed to be alive until, using information about groundness, we can guarantee the constraint is simple enough to be solved in the linear constraint solver. Currently the analyzer uses descriptions which are tuples of seven different domains:

Pos This domain was illustrated in the above example and capture groundness information about variables. We use ROBDDs to represent the Boolean functions.

CallAlive This consists of lists of variables which may be directly referenced later in execution.

Shar This captures information about possible structure sharing of variables between Prolog terms. It is based on descriptions introduced by Søndergaard [20] for eliminating occur checks in Prolog. The description consists of a possible sharing relation for variables. Consider the goal `Y = f(X), p(X), Y = f(Z), q(Z)`. After `Y = f(X)`, `X` and `Y` possibly share, but `Z` does not share with anything else. After `Y = f(Z)`, all variables possibly share.

Free This consists of lists of variables which are free. It makes use of *Shar* to keep track of variable aliasing.

NonLin This consists of list of variables which are possibly contained in a delayed non-linear constraint.

Type This indicates whether a variable is definitely arithmetic or possibly involved in term constraints.

Bounds This gives an interval for each arithmetic variable in which the variable's value must lie. Widening is used with this domain to ensure termination.

The analyzer takes a program and goal and annotates each point in the program with an approximate description of the constraints which will be encountered at that point when the goal is executed. As an example consider the following Fibonacci program. If this program is analyzed for the class of calls in which the first argument is ground, that is bound to a number, then the following annotated program results. The description domain consists of Boolean functions which capture groundness information about variables and definite dependencies among variables [2]. For example the function `N ∧ (F ↔ F2)` indicates that the variable `N` is ground, and that if `F` is ever ground, then so is `F2` and vice versa. The column to the right gives the dependency description of the rule variables after the corresponding statement in the program.

Program	**Annotated Program Point**
`fib(N, F) :-`	`{N}`
`    N = 0,`	`{N}`
`    F = 1.`	`{N ∧ F}`
`fib(N, F) :-`	`{N}`
`    N = 1,`	`{N}`
`    F = 1.`	`{N ∧ F}`
`fib(N, F) :-`	`{N}`
`    N >= 2,`	`{N}`
`    N1 = N - 2,`	`{N ∧ N1}`
`    N2 = N - 1,`	`{N ∧ N1 ∧ N2}`
`    fib(N1, F1),`	`{N ∧ N1 ∧ N2 ∧ F1}`
`    F = F1 + F2,`	`{N ∧ N1 ∧ N2 ∧ F1 ∧ (F ↔ F2)}`
`    fib(N2, F2),`	`{N ∧ N1 ∧ N2 ∧ F1 ∧ F2 ∧ F}`

For example, initially, when the third rule is entered, `N` is ground. The statements `N >= 2` does not change this. After the statement `N1 = N-1`, as `N` is ground, `N1` becomes ground. Similarly, after `N2 = N-1`, `N2` becomes ground. The effect of the call `fib(N1,F1)` is to ground `F1`. The statement `F = F1+F2` adds the information that `F` is ground if and only if `F2` is. The effect of the call `fib(N2,F2)` is to ground `F2` and hence to ground `F`.

Actually the analyzer does not exist as a separate entity in the compiler. Rather it is associated with the *AnnotatedProgram* class of which the current annotated CLIC program is an instance. The optimizer obtains analysis information by way of methods on *AnnotatedPrograms*. These methods are divided into two groups.

The first group of five methods provides information for a given program point in some rule either for a single calling pattern for that rule, or for all such calling patterns. The methods respectively return: the list of *ground arithmetic* variables, the list of *free* variables, the list of variables which are *nofail* for a particular constraint in the sense that if this constraint is added to the solver then it cannot cause failure (they are free and not aliased to any other variable in the constraint), the list of variables which are *dead* in the sense that they will not be referenced directly or indirectly in the future after the next constraint, and for each arithmetic variable, the real interval it is constrained to lie in.

The second group of methods on the annotated CLIC program allows the optimizer to modify the CLIC program and associated goal. There are methods to split atom definitions for different call patterns, and methods which reorder and remove constraints in the body of the rule. There is also a method which requests the analyzer to annotate a hypothetical goal using the current atom definitions in the program. Such hypothetical reasoning is used to determine the applicability of reordering and removal and allows the optimizer to obtain answers to questions of the form, what happens if this constraint is removed.

The Optimizer

The optimizer examines the rules of the annotated CLIC program and where possible performs optimizations upon them. The optimizer performs the optimizations in two passes. In the first pass reordering is performed. In the second pass solver bypass, future redundancy, dead variables and nofail are performed.

