The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal and Reordering

Kim Marriott[1]
IBM - T.J. Watson Research Center
Yorktown Heights, NY 10598, U.S.A.

Peter J. Stuckey
Department of Computer Science
University of Melbourne
Parkville 3052, Australia

Abstract

Central to constraint logic programming (CLP) languages is the notion of a global constraint solver which is queried to direct execution and to which constraints are monotonically added. We present a methodology for use in the compilation of CLP languages which is designed to reduce the overhead of the global constraint solver. This methodology is based on three optimizations. The first, refinement, involves adding new constraints, which in effect make information available earlier in the computation, guiding subsequent execution away from unprofitable choices. The second, removal, involves eliminating constraints from the solver when they are redundant. The last, reordering, involves moving constraint addition later and constraint removal earlier in the computation. Determining the applicability of each optimization requires sophisticated global analysis. These analyses are based on abstract interpretation and provide information about potential and definite interaction between constraints.

1 Introduction

Constraints provide a powerful and declarative programming paradigm, in which the objects of computation are not explicitly constructed but rather they are implicitly defined using constraints. Though the importance of constraints has been widely recognized in computer science [19], only recently have general purpose programming languages in which constraints are primitive elements been developed. These include concurrent constraint languages [16], constraint query languages for databases [12], functional constraint languages [4] and, in particular, the paradigm we shall concentrate on in this paper, constraint logic programming (CLP) languages [8]. Central to all of these languages is the notion of a global constraint solver which is queried to direct execution and to which constraints are monotonically added. Because of monotonicity, these languages are semantically simple. They are also important practically and have been used in such diverse fields as financial analysis [13], circuit design [18] and protocol testing [7]. In part this is because constraint programming allows simple and concise programs which may be used in many different ways. However, this flexibility comes at a price, as general constraint solving is expensive. Thus, in CLP, as in the other constraint paradigms, the main overhead is constraint solving and so a main goal of optimization in the compilation of constraint languages is to reduce this overhead. This is made difficult by the need to determine, at compile time, non-trivial properties of constraint interaction.

We develop a methodology for use in the compilation of CLP languages. The key idea is to transform a (monotonic) program into a non-monotonic program in which constraints are added to the constraint solver only when they are needed and are subsequently removed when they are no longer needed. The methodology has three steps in the optimization: clause refinement, constraint removal and, finally, constraint reordering.

- REFINEMENT adds new constraints to the clauses. These constraints will eventually become redundant, so the declarative meaning of the program remains the same. The advantage is that the new constraints in effect make information available earlier in the computation, and so can improve the operational behavior by guiding subsequent execution away from unprofitable choices.
- REMOVAL involves adding removal instructions to each clause indicating that a constraint previously added to the constraint solver can now be optionally removed as the constraint has become redundant in the sense that its information is duplicated in the solver. Intuitively, removal is advantageous as it reduces the number of constraints in the constraint solver without significantly changing the operational behavior. This optimization is important if programs written in monotonic languages are to approach the efficiency of programs written in non-monotonic languages.

[1]Affiliation from February 1992: Dept. of Computer Science, Monash University, Clayton 3168, Australia.

ACM-20th PoPL-1/93-S.C., USA

• REORDERING moves constraint addition later and constraint removal earlier in the clause (and hence computation) when the constraint involved cannot affect the control flow in the intervening computation. Like removal, reordering, is advantageous as it reduces the number of constraints in the constraint solver without significantly changing the operational behavior.

Each optimization is useful in its own right, but it is their complementary behavior which reveals their full power. In particular, refinement together with reordering allows constraint information to be moved throughout the computation, while removal alleviates the potential overhead created by the new constraints added in refinement as all such constraints will become redundant and so can be removed. Finally, reordering combines with removal to allow constraints to be removed even before they become redundant.

Correctness of each optimization depends on simple algebraic properties of the constraints and monotonicity of the original program. However, determining the applicability of each optimization requires sophisticated global analysis. These analyses are based on abstract interpretation and provide information about potential and definite interaction between constraints. We give generic analyses and specific analyses for the case of constraint solving over linear arithmetic.

The optimizations we introduce have significant practical importance as preliminary test results show that in conjunction with low-level optimizations they can give order of magnitude improvements in execution speed. Though the specific details of the optimizations and analyses are for languages based on the CLP framework, we feel that the underlying ideas are applicable to the compilation of other constraint based programming languages.

Constraint based programming and database languages are a relatively new development. Accordingly there has been little research into global optimization and compilation techniques for these languages. Low level optimization for the language CLP($\mathcal{R}$) [9], where constraint solving is partially compiled into imperative statements is discussed in Jørgensen *et al.* [11], Jaffar *et al.* [10], and Marriott and Søndergaard [15]. While application of these techniques requires global information, the optimizations are essentially local. The optimizations discussed here, however, are global in nature and hence offer potentially greater performance improvements. In fact, as we shall demonstrate, the two types of optimization are complementary. The refinement optimization generalizes the optimizations suggested by Sato and Tamaki [17] and Marriott *et al.* [14] in the context of logic programming, and is related to constraint propagation transformations by Kemp *et al.* [6] for deductive databases. The removal and reordering optimizations are novel, although the future redundancy optimization of Jørgensen *et al.* [11] can be seen as a particular special combination of the two in the case information about calls is ignored.

The remainder of this paper is organized as follows. In the next section we informally introduce the optimizations using a simple example. In Section 3 we give preliminary notation and an operational semantics for CLP programs with removal instructions. In Section 4 we detail each optimization, and in Section 5 we give a generic analysis to support each optimization. Section 6 contains a more realistic example of the optimizations use and Section 7 concludes.

2 An Example

In this section we informally introduce the optimizations through a worked example. Consider the following CLP($\mathcal{R}$) program defining the relation *sum* where $sum(n, s) \leftrightarrow s = 1 + \cdots + n$. It can be viewed as a straightforward translation of the recursive mathematical definition.

```
sum(N, S) ← N = 0 ∧ S = 0.                    (SUM)
sum(N, S) ← N >= 1 ∧ S = N + S1 ∧
            N1 = N - 1 ∧ sum(N1, S1).
```

CLP($\mathcal{R}$) programs may be thought of as Prolog programs extended to allow linear arithmetic constraints as well as syntactic equality constraints. Upper case symbols in the clause denote variables.

