Source to Source Optimizations of CLP($\Re_{Lin}$)

Viswanath Ramachandran and
Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-30
October 1995

Source to Source Optimizations of CLP($\Re_{Lin}$)

Viswanath Ramachandran and Pascal Van Hentenryck
Brown University, Box 1910, Providence, RI 02912 (USA)
{vr,pvh}@cs.brown.edu

Abstract

This paper describes the design and implementation of an optimizing compiler for CLP($\Re_{Lin}$), a constraint logic programming language over linear real constraints. The compiler performs a number of source to source optimizations which aim at replacing constraint solving, the basic operation of the language, by assignments (when, at runtime, all but one variables have a fixed value) and tests (when, at runtime, all variables have a fixed value) These optimizations follow the 3R's methodology which consists of refining, reordering, and removing constraints. The compiler uses abstract interpretation to decide when and how to perform the optimizations. The resulting system (40,000 lines of C, 10,000 of which concern the optimizations) has been evaluated experimentally. Our preliminary results indicate that substantial (often asymptotic) speedups can be obtained. Most of these speedups can be made arbitrary large, since they depend on the input size. This is, to our knowledge, the first operational compiler for CLP($\Re_{Lin}$) with these functionalities and the first implementation of reordering and constraint removal. It is also a convincing demonstration that abstract interpretation can be used to produce dramatic speedups for CLP($\Re_{Lin}$) programs.

1 Introduction

Constraint logic programming (CLP) [9] is a generalization of logic programming where unification is replaced by constraint solving as the basic operation of the language. Constraint solving makes it possible to preserve some traditional advantages of logic programming (e.g., the logical variable and reversibility of programs) even when arithmetic operations are involved. In addition, the combination of constraint solving and nondeterminism (approximated by backtracking) makes these languages appealing for a variety of combinatorial search problems. CLP languages have been applied to a variety of combinatorial search applications in areas.

Constraint solving however is an expensive operation in general and it is natural for a compiler to aim at minimizing its use by replacing constraints by assignments and tests. An interesting methodology to exploit this basic observation was proposed by Marriott and Stuckey [15] for CLP($\Re_{Lin}$), a constraint logic programming language over linear constraints. The key idea behind the methodology consists of refining constraints into tests and assignments, removing redundant constraints (i.e., constraints which are implied by the constraint store), and reordering constraints to maximize refinements and removals. Reordering is the most delicate optimization, since the compiler must make sure that the resulting program preserves the same search tree for the programs. This in turn guarantees the termination of the resulting program and makes sure that the "optimized" program can never be significantly slower than the unoptimized program.

The purpose of this paper is to present the design and implementation of the first optimizing compiler for CLP($\Re_{Lin}$) implementing this methodology. The compiler, which includes about 10,000 lines only for the optimizations, uses abstract interpretation to collect the information necessary to

```
p(X,Y,X) :-          q(X,Y,Z) :-          r(X,Y) :-
   X ≥ Z + 3,           r(X,Y).              X ≤ Y + 2.
   Y ≤ Z,            q(X,Y,Z) :-
   q(X,Y,Z).            Z ≥ Y + 2.
```

Figure 1: A simple CLP($\Re_{Lin}$) program.

perform the optimizations safely. In particular, it uses the generic abstract interpretation algorithm GAIA [11] instantiated to three abstract domains: Prop [13, 20] to deduce information on fixed variables, RatOrder [1] to remove redundant constraints, and LSign [16, 19] to reorder constraints and to determine unconstrained variables. The preliminary experimental results indicate that substantial speedups can be obtained. In particular, on most of the benchmarks, the speedups are asymptotic, i.e., they depend on the size of the inputs.

The main technical contribution of this paper is to present the first implementation of reordering and constraint removal for CLP($\Re_{Lin}$) programs, to report the first implementation of the domain LSign, and to present the first implementation of a compiler integrating all these source to source transformations. The main conceptual contribution of the paper is to show that sophisticated static analyses and source to source transformations can produce dramatic (often asymptotic) speedups for CLP($\Re_{Lin}$) programs. Much research has been devoted to abstract interpretation of CLP($\Re_{Lin}$) (e.g., [6, 7, 14, 16, 19]) but almost no experimental results have appeared to quantify the possible benefits. The only exception we are aware of is the system described in [10] but no reordering or constraint removal is performed in the system. As we show in this paper, reordering is probably the most fundamental optimization.

The rest of the paper is organized as follows. Section 2 gives a brief overview of CLP($\Re_{Lin}$). Section 3 gives an overview of the optimizations by means of a complete example. Section 4 presents the analyses used by the system, Section 5 describes the transformations, and Section 6 describes some additional functionality. Section 7 reports our preliminary experimental results, while Section 8 concludes the paper.

2 Overview of CLP($\Re_{Lin}$)

In this section, we give an informal overview of CLP($\Re_{Lin}$). The reader interested in more details can refer to [8, 21]. CLP($\Re_{Lin}$) programs have essentially the same syntax as Prolog, the main difference being that CLP($\Re_{Lin}$) programs also allow linear constraints over real numbers to appear in the bodies of clauses.[1] In this article, variables are denoted by uppercase letters, constraints by the letter c and conjunctions of constraints by the letter θ.