In the second pass, if competing optimizations are available they are currently handled as follows: *solver bypass* is preferred to *nofail*, *add_remove* and *dead*. Thus for example a *dead* optimization will not be applied in cases where the dead variable is always ground. This decision is based on the effects of the optimizations in the run-time system. Removing a fixed dead variable will not increase the speed of future constraint solving, but will cause extra overhead if backtracking occurs. Using assignment is preferable to *nofail* and *add_remove* since assignment does not involve the solver at all.

In each pass the optimizer examines and optimizes each strongly connected component (SCC) in the program call graph in turn. SCCs are examined from the bottom up, so that predicates at the bottom of the call graph are optimized first. This decision implements the heuristic that optimizations at a lower level are likely to be more important than at a higher level since low level predicates are (in general) executed more frequently. In general, optimizations made for one predicate may prevent optimizations for other predicates.

When optimizing the rules in a single SCC, the optimizer first performs optimizations which are valid for all calling patterns of each predicate. It scans each rule examining each constraint for possible optimizations. Also, if an atom (defined in a lower SCC) is present in the rule which is always called using a calling pattern for which the optimizer has created a specialized version, then the atom is replaced by a call to the specialized version.

Next the calling pattern graph is examined. If there are multiple calling patterns for any predicates in the SCC of the call graph under consideration, the optimizer examines each SCC of the call graph in turn, again bottom up. If, for a particular calling pattern, new optimizations are available, either because a new constraint optimization is possible, or an atom may be replaced with a more specific version, the optimizer creates a new version of the predicate for this specific calling pattern.

Once the optimizer has determined that a predicate must be split, it constructs a new copy of the code — containing the optimizations already made for all calling patterns — and optimizes this code further (multi-variant specialization). Analysis information for the new code is extracted from the original. In general some reanalysis may be required. Splitting an atom definition may allow further optimization of rules in the calling pattern SCC which have already been optimized. Thus for maximum improvement we should continue the process of splitting and optimizing until a fixpoint is reached. However, for simplicity, the optimizer presently examines each calling pattern once only in each pass. In practice we have found this finds all possible optimizations.

The compiler determines the applicability of a particular optimization by querying the annotation information at program points to find variables which are free, ground arithmetic, dead, nofail and what are their bounds. Future redundancy also makes use of the constraint solver. The dependencies between the constraint solver and analysis domains, annotated program interface and the optimizations are detailed in Figure 3.

Some of the analyses are quite complex. For example, to reorder a *nofail* constraint the optimizer creates as a hypothetical goal the remainder of the rule after that constraint and asks for this goal to be annotated for the calling pattern occurring before the constraint. This mimics the behaviour of the remainder of the rule if the constraint were removed. By examining the annotations of this hypothetical goal the optimizer can determine the last point where the *nofail* constraint is still guaranteed not to

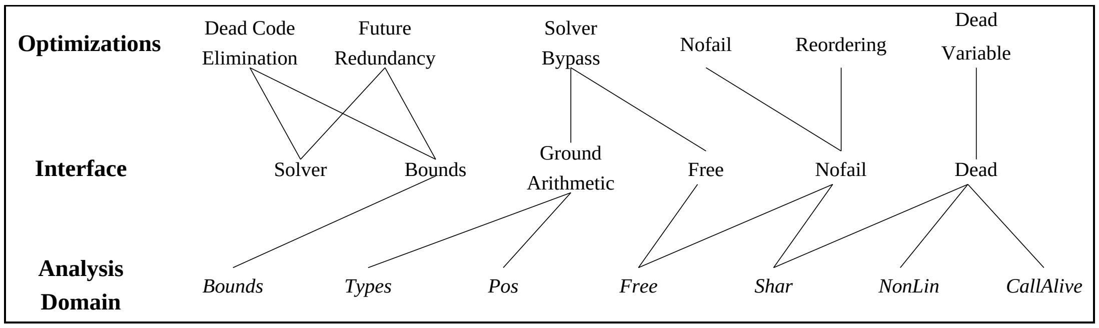

Figure 3: Relationships between analysis domains and optimizations

fail, and then move it to that position. The old rule is replaced by the new rule with the constraint moved and the analyzer updates the annotations. Future redundancy optimization is performed by examining an inequality and determining whether it is made redundant by the constraints occurring between it and the next atom A in the rule, together with the constraints occurring before any atoms in the rules for atom A and the inequalities determined by the bounds information. If the constraint is made redundant by each such combination of constraints (which is determined using the constraint solver) it is marked as *add_remove*.[1]

Code Generator

The code generator maps the CLIC code into CLAM [9] instructions which are executed by the CLAM emulator. The existing CLP($\mathcal{R}$) compiler also produced CLAM code, but used only a core set. The new compiler makes use of extended CLAM instructions for achieving the optimizations that are made possible by global analysis. The CLAM architecture is suited to this as it operates below the level of a constraint and the optimizations described all require modifying the operation of constraint solving to be effective and thus slot neatly with the rest of the CLAM architecture.