The first step in optimizing the program is to determine which constraints are implied by the program for the calls that we are interested in. In this example we consider all possible calls. For any call to `sum(N, S)` every answer satisfies the constraints `N >= 0 ∧ S >= N`. We refine the program by adding the redundant constraints to every clause body where the calls are made. Hence the refined program is

```
sum(N, S) ← N = 0 ∧ S = 0.                    (REF)
sum(N, S) ← N >= 1 ∧ S = N + S1 ∧
            N1 = N - 1 ∧ N1 >= 0 ∧
            S1 >= N1 ∧ sum(N1, S1).
```

The chief advantage of the refined program is that it will work with a wider class of calls than the original. For instance, it terminates (with answer **no**) for the query `?- sum(N, 5)` whereas the original will run forever. As refinement may change the operational behavior of the program, albeit for the better, it is different in nature to the other steps in the optimization. Thus refinement might be performed in an optional pre-compilation stage, in which the programmer can intervene.

The next step in the optimization is to determine which constraints are made redundant in further execution and hence where they can be removed. Clearly in the refined program, for any call, each of the constraints `N1 >= 0` and `S1 >= N1` is redundant after the call `sum(N1, S1)`, this being why they were added, and so they can be removed. Thus, in general, all constraints added by refinement will be subsequently removed. In this case we can do better, `N1 >= 0` is redundant before it is reached because of the constraints `N >= 1 ∧ N1 = N - 1`, hence we can remove it from the program altogether. The constraint `N >= 1` is redundant after the call `sum(N1, S1)` just because `N1 >= 0` was. The removal optimized program therefore is

Query	SUM	REF	REO	SPE	DET	LOW	Speed Up
N <= 100	86.48	143.88	53.95	—	—	—	1.60
S <= 1000	∞	34.00	14.55	—	—	—	∞
N = 100	2.26	19.50	5.63	1.85	0.16	0.80	5.00

Table 1: Timings for the Optimized Sum Program

```
sum(N, S) ← N = 0 ∧ S = 0.                    (REM)
sum(N, S) ← N >= 1 ∧ S = N + S1 ∧
            N1 = N - 1 ∧ S1 >= N1 ∧
            sum(N1, S1) ∧ rem(N >= 1) ∧
            rem(S1 >= N1).
```

Finally we reorder the constraints and their removals. First consider the case of reordering in the context of all possible calls. Consider the constraint `N >= 1`. It is not always satisfiable when reached for every call and hence cannot be moved later in the clause. But during the recursive call to `sum(N1, S1)` the constraint does not affect the computation before it is made redundant, (by `N = 0` or `N >= 1`) and hence its removal can be moved *before* the call to `sum(N1, S1)`. Similar reasoning applies to `S1 >= N1`. The resulting reordering optimized program is

```
sum(N, S) ← N = 0 ∧ S = 0.                    (REO)
sum(N, S) ← N >= 1 ∧ S = N + S1 ∧
            N1 = N - 1 ∧ S1 >= N1 ∧
            rem(N >= 1) ∧ rem(S1 >= N1) ∧
            sum(N1, S1).
```

Note that the resulting program, `REO`, not only adds one less inequality per level of recursion than the original program, `SUM`, but also terminates for a wider class of calls.

Given more information about the intended calls to a program even better code is produced. Consider calls where `N` is given a fixed value and `S` is unconstrained. In this case the constraint `S1 >= N1` is always satisfiable when reached. Hence it can be moved past its removal, and thus removed entirely from the program. Similarly the constraint `S = N + S1` does not affect execution during the recursive call and can be moved afterwards. The resulting program is

```
sum(N, S) ← N = 0 ∧ S = 0.                    (SPE)
sum(N, S) ← N >= 1 ∧ rem(N >= 1) ∧
            N1 = N - 1 ∧ sum(N1, S1) ∧
            S = N + S1.
```

If we combine this with low level optimizations that replace constraints with tests and assignments whenever possible, we arrive at a deterministic program which does not make use of the constraint solver!

```
sum(N) if N == 0 then S := 0                  (DET)
       else if N >= 1 then
            N1 := N - 1
            sum(N1, S)
            S := S + N.
```

This example illustrates the synergy between the transformations given here and previous low level optimizations suggested for CLP languages. In particular, refinement can make non-deterministic programs deterministic, and reordering can allow replacement of calls to the constraint solver by simple Boolean tests and assignments.

Even with this simple program the optimizations offer significant speedup. In Section 6 we give another example, in which the results are even more encouraging. The effect of the optimizations on three different types of call are shown in Table 1. The bottom line is given by the speed up column which is the ratio between the time for the original program and the optimized program, with low level optimizations applied to both wherever possible. Actual times shown are CPU seconds for 100 executions of the given queries on a SparcStation 2 using CLP($\mathcal{R}$) v1.1. The program `LOW` not shown is the result of applying low level optimizations to the original program `SUM`.

3 Operational Semantics of CLP

In this section we give some preliminary notation and an operational semantics for constraint logic programs in which constraints can be removed.

A *constraint logic program*, or *program*, is a finite set of clauses. A *clause* is of the form $H \leftarrow B$, where H, the *head* is an atom, and B the *body* is a sequence of the form $B_1 \wedge \cdots \wedge B_n$ where each B_i is a literal. A *literal* is an atom, a primitive constraint, or a removal instruction of the form $rem(B_j)$ where the primitive constraint B_j occurs earlier in the clause. We let $B_{i,j}$ represent the sequence $B_i \wedge \cdots \wedge B_j$ when $i < j$ and *nil* otherwise. We associate a *program point* $i \in \{0, 1, 2, ..., n\}$ with the point immediately after B_i. Note that program point 0 is the point just before B_1. An *atom* has the form $p(x_1, ..., x_n)$ where p is a predicate symbol and the x_i are distinct variables.

A *primitive constraint* is essentially a pre-defined predicate over some computation domain. For example the primitive constraints in CLP($\mathcal{R}$) are linear arithmetic equalities and inequalities over the real numbers and syntactic equalities over terms. A *constraint* is a multiset of primitive constraints. We let *true* denote the empty multiset of constraints. The reason we introduce the multiset representation is to enable correct removal of constraints, but often we will treat a constraint simply as the conjunction of its primitive constraints. Thus constraints are pre-ordered by logical implication, that is $\theta \leq \theta'$ iff $\theta \Rightarrow \theta'$. We let $\exists_W \theta$ be a non-deterministic function which returns a constraint logically equivalent to $\exists\, V_1 \exists\, V_2 \cdots V_n \theta$ where variable set $W = \{V_1, \ldots, V_n\}$. We let $\overline{\exists}_S \theta$ be constraint θ restricted to the variables in S. That is $\overline{\exists}_S \theta$ is $\exists_{(vars\ \theta) \setminus (vars\ S)} \theta$ where function *vars* takes a syntactic object and returns the set of (free) variables occurring

in it. Finally, we let $\tilde{\exists}\theta$ denote the existential closure of θ and frequently write $\tilde{\exists}\theta$ in place of "where θ is satisfiable."