The operational semantics of the CLP scheme is a simple generalization of the semantics of logic programming, at least from a conceptual standpoint. It can be described as a goal-directed derivation procedure from the initial goal using the program clauses. A *computation state* is best described by *a goal part* (i.e., the conjunction of goals to be solved) and *a constraint store* (i.e., the set of constraints accumulated so far). Initially the constraint store is empty, and the goal part is

[1] The statements of the program may include nonlinear constraints but these constraints are assumed to become linear at runtime. We assume for simplicity that an error occurs otherwise.

the initial goal. In the following, we denote computation states by pairs $\langle\, G \,\Box\, \theta \,\rangle$, where G is the goal part and θ the constraint store. We use ϵ to denote an empty goal part or constraint store. To illustrate the operational semantics of CLP($\Re_{Lin}$), we use the simple program depicted in Figure 1. An example computation state occurring in this program is

$$\langle\, \texttt{q(X,Y,Z)} \,\Box\, \texttt{X}{\geq}\texttt{Z}{+}3 \;\&\; \texttt{Y}{\leq}\texttt{Z} \,\rangle.$$

A *computation step* (i.e., the transition from one computation state to another) can be of two types depending upon the selection of an atom or of a constraint in the goal part. In the first case, a computation step amounts to (1) selecting an atom in the goal part; (2) finding a clause that can be used to resolve the atom; this clause must have the same predicate symbol as the atom, and the equality constraints between the goal and head arguments must be consistent with the constraint store; (3) defining the new computation state as the old one where (a) the selected atom has been replaced by the body of the clause and (b) the equality constraints have been added to the constraint store. In the second case, a computation step amounts to (1) selecting a constraint in the goal part that can be satisfied with the constraint store; (2) defining the new computation state as the old one where the selected constraint has been removed from the goal part and added to the constraint store. For instance, given a computation state

$$\langle\, \texttt{q(X,Y,Z)} \,\Box\, \texttt{X}{\geq}\texttt{Z}{+}3 \;\&\; \texttt{Y}{\leq}\texttt{Z} \,\rangle$$

a computation step can be performed using clause (2) of `q` to obtain a new computation state

$$\langle\, \texttt{Z}{\geq}\texttt{Y}{+}2 \,\Box\, \texttt{X}{\geq}\texttt{Z}{+}3 \;\&\; \texttt{Y}{\leq}\texttt{Z} \,\rangle.$$

Another computation step leads to the configuration

$$\langle\, \epsilon \,\Box\, \texttt{X}{\geq}\texttt{Z}{+}3 \;\&\; \texttt{Y}{\leq}\texttt{Z} \;\&\; \texttt{Z}{\geq}\texttt{Y}{+}2 \,\rangle,$$

since the resulting constraint store is satisfiable.

A computation state is *terminal* if the goal part is empty or no clause can be applied to the selected atom to produce a new computation state or if the selected constraint cannot be satisfied with the constraint store. A *computation* is simply a sequence of computation steps that either ends in a terminal computation state or diverges. A finite computation is *successful* if the final computation state has an empty goal, and *fails* otherwise.

Note that the results of the computation are the constraint stores of the successful computations. Also, nothing has been said so far on the strategy used to explore the space of computations. Most CLP languages use a computation model similar to Prolog: atoms are selected from left to right in the clauses; clauses are tried in textual order; and the search space is explored in a depth-first manner with chronological backtracking in case of failures. For instance, on the simple program, a CLP language typically uses clause (1) for `p`, then clause (1) for `q`, and finally encounters a failure when trying to solve `r`. Execution then backtracks to clause (2) of `q`, giving a successful computation.

3 Overview of the Compiler Optimizations

Our compiler performs high-level optimizations, transforming the source program into another source program which may contain, not only constraints, but also assignments and tests as basic operations. An assignment is of the form `Var := Exp` which, when executed, assumes that `Var` is an unconstrained variable while `Exp` is a fixed expression (i.e., an expression whose variables are fixed to a value). A test is of the form `Exp1 ?> Exp2`, `Exp1 ?≥ Exp2`, `Exp1 ?= Exp2`, `Exp1 ?< Exp2`, and `Exp1 ?≤ Exp2`, where `Exp1` and `Exp2` are fixed expressions. The compiler is organized in four main phases: normalizing, reordering, removal, and refinement.

To illustrate the various phases, consider the mortgage program `mg(P,T,R,B)` relating a loan's principal (P) and a time period (T) with the monthly repayment (R) and the final balance (B) as follows:[2]

```
mg(P,0,R,P*1.01-R).
mg(P,T,R,B) :- T > 0, P ≥ 0, mg(P*1.01 - R,T-1,R,B).
```

We illustrate the optimizations when `mg` is used with `P` and `R` fixed (i.e., they are constrained to take a value) and `T` and `B` are unconstrained. Note that the various uses of a predicate in a program can be obtained automatically by abstract interpretation[3] and that several versions of the program can be generated when the predicate is used in multiple ways.

The first phase of our compiler consists of normalizing the program to make constraints explicit in order to ease analysis and optimization. On our running example, this phase produces the program

```
mg(P,T,R,B) :- T = 0, B = P*1.01 - R.
mg(P,T,R,B) :- T > 0, P ≥ 0, P1 = P*1.01 - R, T1 = T - 1, mg(P1,T1,R,B).
```

The second phase of our compiler tries to move constraints to a place in the clause where, roughly speaking, they can be specialized into tests or into assignments. More precisely, an inequality is moved to a place where all its variables are fixed at runtime, while an equation is moved to a place where all its variables or all its variables but one are fixed at runtime. In reordering goals in a clause, the compiler should make sure that the search space explored by the program is preserved in order to guarantee termination and to avoid significant slowdowns.[4] On our running example, our compiler postpones `T > 0` in the second clause until after the recursive call. Informally speaking, this is possible due to the fact that `T > 0` is always consistent with the input constraint store for `mg` and that `T1` is necessarily positive on return of the recursive call. Hence, `T > 0` cannot prune the search space. Our compiler proves this formally by the `LSign` analysis. Note also that, when postponed until after the recursive call, `T > 0` may be specialized into a test, since `T` is fixed. Similarly, `T1 = T - 1` can be moved until after the recursive call, since `T` is now unconstrained before the recursive call and hence it cannot prune the search space. The resulting program becomes:

```
mg(P,T,R,B) :- T = 0, B = P*1.01 - R.
mg(P,T,R,B) :- P ≥ 0, P1 = P*1.01 - R, mg(P1,T1,R,B), T = T1 + 1, T > 0.
```

[2]The program does not contain the interest rate as a parameter in order to keep the presentation of the paper as simple as possible. Including it as a parameter would complicate the analysis results presented later due to the presence of the nonlinear constraint. Our experimental results are expressed in terms of the general program.