5 Empirical Results

To illustrate the effect of optimization, we show its effect on a number of different programs and goals.

The set of benchmark programs and goals is shown in Table 1.

Table 2 shows how long the optimizing compiler takes to fully optimize the benchmarks, that is when splitting is performed. Two times in seconds are given – the time for initial program analysis, and the time for optimizations plus any subsequent reanalyses used in optimization. The table also gives some of the characteristics of the benchmark programs. It details the number of rules in each program, the number of literals and the maximum number of variables in a rule in the program.

Table 3 shows the effect of the optimizations, comparing the optimized code with and without splitting versus unoptimized code. If no splitting occurred when it was enabled the columns are marked —. Execution times are in CPU seconds using a modified version of the existing compiler, (CLP($\mathcal{R}$) v1.2), on a Sparc 1000. The speedup ratios are given with respect to the original unoptimized program. The space originally used by the constraint solver (measured in solver nodes = 6 words) is given together with the percentage used by the optimized programs.

We see substantial across-the-board improvements in time and space for the benchmarks. Our results also show the advantage of multi-variant specialization as this gives rise to considerably larger speed ups and some space savings while it does not lead to a large increase in code size, even with the current naive splitting strategy. Speedups for some of the larger benchmarks `pic` and `neural` are not as great as for the smaller benchmarks. This is because information is lost in the analysis due to

[1]Note that this strategy can sometimes delay failure, and hence is not guaranteed to always improve performance, however this behaviour is very rare and in fact it does not occur in any of the benchmark programs.

`fib`-*forw*	Fibonacci program with `N` ground and `F` free
`fib`-*back*	Fibonacci program with `N` free and `F` ground
`ack`	Computes Ackerman's function
`mg`	First example goal for mortgage program.
`sumlist`-*gnd*	Adds up numbers from a list
`sumlist`-*var*	Adds up variables from a list (600 elements), then grounds them
`matmul`	Finding the inverse of a matrix
`mg-extend`	Handles complex mortgages with provisions for a number of payments per month and special initial conditions and provisos
`pic`	A picture specification language
`circuit`	Performs simple circuit analysis
`bridge`	Generates a finite element model of a bridge
`neural`	A neural net training program
`amp`	Amplifier design and analysis

Table 1: Benchmark descriptions

	Rules	Literals	Max vars in a rule	Analysis (sec)	Optimization (sec)
`fib`-*forw*	4	13	6	0.36	0.44
`fib`-*back*	4	15	6	0.33	1.36
`ack`	4	23	7	0.36	0.75
`mg`	3	12	8	0.26	0.23
`sumlist`-*gnd*	6	18	5	0.36	0.21
`sumlist`-*var*	9	23	5	0.34	0.18
`matmul`	13	46	10	0.66	0.90
`mg-extend`	12	49	15	0.84	4.14
`pic`	10	67	24	0.56	0.37
`circuit`	16	61	13	0.80	1.05
`bridge`	19	96	20	0.89	3.23
`neural`	75	240	14	1.62	3.71
`amp`	62	326	30	6.15	11.57

Table 2: Time for compilation

Query	Original	Without splitting		With splitting		
	space use	Speedup	Space (%)	Literals	Speedup	Space (%)
`fib`-*forw*	62608	3.2	41	23 (1.8)	5.0	33
`fib`-*back*	33665	2.9	77	41 (2.7)	4.9	62
`ack`	131418	12.9	6	— (1.0)	—	—
`mg`	189	1.8	100	18 (1.5)	1.8	100
`sumlist`-*gnd*	66	2.3	100	18 (1.0)	4.6	100
`sumlist`-*var*	366004	1.0	100	28 (1.2)	45.0	1.3
`matmul`	60840	1.4	100	68 (1.5)	1.4	61
`mg-extend`	27150	1.6	61	83 (1.7)	2.0	61
`pic`	22013	1.1	100	73 (1.1)	1.1	90
`circuit`	1346	1.4	68	66 (1.1)	1.4	66
`bridge`	1345	1.7	100	184 (1.9)	1.8	100
`neural`	4661	1.1	88	338 (1.4)	1.1	88
`amp`	2117	1.5	77	— (1.0)	—	—

Table 3: Impact of optimizations (time ratios, space in solver nodes)

arithmetic variables being passed around in data structures. This loss of accuracy is currently being addressed. None the less, the speedups are impressive when one considers the maturity of the compiler (CLP($\mathcal{R}$) v1.2) that we are comparing against. Indeed the speedup due to global analysis and the resultant simplification of constraint solving is even larger than the speedup resulting from moving from the original CLP($\mathcal{R}$) interpreter to the compiler.