Var is the set of variables, *Atom* the set of atoms, *Prim* the set of primitive constraints, *Rem* the set of removal instructions, *Cons* the set of constraints, *Clause* the set of clauses, and *Prog* the set of CLP programs. Programs without removal instructions are said to be *monotonic*.

A *renaming* is a bijective mapping from *Var* to *Var*. We let *Ren* be the set of renamings, and naturally extend renamings to mappings between atoms, clauses, and constraints. Syntactic objects s and s' are said to be *variants* if there is a $\rho \in Ren$ such that $\rho\ s = s'$. The *definition of an atom A in program P with respect to variables W*, $(defn_P\ A\ W)$, is the set of variants of clauses in P such that each variant has A as a head and has variables disjoint from $(W - vars\ A)$.

The operational semantics of a program is in terms of its "derivations" which are reduction sequences of "states" where a state consists of the current literal sequence, or "goal", and the current constraint. More formally,

$$\begin{aligned} Goal &= (Atom + Prim + Rem)* \\ State &= Goal \times Cons. \end{aligned}$$

A *derivation* of state s for program P is a sequence of states $s_0 \rightarrow \cdots \rightarrow s_n$ where $s = s_0$ and there is a reduction from s_i to s_{i+1} where state $\langle L : G, \theta\rangle$ can be *reduced* as follows:

1. If $L \in Prim$ and $\tilde{\exists}(L \wedge \theta)$, it can be reduced to $\langle G, \{L\} \uplus \theta\rangle$;
2. If $L \in Rem$ and $L = rem(L')$, it can be reduced to $\langle G, \theta - \{L'\}\rangle$ or to $\langle G, \theta\rangle$;
3. If $L \in Atom$, it can be reduced to $\langle B :: G, \theta\rangle$ where $\exists(L \leftarrow B) \in (defn_P\ L\ (vars\ G \cup vars\ \theta))$.

Note that $\uplus$ denotes multiset union and :: concatenation of sequences.

The *length* of a derivation is the number of atom reductions occurring in the derivation, that is the number of reductions using reduction rule (3). A derivation is *successful* if the last state in the derivation has the empty goal. Such a state is called a *success state*. The constraint $\bar{\exists}_s\theta$ is a *partial answer* to state s if there is a derivation from s to some state with constraint θ. An *answer* to state s is a partial answer corresponding to a successful derivation. We denote the set of answers to s for P by $(answers_P\ s)$, the partial answers by $(panswers_P\ s)$ and the derivations by $(deriv_P\ s)$. When restricted to monotonic programs the above definition is equivalent to the usual operational semantics [8].

A derivation is *quasi-monotonic* iff in every application of reduction rule (2), $\bar{\exists}_G\ \theta$ is logically equivalent to $\bar{\exists}_G\ (\theta - \{L'\})$. A program P is *quasi-monotonic* for states S if every derivation from a state $s \in S$ for P is quasi-monotonic.

Information about the intended use of the program, such as which predicates are exportable from a module and which are local, allows stronger optimization as the optimization need only preserve behavior for those uses. Thus we will optimize the program for a given set of initial states and optimize the definitions of an atom only for the particular "calls" made to it in the derivations from those states. More precisely, given a program P and a set of states S, a *call* to atom A is a constraint $\bar{\exists}_A(\rho^{-1}\ \theta)$ such that for some renaming ρ there is state $\langle(\rho\ A) : G, \theta\rangle$ in a derivation from some state in S. We let $(call\ P\ A\ S)$ denote the set of such calls.

4 Optimizations

4.1 Refinement

In refinement, constraints which will become redundant in the future are added to the start of the clause. As the constraints are redundant they do not affect the answers and so the declarative semantics is preserved. The advantage is that constraint information is moved earlier in the execution and so unsuccessful derivations will be pruned earlier. In particular this may transform a non-deterministic atom definition into a deterministic definition and allow the program to be used for wider variety of calls as these calls will now finitely fail rather than lead to an infinite derivation.

Definition. Constraint θ is *redundant* for set of constraints Θ iff $\forall\theta' \in \Theta\ .\ \theta' \Rightarrow \theta$. □

Definition. Let S be a set of initial states and P a monotonic program. We obtain a *refinement* of P for S by taking each clause $H \leftarrow B$ in P and rewriting it to:

$$H \leftarrow \theta \wedge (B - \theta)$$

where θ is redundant for $(answer_P\ \langle B, \theta'\rangle)$ for all $\theta' \in (call\ P\ H\ S)$. In the case θ is *false* the clause is simply removed. □

Theorem 4.1 (Correctness of refinement) Let P' be a refinement of monotonic program P for states S. For all $s \in S$, $answer_P\ s = answer_{P'}\ s$. Furthermore, for each $D' \in (deriv_{P'}\ s)$ there is a corresponding $D \in (deriv_P\ s)$ of the same length. ■

4.2 Removal

In the removal optimization clause bodies are augmented with declarations of the form $rem(\theta)$ indicating that at this point constraint θ, previously introduced in the clause, may be optionally removed without affecting the computation as it has become redundant with regard to the current constraint. This is a very important optimization as it alleviates the overhead associated with languages in which constraints can only be monotonically added. A strength of this optimization is that it can always be used to remove the new constraints added by refinement. In fact we can weaken the conditions for this optimization as we are not concerned about local variables appearing in the constraints, but really only require that the constraint is redundant with respect to the variables appearing in the original goal and the subsequent computation.

Definition. Primitive constraint θ is *W-redundant* for set of constraints Θ iff

$$\forall \theta' \in \Theta \,.\, \tilde{\exists}_W(\theta' \wedge \theta) \Leftrightarrow \tilde{\exists}_W \theta'.$$

Let $H \leftarrow B$ be a clause with n body literals. Primitive constraint B_j is *redundant at program point k*, $k \geq j$, for constraints Θ and program P iff B_j is W-redundant for $(answer_P\ \langle (B_{1,j-1} \wedge B_{j+1,k}), \theta \rangle)$ for each $\theta \in \Theta$ where $W = (vars\ H \cup vars\ B_{k+1,n})$. □

The only difficulty in the optimization is ensuring that the removals do not "interfere" as removing one constraint may invalidate the removal of another. For example, consider optimizing the following program P for the initial state $s = \langle \mathtt{p(X)}, true \rangle$.

```
p(X) ← X = 1 ∧ q(X).
q(X) ← X = 1.
```

Now

$$\begin{array}{lcl} call\ P\ p(X)\ \{s\} & = & \{true\} \\ call\ P\ q(X)\ \{s\} & = & \{X = 1\}. \end{array}$$

However, if p is optimized with respect to the call *true* and q is optimized with respect to the call $X = 1$ we obtain

```
p(X) ← X = 1 ∧ q(X) ∧ rem(X = 1).
q(X).
```

which is not equivalent to the original program. The point is that we must guarantee that the removals are correct for the calls which will be encountered when executing the program P' which *results* from the optimization.