[3]This assumes of course that users specify the top-level goal of the program and an input pattern which, roughly speaking, specifies which arguments are results and which are data.

[4]It is extremely difficult to guarantee that the "optimized" program is at least as efficient as the unoptimized program, since the constraint solver (based on the simplex algorithm) may be sensitive to the ordering of the constraints. No case of slowdowns was observed on our benchmarks however.

It is important to note that there may be several places where a constraint can be moved. Our compiler uses simple heuristics at this point: it chooses the first place where the constraints can be specialized. Future work will investigate various other options.

The third phase of the compiler consists of detecting redundant constraints (i.e., constraints implied by the constraint store each time they are selected). These constraints can be removed, since they do not add information to the constraint store. In our running example, this is the case of `T > 0` after reordering, since, informally speaking, the second argument is assigned to zero in the first clause and incremented by one in the recursive clause.[5] Once again, this fact can be proven formally through abstract interpretation. The resulting program is as follows:

```
mg(P,T,R,B) :- T = 0, B = P*1.01 - R.
mg(P,T,R,B) :- P ≥ 0, P1 = P*1.01 - R, mg(P1,T1,R,B), T = T1 + 1.
```

The fourth phase of the compiler specializes constraints into tests and assignments whenever possible. A constraint can be specialized into a test if the compiler shows that, at runtime, all its variables are fixed. An equation `V = Exp` can be transformed into an assignment if the compiler shows that, at runtime, `V` is unconstrained and `Exp` is a fixed expression. In our running example, it can be shown that, after reordering and removal, `T` and `B` are unconstrained in all calls to `mg`. As a consequence, the constraints in the first clause becomes assignments, while the second clause has two assignments and a test.

```
mg(P,T,R,B) :- T := 0, B := P*1.01 - R.
mg(P,T,R,B) :- P ?≥ 0, P1 := P*1.01 - R, mg(P1,T1,R,B), T := T1 + 1.
```

It is interesting to observe at this point that the resulting program does not invoke the constraint solver. It is essentially a Prolog program enhanced with a rational arithmetic component. As a consequence, traditional Prolog transformations and optimizations can now be applied. For instance, in our running example, the techniques of [5] can be used to transform our final program into a tail-recursive program. Similarly, efficient instructions can be generated for the tests and assignments. These additional optimizations are orthogonal to the contents of this paper and are not discussed here.

4 Program Analysis

Our compiler performs a series of analyses to infer the runtime properties necessary to carry out the optimizations. It uses abstract interpretation [3], a systematic method to develop static analyses. In particular, the compiler uses the abstract interpretation system `GAIA` [11] on three different domains: `LSign`, `RatOrder`, and `Prop`. This section contains an informal review of these components.

4.1 The Abstract Interpretation Framework

The abstract interpretation framework used by the compiler is a natural extension to `CLP` of our logic programming framework [11]. It follows the traditional approach to abstract interpretation [3].

[5] Note that this constraint is useful for other uses of the program.

As is traditional in abstract interpretation, the starting point of the analysis is a collecting semantics for the programming language. Our concrete semantics is a collecting fixpoint semantics which captures the top-down execution of constraint logic programs using a left-to-right computation rule and which ignores the clause selection rule. The semantics manipulates sets of constraint stores, i.e. multisets of linear constraints. Two main operations are performed on constraint stores: addition of a constraint to a constraint store and variable elimination. The semantics associates with each predicate symbol p in the program a set of tuples of the form $(\Theta_{in}, p, \Theta_{out})$ which can be interpreted as follows:

> "the execution of $p(x_1, \ldots, x_n) \ \wedge \ \theta$ with $\theta \in \Theta_{in}$ produces a set of constraint stores $\{\theta_1, \ldots, \theta_n, \ldots\}$, all of which belong to Θ_{out}."

The second step of the methodology is the abstraction of the concrete semantics. Our abstract semantics consists in abstracting a set of constraint stores by a single abstract store, i.e. an abstract store represents a set of constraint stores. As a consequence, the abstract semantics associates with each predicate symbol p a set of tuples of the form $(\beta_{in}, p, \beta_{out})$ which can be read informally as follows:

> "the execution of $p(x_1, \ldots, x_n) \ \wedge \ \theta$ with θ satisfying the property described by β_{in} produces a set of constraint stores $\theta_1, \ldots, \theta_n, \ldots$, all of which satisfy the property β_{out}."

β_{in} is called an input store of p while β_{out} is called an output store. In our approach, the link between the abstract and the concrete domain is given by a monotone concretization function. The abstract semantics assumes a number of operations on abstract stores, in particular addition of a constraint, projection, and upper bound. The first two operations are consistent approximations of the corresponding concrete operations. The upper bound operation is a consistent abstraction of union of sets of constraint stores.

The last step of the methodology consists of computing the least fixpoint or a postfixpoint of the abstract semantics using an algorithm such as `GAIA` [12] or `PLAI` [18]. Both of these are top-down algorithms computing a small, but sufficient, subset of least fixpoint (or of a postfixpoint) necessary to answer a user query.

¿From the abstract interpretation results, our current optimizing compiler uses basically two pieces of information for each predicate `p` in the program. First, it uses the output store β_{out} describing the constraint stores obtained by running `p` on an empty constraint store. This output store is called the *output description* of `p`. Second, it collects all the input abstract stores β_{in} encountered when analyzing the program for the query. These abstract stores, which describe all possible constraint stores than can be encountered when `p` is called, are summarized into a single abstract store that we call the *input description* of `p`. This input description of course characterizes all input stores for `p` encountered at runtime. The fact that these two descriptions suffice to produce good results comes from the nature of the abstract domains.