6 Conclusion

The highly optimizing compiler presented in this paper is an implementation of the major components of the compiler proposed in [16]. The results demonstrate that global analysis and optimization can significantly improve the performance of CLP languages. Many of the techniques applied herein we expect will prove useful for other CLP languages, especially reordering of constraints and removal of redundant variables.

Considerable space and time savings have been obtained on our benchmark programs. We expect to improve these results significantly in the near future. In addition, our larger benchmarks have a significant proportion of Prolog terms, thereby limiting the impact of our optimizations. We expect the inclusion of optimizations for the Prolog part of the programs to improve performance. For some Prolog optimizations further analysis will be unnecessary as our analyzer already computes *pos*, *free* and *sharing* information about Prolog variables. This analysis information has previously been shown to be useful for the optimization of Prolog programs.

We plan to extend the compiler presented herein to provide new optimizations. In the near future these include transforming mutually exclusive rules into if-then-else statements, if the arithmetic test are mutually exclusive. This optimization is the analog of indexing instructions for mutually exclusive Prolog terms. We also plan to provide unboxing of solver variables in calls to rule definitions and to improve the handling of splitting to generate all appropriate versions and collapse versions when they are indistinguishable.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* MIT Press, 1991.

[2] T. Armstrong, K. Marriott, P. Schachte and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Static Analysis: Proc. First Int. Symp.* (Lecture Notes in Computer Science 864), 266–280. Springer-Verlag, 1994.

[3] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming* **13** (2&3): 103–179, 1992.

[4] J. Darlington, Y. Guo and H. Pull. A new perspective on integrating functional and logic languages. *Proc. Fifth Generation Computer Systems*, 682–693. Tokyo, 1992.

[5] B. Freeman-Benson and A. Borning. The design and implementation of Kaleidoscope'90: A constraint imperative programming language. *Proc. IEEE Int. Conf. Computer Languages*, 174–180. IEEE Computer Soc. Press, 1992.

[6] M. Hermenegildo, G. Peubla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *Proc. of the Int. Conf. on Logic Programming*, Tokyo, Japan, MIT Press, June 1995, 797–814.

[7] P.C. Kanellakis, G.M. Kuper and P. Revesz. Constraint query languages. *Proc. ACM Symp. Principles of Database Systems*, 299–313, ACM Press, 1990.

[8] A. Kelly, A. Macdonald, K. Marriott, H. Sondergaard, P. Stuckey, and R. Yap. An Optimizing Compiler for CLP($\mathcal{R}$). In *Proc of the First Int. Conf. on Principles and Practices of Constraint Programming*, LNCS 976, Springer-Verlag, 222–239, 1995.

[9] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap. An abstract machine for CLP($\mathcal{R}$). *Proc. ACM Conf. Programming Language Design and Implementation*, 128–139. ACM Press, 1992.

[10] J. Jaffar, S. Michaylov, P. Stuckey and R. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems* **14** (3): 339–395, 1992.

[11] N. Jørgensen, K. Marriott and S. Michaylov. Some global compile-time optimizations for CLP($\mathcal{R}$). In V. Saraswat and K. Ueda, editors, *Logic Programming: Proc. 1991 Int. Symp.*, 420–434. MIT Press, 1991.

[12] B. Le Charlier and P. Van Hentenryck. Experimental evaluation of a generic abstract interpretation algorithm for Prolog. *ACM Transactions on Programming Languages and Systems* **16** (1): 35–101, 1994.

[13] A. Macdonald, P. Stuckey and R. Yap. Redundancy of variables in CLP($\mathcal{R}$). In *Logic Programming: Proc. 1993 Int. Symp.*, 75–93. MIT Press, 1993.

[14] K. Marriott and H. Søndergaard. Analysis of constraint logic programs. In S. Debray and M. Hermenegildo, *Logic Programming: Proc. North American Conf. 1990*, 531–547. MIT Press, 1990.

[15] K. Marriott, H. Søndergaard and N. D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems* **16** (3): 607–648, 1994.

[16] K. Marriott, H. Søndergaard, P. Stuckey and R. Yap. Optimizing compilation for CLP($\mathcal{R}$). In G. Gupta, editor, Proc. Seventeenth Australian Computer Science Conf., *Australian Computer Science Comm.* **16** (1): 551–560, 1994.

[17] K. Marriott and P. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. *Proc. Twentieth ACM Symp. Principles of Programming Languages*, 334–344. ACM Press, 1993.

[18] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming* **13** (2&3): 315–347, 1992.

[19] V. Saraswat. *Concurrent Constraint Programming Languages.* MIT Press, 1993.

[20] H. Søndergaard. An application of abstract interpretation of logic programs: Occur check reduction. In B. Robinet and R. Wilhelm, editors, *Proc. ESOP 86* (Lecture Notes in Computer Science 213), 327–338. Springer-Verlag, 1986.