A sufficient condition for this to hold is that the removals are correct with respect to the the calls encountered when executing the "skeleton" of the original program, where the *skeleton* of a program P is the program obtained by removing all constraints from P. We note that this definition of skeleton can be relaxed so as to leave all constraints which are not affected by the removal or reordering optimization.

Definition. Let S be a set of states and P a program with skeleton P_{ske}. We obtain a *removal optimization* of P for S by taking each clause $H \leftarrow B$ in P and repeatedly rewriting it as follows. Assume that B has n literals. Let B_j be some primitive constraint in B. If B_j is redundant at program point j for $(call\ P_{ske}\ H\ S)$ and P there is no need to add B_j, and so we rewrite the clause to:

$$H \leftarrow B_{1,j-1} \wedge B_{j+1,n}.$$

Otherwise if B_j is redundant at program point $k > j$ for $(call\ P_{ske}\ H\ S)$ and P we rewrite to:

$$H \leftarrow B_{1,k} \wedge rem(B_j) \wedge B_{k+1,n}.\square$$

Execution of the optimized program mimics that of the original program, except that constraints which have become redundant with respect to the other current constraints may be removed, and so:

Theorem 4.2
(Correctness of removal optimization).
Let P' be a removal optimization of monotonic program P for S. Then for all $s \in S$, $answer_P\ s = answer_{P'}\ s$. Furthermore, there is an isomorphism between $(deriv_{P'}\ s)$ and $(deriv_P\ s)$ in which length is preserved. ■

The intuitive advantage of the removal optimization is clear: at each point in the corresponding derivation there are fewer constraints in the solver, leading to faster tests for satisfiability. However we note that the actual benefits obtained depend on the particular constraint domain and solver, and must take into account the cost of the removal itself. For example, in the case of term equations there is little gain as unification automatically removes redundant equalities, while in the case of more complex constraint domains such as linear inequalities, the benefits are large as redundant constraints are a major source of overhead in the solver. For this reason, removal instructions are optional advice to the compiler and run-time system.

4.3 Reordering

For different calls the order in which constraints are added during execution should be different so as to reflect their actual use in the computation. This motivates the final optimization, called reordering, in which a primitive constraint is moved later in the clause body until it will be actually used to prune a derivation, and a removal instruction $rem(\theta)$ is moved earlier in the body to where θ is last used to prune a derivation. The advantage of reordering is clear: adding constraints later and removing constraints earlier leads to smaller current constraints and consequent reduced cost of testing satisfiability. Correctness of the reordering optimization is based on the observation that a constraint does not prune a derivation from a state iff it is "consistent" with all of the state's partial answers.

Definition. A primitive constraint θ is *consistent* with set of constraints Θ iff $\forall \theta' \in \Theta \,.\, \tilde{\exists}\theta' \Rightarrow \tilde{\exists}(\theta' \wedge \theta)$. Let $H \leftarrow B$ be a clause with n body literals. Primitive constraint B_j is *consistent between program points i and k*, where $j \leq i < k$, for constraints Θ and program P iff B_j is consistent with $panswers_P\ \langle B_{i+1,k}, \theta' \rangle$ for each $\theta' \in answers_P\ \langle B_{1,j-1} \wedge B_{j+1,i}, \theta \rangle$ and $\theta \in \Theta$. We say B_j is *weakly consistent between program points i and k*, where $j \leq i < k$, for constraints Θ and program P iff B_j is consistent with $panswers_P\ \langle B_{i+1,k}, \theta' \rangle$ for each $\theta \in \Theta$ and $\theta' \in answers_P\ \langle B_{1,j-1} \wedge B_{j+1,i}, \theta \rangle$ such that $\tilde{\exists}(\theta' \wedge B_j)$. □

Analogously to the optimization for removal, we must be careful to ensure that the reorderings do not interfere. Again we make use of the skeleton. However this time we require that the reordering is correct for the "convex closure" of the calls encountered using the original program and its skeleton. We define the *convex closure*, $(convex\ \Theta)$, of a set of constraints Θ to be

$$\{\theta \mid \exists \theta', \theta'' \in \Theta \,.\, \theta'' \leq \theta \leq \theta'\}.$$

Definition. Let S be a set of states and P a program with skeleton P_{ske}. We obtain a *reordering optimization* of P for S by taking each clause $H \leftarrow B$ from P and repeatedly rewriting it as follows.

If primitive constraint B_j is consistent between program points j and k for

$$convex\ (call\ P_{ske}\ H\ S) \cup (call\ P\ H\ S)$$

and P then we rewrite the clause to:

$$H \leftarrow B_{1,j-1} \wedge B_{j+1,k} \wedge B_j \wedge B_{k+1,n}.$$

If literal $B_l = rem(B_j)$ and B_j is weakly consistent between program points i and $l-1$ for

$$convex\ (call\ P_{ske}\ H\ S) \cup (call\ P\ H\ S)$$

and P then we rewrite the clause to

$$H \leftarrow B_{1,i} \wedge rem(B_j) \wedge B_{i+1,l-1} \wedge B_{l+1,n}.\square$$

Theorem 4.3
(Correctness of reordering optimization).
Let P' be a reordering optimization of program P for states S. For all $s \in S$, $answer_P\ s = answer_{P'}\ s$. Furthermore, there is an isomorphism between $(deriv_{P'}\ s)$ and $(deriv_P\ s)$ in which length is preserved. ■

5 Analyses

In this section we present analyses to support the optimizations given in Section 4. These analyses are formalized in terms of abstract interpretation and are generic in the descriptions used for the constraints. The exact descriptions chosen depend on the underlying constraint domain. Here we give descriptions and analyses for the case that the underlying constraint domain is, *Lin*, the linear arithmetic equalities and inequalities. This domain is simple, yet still non-trivial to analyze as the programs over it (CLP(*Lin*) programs) are Turing complete. Furthermore *Lin* is a significant subset of the constraints used in all general-purpose CLP languages and so any analysis of programs in these languages must handle such constraints.

5.1 Abstract Interpretation

In abstract interpretation [1] an analysis is formalized as a non-standard interpretation of the data types and functions over those types. Correctness of the analysis with respect to the standard interpretation is argued by providing an "approximation relation" which holds whenever an element in a non-standard domain describes an element in the corresponding standard domain. We define the approximation relation in terms of an "abstraction function" which maps elements in in the standard domain to their "best" description.

Definition. A *description* $\langle D, \alpha, E\rangle$ consists of a *description domain* (a complete lattice) D, a *data domain* (a complete lattice) E, and a strict and continuous *abstraction function* $\alpha : E \to D$.