4.2 The Domain `LSign`

The domain `LSign` [16, 19] is fundamental for the reordering phase and for detecting unconstrained variables in the refinement phase. Its two critical ideas are the replacement of coefficients by their signs and the association of multiplicity information with constraints. A sign is an element of $\{0, \oplus, \ominus, \top\}$, where $\oplus$ denotes the (strictly) positive real numbers, $\ominus$ denotes the (strictly) negative real numbers, 0 denotes zero, and $\top$ denotes all real numbers. An abstract constraint is an expression of the form $s_0 \ op \ \sum_{i=1}^{n} s_i x_i$ where s_i is a sign and op is a relational operator (e.g., $\geq$). An abstract constraint denotes the set of constraints obtained by replacing the signs by coefficients in their denotations.

Example 1 The abstract constraint $\oplus = \ominus x_1 + \top x_2$ represents both the constraint $3 = -x_1 + x_2$ and $3 = -x_1 - x_2$ but not the constraint $3 = x_1 - x_2$.

The second key concept is the notion of an abstract constraint with multiplicity which represents a multiset of constraints. The multiplicity information specifies the size of the multiset. We consider three multiplicities, *One*, *ZeroOrOne*, and *Any*, which are used respectively to represent a multiset of size 1, a multiset of size 0 or 1, or a multiset of arbitrary size. An abstract constraint with multiplicity is the association of an abstract constraint and a multiplicity. It denotes sets of abstract stores (i.e., multiset of constraints).

Example 2 The abstract constraint with multiplicity $\langle \oplus = \ominus x_1 + \top x_2, One \rangle$ represents only multisets of size 1, e.g., $\{3 = -x_1 + x_2\}$. $\langle \oplus = \ominus x_1 + \top x_2, Any \rangle$ represents multisets of any size, e.g., $\emptyset$, $\{3 = -x_1 + x_2\}$ and $\{3 = -x_1 + x_2, 3 = -x_1 - x_2\}$.

An abstract store is a set of abstract constraints with multiplicities. An abstract store obviously denotes a set of constraint stores. An `LSign` description is a finite set of abstract stores. Note that it would be sufficient to use abstract stores as `LSign` descriptions but the use of their powerset as abstract domain enhances precision significantly in many examples.

Example 3 The abstract store $\{\langle \oplus = \ominus x_1 + \top x_2, One \rangle \ , \ \langle \oplus = \ominus x_1 + \oplus x_2, Any \rangle\}$ represents constraint stores with at least one constraint, e.g., $\{3 = -x_1 + x_2 \ , \ 2 = -x_1 + 3x_2\}$.

Note that abstract operations on this domain are non-trivial and use abstract versions of Fourier and Gaussian elimination. See [16, 19] for the details. We now illustrate the domain `LSign` on our running example. The top-level query, where `P` and `R` are fixed, is associated with the abstract store

```
{⟨ ⊤ = ⊕P , One ⟩,⟨ ⊤ = ⊕R , One ⟩}
```

stating that `P` and `R` are fixed to a real number. The output description for program `mg` is as follows:

```
{
   {⟨ 0 = ⊕T , One ⟩,⟨ 0 = ⊕P + ⊖R + ⊖B , One ⟩},
   {⟨ 0 < ⊕T , One ⟩,⟨ 0 ≤ ⊕P , One ⟩,⟨ ⊕ = ⊕T , One ⟩,⟨ 0 = ⊕P + ⊖R + ⊖B , One ⟩},
   {⟨ 0 < ⊕T , One ⟩,⟨ 0 ≤ ⊕P , One ⟩,⟨ ⊕ = ⊕T , One ⟩,⟨ 0 ≤ ⊕P + ⊖R , One ⟩,
    ⟨ 0 ≤ ⊕P + ⊖R , Any ⟩,⟨ 0 = ⊕P + ⊖R + ⊖B , One ⟩}
}
```

The first abstract store represents the results of the first clause. It requires the time period to be zero and it imposes the constraint linking the remaining variables. The second abstract store is the result of the second clause, when the recursive call uses the first clause once. The third abstract store is the result of the second clause when the recursive call uses the second clause at least once (and finitely often) and the first clause once. The input description for `mg`, which captures all constraint stores used as input for `mg`, is as follows:

```
{
   {⟨ ⊤ = ⊕P , One ⟩,⟨ ⊤ = ⊕R , One ⟩},
   {⟨ ⊤ = ⊕R , One ⟩,⟨ ⊖ < ⊕T , One ⟩,⟨ ⊤ = ⊕P + ⊕R , One ⟩},
   {⟨ ⊤ = ⊕R , One ⟩,⟨ ⊖ < ⊕T , One ⟩,⟨ ⊤ ≤ ⊖R , One ⟩,⟨ ⊖ < ⊕T , Any ⟩,
    ⟨ ⊤ ≤ ⊖R , Any ⟩,⟨ ⊤ = ⊕P + ⊕R , One ⟩}
}
```

It contains of course the query but also many other abstract stores which characterize constraint store occurring at runtime as input to `mg`. For instance, the second abstract store captures the stores encountered for the first recursive call to `mg`.

4.3 The Domain `Prop`

The domain `Prop` [14, 20] is an effective domain to compute groundness information for Prolog. It can be extended easily to infer fixed variables in `CLP` (e.g. [7]). Its key idea is to represent the information through a Boolean formula. Informally speaking, a formula $x \leftrightarrow y$ means that, whenever x is fixed, y is fixed and vice-versa. A formula $x \wedge y \rightarrow z$ means that, whenever x and y are fixed, z is fixed as well. An equation $x = y$ is abstracted by a formula $x \leftrightarrow y$, while an equation $a_0 = a_1 x_1 + \ldots + a_n x_n$ is abstracted by a set of formula

$$
\begin{array}{l}
x_1 \wedge \ldots \wedge x_{n-1} \rightarrow x_n, \\
x_1 \wedge \ldots \wedge x_{n-2} \wedge x_n \rightarrow x_{n-1}, \\
\ldots \\
x_2 \wedge \ldots \wedge x_n \rightarrow x_1.
\end{array}
$$

Inequalities are abstracted by 1, i.e., they are not used to infer fixed variables. We now illustrate the domain `Prop` on our running example. The output description for `mg` is the Boolean formula

$$\mathtt{T} \wedge (\mathtt{R} \leftarrow \mathtt{P} \wedge \mathtt{B}) \wedge (\mathtt{P} \leftarrow \mathtt{R} \wedge \mathtt{B}) \wedge (\mathtt{B} \leftarrow \mathtt{P} \wedge \mathtt{R})$$

It states that `T` has a fixed value and specifies the relationship between the other variables. The input description for `mg` is the Boolean formula

$$\mathtt{P} \wedge \mathtt{R}$$

stating that all calls to `mg` have fixed values for `P` and `R`.

4.4 The Domain `RatOrder`

The domain `RatOrder` is used for the detection of redundant constraints. It approximates information about arithmetic relationships by rational order constraints, i.e. binary constraints of the form $x\ \delta\ y$ and unary constraints of the form $x\ \delta\ c$, where x, y are variables, $\delta \in \{>, \geq, =, \leq, <\}$, and c is a real number. For instance, a constraint $x \geq y + 2$ is approximated by a constraint $x > y$ in this domain. Consider the program

```
mg(P,T,R,B) :- T = 0, B = P*1.01 - R.
mg(P,T,R,B) :- P ≥ 0, P1 = P*1.01 - R, mg(P1,T1,R,B), T = T1 + 1, T > 0.
```

`GAIA` on this domain produces the output description $\{\mathtt{T1} \geq 0\ \&\ \mathtt{T} > \mathtt{T1}\}$ for the program point just before constraint `T > 0`. From this information, it is easy to conclude that `T > 0` is redundant.

5 Program Transformations

In this section, we discuss in more detail how the compiler performs the optimizations given the result of the analysis. Figure 2 depicts the overall organization of the compiler.

5.1 Reordering

Reordering is the most complicated phase of the compiler. This phase first performs `LSign` and `Prop` analyses on the normalized program. The `Prop` analysis is necessary to identify where to move constraints, while the `LSign` analysis is needed to determine if the move is acceptable.[6] To understand when a reordering is possible, it is interesting to reconsider the operational semantics. Consider a clause

p `:-` $g_1, \ldots, g_i, c, g_{i+1}, \ldots$

and assume that the compiler is interested in moving constraint c after goal g_{i+1}. When p is called with a constraint store θ, this move is acceptable if (1) all solutions to $\langle g_1, \ldots, g_i \ \Box\ \theta \rangle$ are also solutions of c; and (2) all solutions to $\langle g_1, \ldots, g_{i+1} \ \Box\ \theta \rangle$ are also solutions of c. This guarantees that c does not prune the search space in this prefix and can be moved past g_{i+1}. Note also that condition (2) alone is not sufficient, since c may prune the search space before g_{i+1} and the program postponing c may be much less efficient than the original program or may even diverge. Finally, observe that it is necessary to make sure that these conditions are verified for all constraint stores occurring at runtime.

The `LSign` analysis produces a representation of these constraints stores: this is the input store for `p`. In addition, this information can be used to produce a representation of the constraint stores occurring at the program point just before constraint c. Denote this representation β. The first proof obligation consists of showing that every satisfiable store θ in the denotation of β is consistent with c.

To explain the way this can be proven using `LSign`, the following reasoning is helpful. The satisfiability of $\theta \wedge c$ is equivalent to the satisfiability of $\exists V\ \theta \wedge c$, where V are the variables of θ which do not appear in c. In addition, $\exists V\ \theta \wedge c$ is equivalent to $(\exists V\ \theta) \wedge c$, by definition of V. The satisfiability of $(\exists V\ \theta) \wedge c$ can be tested by computing $\exists W\ (\exists V\ \theta) \wedge c$, where W are all the variables of c.

The same reasoning can be applied using `LSign` descriptions because of the consistency of the abstract operations. The consistency check becomes as follows:

1. Project β on the variables of c to obtain β_c;

2. Add the abstraction of c (i.e., the constraint where the coefficients have been replaced by signs) to β_c in order to obtain β_s;

3. Eliminate all variables from β_s to produce β_r;

4. Test if β_r is always satisfiable, i.e., if it contains only constraints of the form $\ominus \leq 0$, $\ominus \leq \oplus$, $0 \leq 0$, $0 \leq \oplus$, $\ominus < 0$, $\ominus < \oplus$, $0 < \oplus$, and $0 = 0$.

[6] The astute reader has probably realized that this means that the `Prop` information needs to be recomputed. Fortunately, the recomputation is only local to the clause being optimized.