We say that d α-approximates e, written $d \propto_\alpha e$, iff $\alpha\ e \leq d$. The approximation relation is lifted to functions as follows. Let $\langle D_1, \alpha_1, E_1\rangle$ and $\langle D_2, \alpha_2, E_2\rangle$ be descriptions, and $F : D_1 \to D_2$ and $F' : E_1 \to E_2$ be functions. Then $F \propto F'$ iff

$$\forall d \in D_1.\ \forall e \in E_1.\ d \propto_{\alpha_1} e \Rightarrow (F\ d) \propto_{\alpha_2} (F'\ e).$$

We lift to predicates by taking the convention that *Bool* is ordered by *true* $\leq$ *false* and that the description associated with *Bool* is $\langle Bool, Id, Bool\rangle$ where $Id = \lambda b.b$. This means that approximation on predicates is conservative in the sense that it gives information about things which are definitely true. When clear from the context we say that d approximates e and write $d \propto e$ and we will sometimes let D denote both the description and the description domain.

In our analyses we will be concerned with describing sets of constraints. Such a description is called a *constraint description*. We now give three example constraint descriptions for the powerset of *Lin* constraints, $\wp\, Lin$. These can be used with the generic analyses developed in the next section to analyze CLP(*Lin*) programs.

The first description is the "convex hull" description, which is based on descriptions used by Cousot and Halbwachs [3] for bounds analysis in conventional programming languages.

Definition. The *convex hull description*

$$\langle CHull, \alpha_{CHull}, \wp\, Lin\rangle$$

is defined as follows. The description domain $CHull \subset Lin$ consists of all sets of linear constraints of the form $s \leq s'$ and $s = s'$. It is ordered by logical implication. The abstraction function $\alpha_{CHull} : \wp\, Lin \to CHull$ is defined by

$$\alpha_{CHull}\ \Pi = convex\_hull\ \{loosen\ \theta \mid \theta \in \Theta\}$$

where $(convex\_hull\ \Theta)$ is a set of constraints representing the convex hull of the polytopes $\theta \in \Theta$ and $(loosen\ \theta)$ returns a set of constraints in which strict inequalities are relaxed to non-strict inequalities. □

Sign descriptions are a novel constraint description in which the actual coefficients in the linear constraints are abstracted by their "sign".

Definition. The *sign description*

$$\langle LSign, \alpha_{LSign}, \wp\, Lin\rangle$$

is defined as follows. The description domain *LSign* consists of sets of "linear" equality and inequality constraints in which the coefficients and constants are elements of $Sign = \{\oplus, 0, \ominus, \top\}$. The domain is finite for a fixed finite set of variables. The abstraction function $\alpha_{LSign} : \wp\, Lin \to LSign$ is defined by

$$\alpha_{LSign}\ \Theta = \{(Csign\ \theta) \mid \theta \in \Theta\}$$

where, if $(sign\ c)$ returns the sign of the coefficient c and θ is

Let P be a program and $ACons$ a constraint description. The *answer semantics* for P and $ACons$ has semantic functions

$$\begin{array}{lcl} \mathbf{ans\_lit}_P & : & (Atom + Prim + Rem) \to ACons \to ACons \\ \mathbf{ans\_cla}_P & : & Clause \to ACons \to ACons \\ \mathbf{ans\_goal}_P & : & Goal \to ACons \to ACons. \end{array}$$

The semantic equations are

$$\begin{array}{lcl} \mathbf{ans\_lit}_P\ A\ \Pi & = & \bigsqcup\{\mathbf{ans\_cla}_P\ C\ \Pi) \mid C \in (defn_P\ A\ \{\})\} \text{ when } A \in Atom \\ \mathbf{ans\_lit}_P\ \theta\ \Pi & = & (Aadd\ \theta\ \Pi) \text{ when } \theta \in Prim \\ \mathbf{ans\_lit}_P\ rem(\theta)\ \Pi & = & (Aremove\ \theta\ \Pi) \text{ when } rem(\theta) \in Rem \\ \mathbf{ans\_cla}_P\ (A \leftarrow B)\ \Pi & = & (Ameet\ \Pi\ (Arestrict\ (vars\ A)\ (\mathbf{ans\_goal}_P\ B\ (Arestrict\ (vars\ A)\ \Pi)))) \\ \mathbf{ans\_goal}_P\ nil\ \Pi & = & \Pi \\ \mathbf{ans\_goal}_P\ A : B\ \Pi & = & \mathbf{ans\_goal}_P\ B\ (\mathbf{ans\_lit}_P\ A\ \Pi). \end{array}$$

where the functions *Aadd*, *Ameet*, *Arestrict* and *Aremove* are required to approximate *add*, *meet*, *restrict* and *remove* respectively.

Figure 1: The answer semantics for P and $ACons$

$$c_0 \left\{ \begin{array}{c} \leq \\ = \\ < \end{array} \right\} c_1 \cdot x_1 + ... + c_n \cdot x_n$$

then $(Csign\ \theta)$ is

$$(sign\ c_0) \left\{ \begin{array}{c} \leq \\ = \\ < \end{array} \right\} (sign\ c_1) \cdot x_1 + ... + (sign\ c_n) \cdot x_n . \square$$

Elements of *LSign* can be used to determine when sets of constraints are definitely satisfiable. Thus *LSign* acts as a kind of "definite degrees of freedom" analysis. Manipulation of *LSign* is performed with an abstract Fourier algorithm which mimics the manipulation of Fourier's algorithm on concrete linear constraints. Fourier's algorithm works by repeatedly eliminating variables, much like Gaussian elimination, until no variables are left. The original system is satisfiable iff the final system is.

As an example of the abstract Fourier algorithm's execution we show how it can be used to prove that the constraints $\{x \geq 0, y + x = 3z, -z \geq -4, y \geq -3\}$ are satisfiable. The algorithm starts with the description

$$\{\oplus x \geq 0,\ \ominus y + \ominus x + \oplus z = 0,\ \ominus z \geq \ominus,\ \oplus y \geq \ominus\}.$$

It first eliminates z to obtain

$$\{\oplus x \geq 0,\ \ominus y + \ominus x \geq \ominus,\ \oplus y \geq \ominus\}.$$

Then x is eliminated, giving

$$\{\ominus y \geq \ominus,\ \oplus y \geq \ominus\}.$$

Finally y is eliminated to give $\{0 \geq \ominus\}$ which is always satisfiable, indicating that any system described by the original description is satisfiable.

The final example is a description which combines *CHull* and *LSign*.