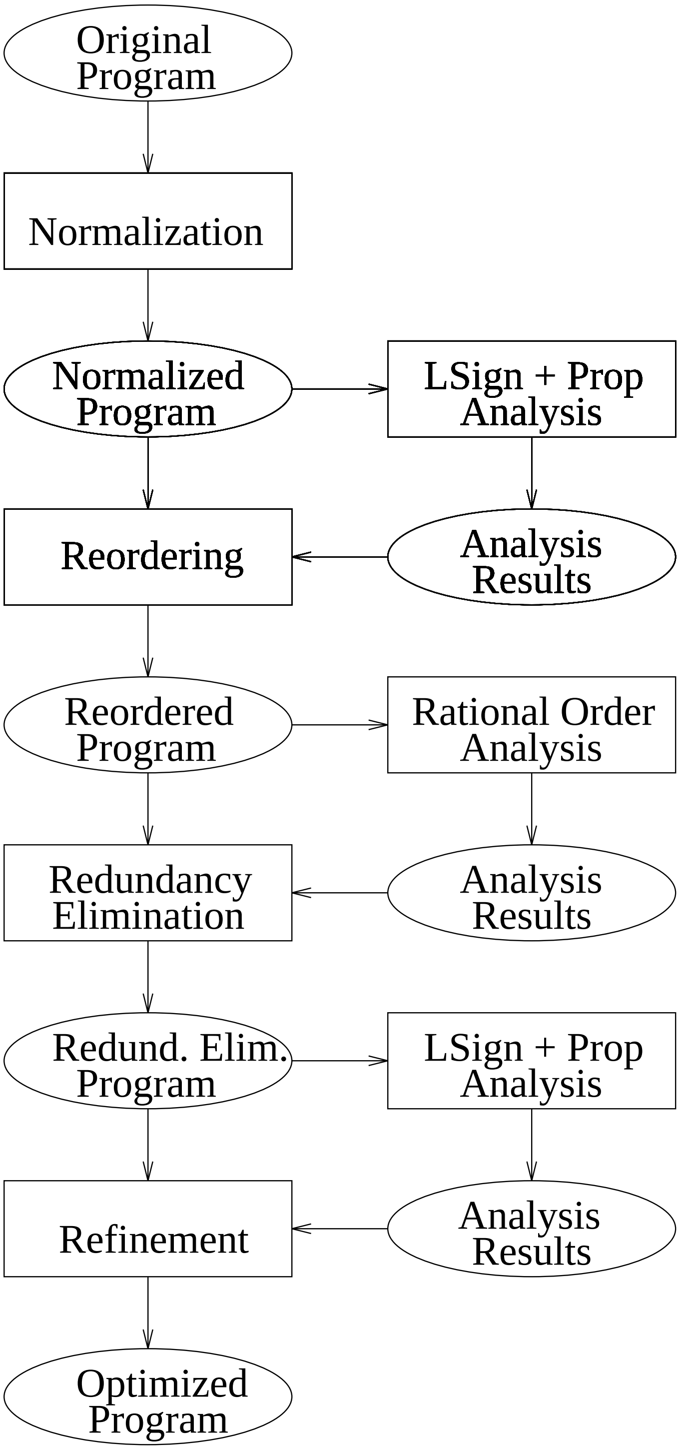

Figure 2: The Organization of The Compiler.

Note that it is only necessary to consider the satisfiable stores in β. As a consequence, it is useful to eliminate obvious sources of inconsistencies from β_c (e.g., all constraints without variables) to increase the precision of the abstract test.

The second proof obligation mentioned previously follows the same lines. The compiler considers the clause

p `:-` $g_1, \ldots, g_i, g_{i+1}, c, \ldots$

and must show that constraint c does not prune the search space. The same techniques as for the first proof obligation can be used. We now illustrate this on our running example. Consider the normalized program

```
mg(P,T,R,B) :- T = 0, B = P*1.01 - R.
mg(P,T,R,B) :- T > 0, P ≥ 0, P1 = P*1.01 - R, T1 = T - 1, mg(P1,T1,R,B).
```

and assume that the compiler tries to move constraint `T > 0` after the recursive call. The first proof obligation consists of checking whether `T > 0` is always consistent with the input store of `mg` which were depicted previously. When projected on `T`, the input store becomes

$\{\{\langle\ \ominus$ `<` $\oplus$`T , One` $\rangle\}$, $\{\langle\ \ominus$ `<` $\oplus$`T , One` $\rangle, \langle\ \ominus$ `<` $\oplus$`T , Any` $\rangle\}\ \}$.

Adding the abstraction of the constraint produces

$\{$
  $\{\langle\ \ominus$ `<` $\oplus$`T , One` $\rangle$, $\langle$ `0 <` $\oplus$`T , One` $\rangle\}$,
  $\{\langle\ \ominus$ `<` $\oplus$`T , One` $\rangle, \langle\ \ominus$ `<` $\oplus$`T , Any` $\rangle$, $\langle$ `0 <` $\oplus$`T , One` $\rangle\}$
$\}$.

All the stores in the denotation are obviously satisfiable, since the projection gives us the empty set of constraints. The second proof obligation must show that `T > 0` can be moved after `P ≥ 0`, after `P1 = P*1.01 - R`, after `T1 = T - 1`, and after the recursive call. Consider the last step of this proof. The compiler considers the clause

```
mg(P,T,R,B) :- P ≥ 0, P1 = P*1.01 - R, T1 = T - 1, mg(P1,T1,R,B), T > 0.
```

and the `LSign` description after the recursive call. This (complicated) description, when projected on `T`, produces the description

$\{\ \{\langle\ \oplus$ `=` $\oplus$`T , One` $\rangle\}\ \}$

which is obviously consistent with `T > 0`.

5.2 Constraint Removal

This phase receives the reordered program and performs the `RatOrder` analysis on the reordered program. Each constraint in each clause is then considered for constraint removal. Consider the running example again. The program at this stage is as follows:

```
mg(P,T,R,B) :- T = 0, B = P*1.01 - R.
mg(P,T,R,B) :- P ≥ 0, P1 = P*1.01 - R, mg(P1,T1,R,B), T = T1 + 1.
```

The `RatOrder` analysis produces the description $\{$`T1` $\geq 0,$ `T` $>$ `T1`$\}$ for the program point just before constraint `T > 0`, showing that the constraint is redundant.

5.3 Refinement Phase

After constraint removal, the refinement phase considers all constraints in all clauses and specializes them whenever possible. It perform both a Prop analysis and an LSign analysis on the program obtained after reordering and removal. The specialization of inequalities only uses the results of Prop and produces a test whenever all variables are fixed. The specialization of equations also uses the results of LSign to determine unconstrained variables. Consider the running example during this phase:

```
mg(P,T,R,B) :- T = 0, B = P*1.01 - R.
mg(P,T,R,B) :- P ≥ 0, P1 = P*1.01 - R, mg(P1,T1,R,B), T = T1 + 1.
```

In the second clause, the first inequality is transformed into a test, since P is fixed in the input description of mg. Since R is also fixed and since P1 is unconstrained, the second constraint can be transformed into an assignment. Similar reasoning can be applied for the remaining constraints to obtain the final program.