Definition. The *combination description* is

$$\langle LComb, \alpha_{LComb}, \wp\, Lin \rangle$$

where $LComb = CHull \times LSign$ and the abstraction function $\alpha_{LComb} : \wp\, Lin \to LComb$ is defined by

$$\alpha_{LComb}\ \Theta = \langle (\alpha_{CHull}\ \Theta), (\alpha_{LSign}\ \Theta) \rangle . \square$$

5.2 Refinement

Refinement requires us to find primitive constraints which are redundant with respect to the answers of a state. This may be done by choosing a suitable subset *DCons* of the constraints *Cons* as the description domain with an approximation relation $\propto_{DCons}$ defined by $\pi \propto_{DCons} \Theta$ iff π is redundant for Θ. For instance *CHull* is such a description in the case of linear constraints. These descriptions are then used in an analysis which finds an approximation to the answers of the state. By construction of the description and correctness of the analysis, any primitive constraint in the approximation must be redundant with respect to the state's answers.

The generic problem of finding a description for the answers of a state has been addressed by Marriott and Søndergaard [15] in the case of monotonic programs. The semantic equations given in Figure 1 are a straightforward modification of those given in [15] and can be used as the basis for such analyses. The semantic equations make use of the functions *add* which adds a primitive constraint to a set of constraints, *meet* which adds a set of equations, *restrict* which restricts a set of constraints to a variable set and *remove* which removes a primitive constraint from a set of constraints. They are defined by:

$$\begin{array}{lcl} add\ \theta\ \Theta & = & \{\{\theta\} \uplus \theta' \mid \theta' \in \Theta, \tilde{\exists}(\theta \wedge \theta')\} \\ meet\ \Theta\ \Theta' & = & \{\theta \uplus \theta' \mid \theta \in \Theta \wedge \theta' \in \Theta', \tilde{\exists}(\theta \wedge \theta')\} \\ restrict\ W\ \Theta & = & \{\overline{\exists}_W\ \theta \mid \theta \in \Theta\} \\ remove\ \theta\ \Theta & = & \{\theta' - \{\theta\} \mid \theta' \in \Theta\} \cup \Theta. \end{array}$$

The answer semantics is best understood by considering the equations obtained by replacing the abstract

Let P be a monotonic program and $ACons$ a constraint description. The *definitely redundant semantics* for P and $ACons$ has semantic functions

$$\begin{array}{lcl} \mathbf{red\_lit}_P & : & Prim \to \wp\, Var \to (Atom + Prim + Rem) \to ACons \to Bool \\ \mathbf{red\_cla}_P & : & Prim \to \wp\, Var \to Clause \to ACons \to Bool \\ \mathbf{red\_goal}_P & : & Prim \to \wp\, Var \to Goal \to ACons \to Bool. \end{array}$$

The semantic equations are

$$\begin{array}{lcl} \mathbf{red\_lit}_P\ \theta\ W\ A\ \Pi & = & \bigwedge\{(\mathbf{red\_cla}_P\ \theta\ W\ C\ \Pi) \mid C \in (defn_P\ A\ (vars\ \theta \cup W))\}, A \in Atom \\ \mathbf{red\_lit}_P\ \theta\ W\ \theta'\ \Pi & = & (Ared\ \theta\ W\ (Aadd\ \theta'\ \Pi)) \text{ when } \theta' \in Prim \\ \mathbf{con\_lit}_P\ \theta\ W\ rem(\theta')\ \Pi & = & false \text{ when } rem(\theta') \in Rem \\ \mathbf{red\_cla}_P\ \theta\ W\ (A \leftarrow B)\ \Pi & = & (\mathbf{red\_goal}_P\ \theta\ W\ B\ (Arestrict\ (vars\ A \cup vars\ \theta \cup W)\ \Pi)) \\ \mathbf{red\_goal}_P\ \theta\ W\ nil\ \Pi & = & false \\ \mathbf{red\_goal}_P\ \theta\ W\ A : B\ \Pi & = & (\mathbf{red\_lit}_P\ \theta\ W\ A\ \Pi) \vee (\mathbf{red\_goal}_P\ \theta\ W\ B\ (\mathbf{ans\_lit}_P\ A\ \Pi)). \end{array}$$

where *Aadd*, *Arestrict* and *Ared* are required to approximate *add*, *restrict* and *red* respectively.

Figure 2: The definitely redundant semantics for P and $ACons$

functions *Aadd*, *Ameet*, *Arestrict* and *Aremove* by *add*, *meet*, *restrict* and *remove* respectively and $ACons$ by $\wp\, Cons$. In this case we can show that

$$\forall \theta \in \Theta\,.\,(\mathbf{ans\_goal}_P\ G\ \Theta) \supseteq answers_P\langle G, \theta\rangle.$$

Using results from abstract interpretation theory we therefore have that:

Theorem 5.1 If $\Pi \propto \Theta$,

$$\forall \theta \in \Theta\,.\,(\mathbf{ans\_goal}_P\ G\ \Pi) \propto answers_P\langle G, \theta\rangle. \quad \blacksquare$$

Note that in the semantic equations when finding the answers to an atom, the current set of abstract constraints is restricted to the variables in the atom. On return the lost information is regained by "meeting" the answer constraints with the call constraints. This is important, as it means that if these equations are interpreted using a memoization approach [5], then they give a terminating analysis whenever the constraint description domain has finite height. This is because for any goal G and description Π, $(\mathbf{ans\_goal}_P\ G\ \Pi)$ depends on only a finite number of calls to **ans_goal**, modulo variable renaming. If the description domain does not have finite height, then widening/narrowing techniques [2] may be used to ensure termination. Similar comments will apply to the other analyses we introduce.

Consider P the program for *sum*, given in Section 2. Using *CHull* as the description domain, we can show that

$$\mathbf{ans\_goal}_P\ sum(N1, S1)\ true = N1 \geq 0 \wedge S1 \geq N1.$$

This information allows the refinement optimization given in the example.

It is straightforward to modify the answer semantics so that it returns descriptions of the calls made to each atom for a particular set of goals. Intuitively we need only to change the first semantic equation in Figure 1 so it "collects" the abstract call $(Arestrict\ (vars\ A)\ \Pi)$ for A.

5.3 Removal

To perform the removal optimization we must find when a primitive constraint is definitely W-redundant for a set of goals. The analysis given here is only for monotonic programs. In our context this is not a limitation as it will only be used to analyze programs which are monotonic. This reason for our restriction is that it allows us to make the analysis more efficient, because monotonicity ensures that once the constraint is W-redundant on a derivation it will remain so. Thus when the analysis finds a constraint is definitely W-redundant, it terminates with *true*. The semantic equations for the analysis are shown in Figure 2. They make use of the function

$$red\ \theta\ W\ \Theta \quad \Leftrightarrow \quad \theta \text{ is } W\text{-redundant for } \Theta.$$

Theorem 5.2 Let P be a monotonic program. If $\Pi \propto \Theta$ and $(\mathbf{red\_goal}_P\ \theta\ W\ G\ \Pi)$ holds, then θ is W-redundant for $answer_P\ \langle G, \theta'\rangle$ for any $\theta' \in \Theta$. $\blacksquare$

Consider the program for *sum*, given in Section 2. Using *CHull* as the description domain, we can use this analysis to show that the removal optimizations in the example are allowed.