6 Additional Functionalities

The compiler has a number of additional functionalities that have not been discussed so far. We mention briefly some of them in this section. Many programs contain constraints that are nonlinear in the original programs but are linear at runtime due to the initial constraint store. Our compiler analyzes these programs as well. The main problem occurs in the domain LSign. The domain Prop can be used to determine which variables become ground at runtime so that they can be replaced by a sign. Finally, our compiler specializes constraints even if they cannot be fully transformed into tests and assignments. These optimizations are at the level of the abstract machine and cannot be described in this paper for space reasons.

7 Experimental Results

The purpose of this section is to evaluate the effect of the optimizations on a number of typical benchmarks. The benchmarks are relatively small, since most large CLP($\Re_{Lin}$) programs use Prolog terms in addition to linear constraints and our analyzer has a very naive handling of functors at this stage. Nevertheless, the benchmarks indicate clearly the potential of the optimizations. Most of the programs are also multi-directional and they are run with various modes: u stands for an unconstrained variable while f stands for a fixed variable. Program, Integer can be used either to check if its argument is a non-negative integer or to generate the non-negative integers. In the results, it is used to generate the first 250 integers. The next three programs, Exp2(N,R), Sum(N,R) and Factorial(N,R) are programs that compute 2^N, the sum of the first N integers, and $N!$ respectively. Fibonacci(N,S) is the well known recursive program to compute the N^{th} Fibonacci number. Length is the program to compute the length of a list. Mortgage is our running example, except that the interest rate is a argument. In our benchmarks, we have used an interest rate of 1% per month and a monthly repayment of 2 units; the final balance is unconstrained. The value of the principal and time period vary to illustrate various tradeoffs in the optimizations. Ode-euler [17] is a program solving the ordinary differential equation $y' = t$ numerically using Euler method. Some of the programs (Factorial, Mortgage and Ode-euler) are syntactically

Program	Mode	Description	`UnOpt`	`Opt`	Speedup	Annotation
Integer	f	Is 25000 an integer?	1350	1200	1.12	`REF`
	u	Generate $0 \ldots 250$	5340	320	16.69	`REO,REF,REM`
Exp2	fu	2^{25}	20	10	2.00	`REO,REF,`
	uf	$\lg 2^{25}$	90	10	9.00	`REO,REF,REM`
	uu	Generate $(0,1) \ldots (25, 2^{25})$	150	10	15.00	`REO,REF,REM`
Sum	fu	$0+1+\ldots+500$	270	30	9.00	`REO,REF,`
	uf	N s.t. $0+1+\ldots+N = 125250$	24020	20440	1.18	`REO, REM`
	uu	Generate $(0,0) \ldots (500, 125250)$	36379	2520	14.44	`REO,REF,REM`
Factorial	fu	40! 40 times	2260	80	28.25	`REO,REF,`
Fibonacci	fu	15^{th} Fibonacci number	630	90	7.00	`REO,REF`
	uf	N s.t. 987 is N^{th} Fibonacci number	8820	2540	3.47	`REF`
	uu	Generate $(0,1) \ldots (15, 987)$	8840	780	11.33	`REO,REF`
Length	fu	Length of a list of length 500	10520	10	1052.00	`REO,REF,REM`
Mortgage	ffffu	$Principal = 100, Time = 50$	2060	1740	1.18	`REF`
	ffffu	$Principal = 200, Time = 100$	30	20	1.50	`REF`
	fuffu	$Principal = 100, Time = 0 \ldots 50$	3970	2990	1.33	`REO,REF,REM`
	fuffu	$Principal = 200, Time = 0 \ldots 100$	1070	80	13.37	`REO,REF,REM`
Ode-Euler	ffffu	Compute final y value	1250	1129	1.11	`REF`
	fufff	Compute initial y value	1610	1130	1.42	`REO,REF,`
	fuffu	Relate inital and final y values	1390	1310	1.06	`REF`

Table 1: Experimental Results: Comparaison of the Running Times

non-linear, but the queries make them linear at runtime. The benchmarks compare the optimizing compiler with the standard compiler. They both generate code for the same runtime system which is a WAM-based compiler with special intructions for tests and assignments. The runtime system uses infinite precision integers to guarantee numerical stability.

Table 1 reports the computation times of the two systems and their ratios. It also specifies which optimizations have been performed. `REO` indicates that constraints were reordered, `REF` indicates that constraints were refined to tests or assignments and `REM` indicates that (redundant) constraints were removed from the program. The speedups vary from 1.06 on one of the `Ode-euler` query to 1052 on the `Length` program. Eight programs exhibit speedups greater than 10 and only eight programs exhibit speedups lower than 2. The speedup on `Length` illustrates that the optimizations can produce more than constant factors of improvement. In `Length`, the speedup depends on the length of the input list. Reordering and refinement make it possible to bypass calls to the constraint solver completely, avoiding the costly Gaussian elimination in the solver which takes place in the original program. Reordering has a similar effect on most of the other benchmarks for certain queries and the speedup depends on the size of numbers in these problems. These speedups also indicate the importance of reordering, which is the most fundamental optimization of the three. When no reordering takes place, refinement still produces interesting speedups (up to 3.47) but they are not as dramatic. `Mortgage` also illustrates a side-effect of reordering that we did not expect. As mentioned previously, the system uses infinite-precision numbers but it makes an effort (especially in the interface between the engine and the solver) to keep numbers in standard integer technology as long as no overflow occurs. When a program is reordered, it is possible that it can be run without infinite precision numbers at all. When this happens, the speedups are magnified and this explains the discrepancy between the speedups of `Mortgage` with a value 200 and with a value 100 for the principal.