5.4 Reordering

To reorder addition and removal of constraints we must determine if a primitive constraint is definitely consistent for a goal and set of calls. The analysis is only for quasi-monotonic programs, as similarly to the redundancy analysis, the analysis use redundancy information to terminate the analysis with success because redundancy implies definite consistency. The new semantic function of importance is the consistency test

$$cons\ \theta\ \Theta \Leftrightarrow \theta \text{ is consistent with } \Theta.$$

Let P be a program and $ACons$ a constraint description. The *definitely consistent semantics* for P and $ACons$ has semantic functions:

$$\begin{array}{ll} \mathbf{con\_lit}_P & : \ Prim \to (Atom + Prim + Rem) \to ACons \to Bool \\ \mathbf{con\_cla}_P & : \ Prim \to Clause \to ACons \to Bool \\ \mathbf{con\_goal}_P & : \ Prim \to Goal \to ACons \to Bool. \end{array}$$

The semantic equations are

$$\begin{array}{lcl} \mathbf{con\_lit}_P\ \theta\ A\ \Pi & = & \bigwedge\{(\mathbf{con\_cla}_P\ \theta\ C\ \Pi) \mid C \in (defn_P\ A\ (vars\ \theta))\} \text{ when } A \in Atom \\ \mathbf{con\_lit}_P\ \theta\ \theta'\ \Pi & = & (Acons\ \theta\ (Aadd\ \theta'\ \Pi)) \text{ when } \theta' \in Prim \\ \mathbf{con\_lit}_P\ \theta\ rem(\theta')\ \Pi & = & true \text{ when } rem(\theta') \in Rem \\ \mathbf{con\_cla}_P\ \theta\ (A \leftarrow B)\ \Pi & = & (\mathbf{con\_goal}_P\ \theta\ B\ (restrict\ (vars\ A \cup vars\ \theta)\ \Pi)) \\ \mathbf{con\_goal}_P\ \theta\ nil\ \Pi & = & true \\ \mathbf{con\_goal}_P\ \theta\ A : B\ \Pi & = & (Ared\ \theta\ \Pi) \vee ((\mathbf{con\_lit}_P\ \theta\ A\ \Pi) \wedge (\mathbf{con\_goal}_P\ \theta\ B\ (\mathbf{ans\_lit}_P\ A\ \Pi))). \end{array}$$

where *Aadd*, *Arestrict*, *Ared* and *Acons* are required to approximate *add*, *restrict*, *red* and *cons* respectively.

Figure 3: The definitely consistent semantics for P and $ACons$

The semantic equations for the analysis are shown in Figure 3. Correctness of the analysis requires that the program is quasi-monotonic for all derivations encountered in the analysis. In our context this is not a problem as the analysis will only be applied to programs resulting from the removal optimization, and they have the necessary property.

Theorem 5.3 Let Θ be a set of constraints and G a goal. Let P be a program which is quasi-monotonic for the states $S = \{\langle G, \theta' \rangle \mid \theta' \in \Theta\}$. If $\Pi \propto \Theta$ and $(\mathbf{cons\_goal}_P\ \theta\ G\ \Pi)$ holds, then θ is consistent with $answers_P\ s$ for any state $s \in S$. ■

Now,

$$cons\ \theta\ \Theta \Leftarrow (satis\ \theta\ \Theta) \vee (red\ \theta\ \Theta)$$

where

$$satis\ \theta\ \Theta \Leftrightarrow \forall \theta' \in \Theta\ .\ \tilde{\exists}(\theta' \wedge \theta).$$

This observation leads to an analysis in which the set of constraint descriptions is *LComb* and *Acons* is defined by

$$\begin{array}{l} Acons\ \theta\ \langle \pi_{CHull}, \pi_{LSign} \rangle \Leftrightarrow \\ (Asatis\ \theta\ \pi_{LSign}) \vee (Ared\ \theta\ \pi_{CHull}) \end{array}$$

where *Asatis* and *Ared* approximate *satis* and *red* respectively. The definition of *Asatis* for the domain *LSign* is defined in terms of the abstract Fourier algorithm.

6 Another Example

The example program used in Section 2 is deliberately simple to illustrate the central ideas without difficulty. Correspondingly the advantages of the analysis are not enormous in this case because the interaction between the constraints of the program is limited. In this section we examine a more complex program that illustrates the effect of the optimizations where there is significant interaction between constraints.

Consider the following program MG that relates parameters in a mortgage

```
mg(P, T, I, R, B) ←                              (MG)
     T = 1 ∧
     B = P * (1 + I/1200) - R.
mg(P, T, I, R, B) ←
     T >= 2 ∧
     T1 = T - 1 ∧
     P1 = P * (1 + I/1200) - R ∧
     P1 >= 0 ∧
     mg(P1, T1, I, R, B).
```

The first clause of the program states that for a loan of length T = 1 month, the balance owing, B, is given by the principal, P, plus the addition of the interest (calculated from the annual interest rate I) minus the repayment R. The second clause states for loans of length greater than or equal to 2 months, the new principal is the old principal plus interest minus repayment, the new principal must be non-negative, and then what remains is a loan with new principal and one less month.

We consider the optimization of this program with respect to calls in which interest, repayment and balance are non-negative. That is we are interested in calls described by π, the convex hull description

$$I \geq 0 \wedge R \geq 0 \wedge B \geq 0.$$

Clearly this is the most common use of the program. Using *CHull* we can determine that

$$\begin{array}{l} \mathbf{ans\_goal}_P\ mg(P, T, I, R, B)\ \pi\ = \\ T \geq 1 \wedge P \geq 0 \wedge I \geq 0 \wedge R \geq 0 \wedge B \geq 0. \end{array}$$

This allows us to refine the program by adding constraints before the recursive call to mg. Each of the new constraints added is immediately redundant either with respect to the calling pattern or to the constraints in the clause. Hence refinement in this case produces no new constraints. Using the above knowledge about answers it is easy to determine that $T \geq 2$ and $P1 \geq 0$ are

Query qn	MG	SQn	Speed Up	SQ*	SQ* Speed Up
q1	0.129	0.129	1.00	0.129	1.00
q2	5.434	0.176	30.88	0.324	16.77
q3	6.279	0.188	33.40	0.362	17.34
q4	9.166	0.252	36.37	2.882	3.18

Table 2: Timings for the Optimized MG Program

both redundant after the recursive call, and hence can be removed. After removal optimization the resulting program is

```
mg(P, T, I, R, B) ←                    (REM)
     T = 1 ∧
     B = P * (1 + I/1200) - R.
mg(P, T, I, R, B) ←
     T >= 2 ∧
     T1 = T - 1 ∧
     P1 = P * (1 + I/1200) - R ∧
     P1 >= 0 ∧
     mg(P1, T1, I, R, B) ∧
     rem(P1 >= 0) ∧
     rem(T >= 2).
```

We now consider four different types of calls, each implying $I \geq 0 \wedge R \geq 0 \wedge B \geq 0$, and reorder the program separately for each

```
(q1) ?- B >= 0 ∧ mg(100000,360,12,1025,B).
(q2) ?- mg(P,360,12,1025,12625.9).
(q3) ?- R > 0 ∧ B >= 0 ∧ mg(P,360,12,R,B).
(q4) ?- B >= 0 ∧ B <= 1030 ∧
        mg(100000, T, 12, 1030, B).
```

We shall concentrate on q4 since it is where the most reordering is possible. Because in every call to mg T is either unconstrained or bounded from below, the analysis determines that the constraint $T \geq 2$ is always satisfiable when reached, hence the constraint can be moved later in the computation. It does not affect satisfiability of any of the constraints $T1 = T - 1$, $P1 = P*(1+I/1200)-R$, and $P1 \geq 0$ and is consistent with the recursive call because it is made redundant immediately. Thus it can be moved past the $rem(T \geq 2)$ literal and both such literals can be removed. Similarly the constraint $T1 = T - 1$ is consistent when reached for all calls. Again it is consistent across the goal

$$P1 = P*(1+I/1200)-R, P1 \geq 0, mg(P1, T1, I, R, B)$$

and so it can be moved after the recursive call. The constraint $P1 \geq 0$ is not always satisfiable when reached but it is weakly consistent across the recursive call, so its removal can be moved before the recursive call. The resulting program is

```
mg(P, T, I, R, B) ←                    (SQ4)
     B = P * (1 + I/1200) - R ∧
     T = 1.
mg(P, T, I, R, B) ←
     P1 = P * (1 + I/1200) - R ∧
     P1 >= 0 ∧
     rem(P1 >= 0) ∧
     mg(P1, T1, I, R, B) ∧
     T1 = T - 1.
```

For q1 the analysis determines that the literals $rem(T \geq 2)$ and $rem(P1 \geq 0)$ can be moved just after their respective constraints. In this case because T and $P1$ are ground when the constraints are reached there is no benefit in removing them as they will be executed as Boolean tests. In the case of q2 and q3 the analysis determines that $P1 \geq 0$ is always consistent and so can be removed altogether. The literal $rem(T \geq 2)$ can be moved just after the constraint $T \geq 2$ but again there is no benefit since T is ground.

Table 2 compares the execution time of the original program MG, with each of the specialized programs SQ1, SQ2, SQ3, SQ4, and with the program SQ* resulting from specializing with respect to all four queries simultaneously. In fact SQ* is just SQ1. Timings are CPU seconds for execution using CLP($\mathcal{R}$) v1.1 on a SparcStation 2. In this case we ignore the low-level optimizations since their effect is swamped by the effects of the high-level optimizations. The table shows order of magnitude improvements in the execution of all but the first query. However this is the query which would have been most helped by low-level optimization. It is interesting to note that the program SQ* resulting from optimizing with respect to all calls is obviously not as efficient as the individual optimized programs but it still shows order of magnitude improvements on the original. This indicates that, in this example, most of the speedup has resulted from synergy between removal and reordering which has allowed the removal of a constraint before an iterative call.

7 Conclusion

The combined optimizations and analyses give a powerful methodology for optimizing constraint logic programs based on reordering and removal of useless constraints. The optimizations have significant practical importance as they are automatizable and test results show they can give order of magnitude improvements in execution speed. Furthermore, though the specific details of the optimizations and analyses are for CLP languages, we feel that the underlying ideas of removing constraints when they become redundant and reordering of constraint addition and removal when this does not effect control flow, are applicable to the compilation of other constraint based programming languages.

References

[1] P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth Ann. ACM Symp. Principles of Programming Languages*, pages 238–252. Los Angeles, California, 1977.

[2] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. Sixth Ann. ACM Symp. Principles of Programming Languages*, pages 269–282. San Antonio, Texas, 1979.

[3] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Fifth ACM Symp. on Principles of Programming Languages*, 84–96, 1978.

[4] J.D. Darlington, Y.G. Guo, and H.P. Pull. A New Perspective on Itegrating Functional and Logic Languages. In *Procs. Fifth Generation Computer Systems 1992*, Tokyo, Vol. 2, 682–693. June 1992.

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

[6] D.B. Kemp, K. Ramamohanarao, I. Balbin, and K. Meenakshi. Propogating constraints in recursive deductive databases. In E. Lusk and R. Overbeek, editors, *Proc. First North American Conf. on Logic Programming*, 981–998, Cleveland, October 1989.

[7] M.M. Gorlick, C.F. Kesselman, D.A. Marotta, and D.S. Parker. Mockingbird: a Logical Methodology for Testing. *Journal of Logic Programming* 8:95–119, 1990.

[8] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Ann. ACM Symp. Principles of Programming Languages*, 111-119. San Francisco, California, 1987.

[9] 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, July 1992.

[10] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. An Abstract Machine for CLP($\mathcal{R}$). In *Proc ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 128–139, San Francisco, June 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, *Proc. 1991 Int. Symp on Logic Programming* 420–434, San Diego, October 1991.

[12] P.C. Kanellakis, G.M. Kuper, and P. Revesz. Constraint Query Languages. In *Proc. ACM Symp. on Principles of Database Systems*, 299–313, Nashville, April 1990.

[13] C. Lassez, K. McAloon, and R. Yap. Constraint Logic Programming and Options Trading. *IEEE Expert* 2:42–50, 1987.

[14] K. Marriott, L. Naish and J.-L. Lassez. Most Specific Logic Programs. In R. Kowalski and K. Bowen, editors, *Logic Programming: Proc. Fifth Int. Conf. Symp.* MIT Press, 1988.

[15] K. Marriott and H. Søndergaard. Analysis of Constraint Logic Programs. In *Proc. of the North American Conf. on Logic Programming*, 521–540, Austin, October 1990.

[16] V.J. Saraswat. *Concurrent Constraint Programming Languages.* Ph.D. Thesis, CMU. 1989. Also in *ACM Distinguished Dissertation Series.*

[17] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[18] H. Simonis and M. Dincbas. Using an Extended Prolog for Digital Circuit Design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, 165–188, Munich, October 1987.

[19] G.L. Steele and G.J. Sussman. Constraints. In *Procs APL79, ACM SIG-PLAN STAPL APL Quote Quad*, 9:208–225, June 1979.