Program	Mode	1st phase	2nd phase	3rd phase	Total
Integer	f	170	40	180	390
	u	200	30	110	340
Exp2	fu	260	60	220	540
	uf	330	40	220	590
	uu	290	50	170	510
Sum	fu	340	70	140	550
	uf	340	40	180	560
	uu	490	40	220	750
Factorial	fu	240	60	170	470
Fibonacci	fu	480	70	340	890
	uf	1330	70	810	2210
	uu	940	50	780	1770
Length	fu	370	60	150	580
Mortgage	ffffu	950	50	160	1160
	fuffu	960	70	140	1170
Ode-Euler	ffffu	710	60	190	960
	fufff	670	110	120	900

Table 2: Compilation Times of the Optimizing Compiler

Table 2 describes the compilation time for the same benchmarks, although no effort has been spent making the compiler fast at this point. The table depicts the time of the three optimization phases (the first phase containing the parsing and the normalization as well). As can be seen, the compilation times are reasonable for our benchmark programs.

8 Conclusion

This paper has described an optimizing compiler for CLP($\Re_{Lin}$), a constraint logic programming language over linear real constraints. The compiler performs a series of optimizations (reordering, removal, refinement) and it uses abstract interpretation on three domains (`LSign`, `RatOrder` and `Prop`) to decide when and how to perform the optimizations. The resulting system exhibits significant speedups compared to the standard compiler. In many cases, these speedups can be made arbitrarily large, since they depend on the input size.

The main technical contribution of this paper is to present the first implementation of reordering and constraint removal for CLP($\Re_{Lin}$) programs, to report the first implementation of the domain `LSign`, and to present the first implementation of a compiler integrating all these source to source transformations. The main conceptual contribution of the paper is to show that sophisticated static analyses and source to source transformations can produce dramatic (often asymptotic) speedups for CLP($\Re_{Lin}$) programs and that reordering is the most fundamental optimization.

There are many ways to extend this work. First, the domains used in our compiler can be improved to increase precision. In particular, signs in `LSign` should be replaced by intervals and the domain `RatOrder` may be replaced by the domain `Hull` of [4]. Second, it is necessary to keep track of structural information to extend the scope of this work. The basic idea is to instantiate the generic domain Pat($\Re$) [2] with the domains handling linear constraints. Pat($\Re$) automatically upgrades a domain with structural information. Third, the present source to source optimizations can be combined with lower-level optimizations such as dead-variable elimination [10]. Finally, we would like to explore various tradeoffs in the compiler organization and the optimizations.

Acknowledgments

Special thanks to Baudouin Le Charlier for his numerous contributions to this research. Many of the abstract interpretation tools used in the work have in fact been developed jointly with Baudouin Le Charlier. This work was supported in part by the Office of Naval Research under grant ONR Grant N00014-94-1-1153 and a NSF National Young Investigator Award with matching funds of Hewlett-Packard.

References

[1] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality Analysis of Prolog. In *Proceedings of the International Symposium on Logic Programming (ILPS-94)*, pages 457–471, Ithaca, NY, November 1994.

[2] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of Abstract Domains for Logic Programming. In *21st Annual ACM SIGPLAN–SIGACT Symposium on Principles Of Programming Languages*, Portland, OR, January 1994.

[3] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In New York ACM Press, editor, *Conf. Record of Fourth ACM Symposium on Programming Languages (POPL'77)*, pages 238–252, Los Angeles, CA, January 1977.

[4] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conf. Record of Fifth ACM Symposium on Principles of Programming Languages (POPL'78)*, 1978.

[5] S. Debray. Unfold/Fold Transformations and Loop Optimizations of Logic Programs. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI'92)*, pages 297–307, Atlanta, 1988.

[6] V. Dumortier and G. Janssens. Towards a Practical Full Mode Inference System for CLP(H,N). In *Eleventh International Conference on Logic Programming (ICLP-94)*, Santa Marguerita Ligure, Italy, June 1994.

[7] M. Garcia de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of CLP Programs. In *Proc. of the International Symposium on Logic Programming (ILPS'93)*, Vancouver, Canada, November 1993.

[8] N. Heintze and J. Jaffar. An Engine for Logic Program Analysis. In *IEEE 7th Annual Symposium on Logic in Computer Science*, 1992.

[9] J. Jaffar and J-L. Lassez. Constraint logic programming. In *POPL-87*, (Munich, Germany), January 1987.

[10] A.D. Kelly, A. MacDonald, K. Marriott, H. Sondergaard, P. Stuckey, and R. Yap. An Optimizing Compiler for CLP($\Re$). In *First International Conference on Principles and Practice of Constraint Programming (CP'95)*, Cassis, France, September 1995. Springer Verlag.

[11] 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, January 94.

[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, January 94.

[13] K. Marriott and H. Sondergaard. Abstract Interpretation of Logic Programs: the Denotational Approach, June 1990. To appear in ACM Transaction on Programming Languages.

[14] K. Marriott and H. Sondergaard. Analysis of Constraint Logic Programs. In *Proceedings of the North American Conference on Logic Programming (NACLP-90)*, Austin, TX, October 1990.

[15] K. Marriott and P. Stuckey. The 3 R's of optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *Proc. of the 20th ACM Symposium on Principles of Programming Languages (POPL'93)*, Charleston, South Carolina, January 1993.

[16] K. Marriott and P. Stuckey. Approximating Interaction Between Linear Arithmetic Constraints. In *Proc. of the International Symposium on Logic Programming (ILPS'94)*, Ithaca, NY, November 1994.

[17] S. Michaylov and B. Pippin. Optimizing Compilation of Linear Arithmetic in a Class of Constraint Logic Programs. In *Proceedings of the International Symposium on Logic Programming (ILPS-94)*, pages 586–600, Ithaca, NY, November 1994.

[18] K. Muthukumar and M. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2-3):315–347, August 1992.

[19] V. Ramachandran and P. Van Hentenryck. LSign Reordered. In *Static Analysis Symposium (SAS-95)*, Glasgow, UK, September 1995. Lecture Notes in Computer Science, Springer Verlag.

[20] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Evaluation of `Prop`. *Journal of Logic Programming*, 23(3), June 1995.

[21] P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in CLP($\Re_{lin}$). *ACM Transactions on Programming Languages and Systems*, 1995. (to appear).