Repeated Redundant Inequalities in Constraint Logic Programming*

Technical Report OSU-CISRC-6/94-TR31

Spiro Michaylov

Department of Computer and Information Science,
The Ohio State University,
228 Bolz Hall,
2036 Neil Avenue Mall,
Columbus, OH 43210-1277, U.S.A.,
Phone: +1 (614) 292 6377
FAX: +1 (614) 292 2911
`spiro@cis.ohio-state.edu`.

June 30, 1994

Abstract

A number of Constraint logic Programming systems, including CLP($\mathcal{R}$) and Prolog III, decide simultaneous linear inequalities as part of the fundamental operational step of constraint solving. While this can contribute tremendously to the usefulness of the systems, it is computationally quite expensive. Non-ground inequalities must generally be tested for consistency with the collected constraint set and then added to it, increasing its size, and thus making the next such test more expensive.

Future redundant inequalities in a program are those that are guaranteed to be subsumed after no more than one subsequent procedure call, usually in the context of a recursive procedure. It has been noted that such inequalities need only be tested for consistency with the current constraint set, thus resulting in dramatic savings in execution speed and space usage. In this paper we generalize the notion of future redundancy in a number of ways and thus broaden its applicability. Thus we show how to dramatically improve the performance of a wider class of programs that rely heavily on linear inequalities.

*This research is supported in part by the National Science Foundation Grant #CCR-9308631. An abridged version of this paper appears in the proceedings of the 1st International Symposium on Constraints in Computational Logics, Munich, September 1994.

1 Introduction

A number of Constraint logic Programming systems, including CLP($\mathcal{R}$) and Prolog III, decide simultaneous linear inequalities as part of the fundamental operational step of constraint solving. While this can contribute tremendously to the usefulness of the systems, it is computationally quite expensive. Non-ground inequalities must generally be tested for consistency with the collected constraint set and then added to it, increasing its size, and thus making the next such test more expensive.

We will discuss this problem in the context of CLP($\mathcal{R}$) [2], a constraint logic programming language dealing with real arithmetic, although the results translate directly to other languages that decide linear inequalities. A locally optimizing compiler for CLP($\mathcal{R}$), utilizing an abstract code interpreter, was described by Jaffar *et al*, in [1]. Global optimization techniques for CLP($\mathcal{R}$) and related languages have been discussed by Jørgensen *et al* [3], Marriott and Søndergaard [4], Marriott and Stuckey [5], and McDonald *et al* [6]. For an overview, see [7]. Broadly speaking, these optimizations rely on the use of global analysis techniques such as abstract interpretation to infer information about the state of variables involved in arithmetic constraints, given information about allowed query patterns. Some of the techniques, such as the future redundancy optimization in [3], also rely on analyzing the patterns of constraint invocation for classes of queries, and observing how the constraints interact with each other. Furthermore, in order to expose opportunities for applying these optimizations, it may be necessary to use multiple specialization, most fully described by Winsborough in [9], and reordering of subgoals, described by Marriott and Stuckey in [5], and studied systematically by Michaylov [8].

The notion of future redundancy for arithmetic inequality constraints was first defined by Jørgensen *et al* [3]. Future redundant inequalities in a program are those that are guaranteed to be subsumed after no more than one subsequent procedure call, usually in the context of a recursive procedure. Inequalities must generally be tested for consistency with the collected constraint set and then added to it. However, future redundant inequalities need only be tested for consistency, thus resulting in dramatic savings in execution speed and space usage. This technique is closely related to the constraint removal technique of Marriott and Stuckey [5]. It is less frequently applicable, but results in a greater efficiency improvement when it applies because it is static, and avoids adding the constraint to the collected set.

In this paper, we consider the essential features of a program that lead to the repeated uniform subsumption of linear inequalities. Thus we define a broad class of programs that lead to such subsumption, show that this class is important in practice, and consider how such programs can be optimized. We show that the future redundancy optimization due to Jørgensen *et al* is not sufficiently general to optimize this class of programs, and that more general notions of future redundancy are needed. We define the notions of

- *partial future redundancy*
  which relaxes the subsumption requirement for procedure calls matching base cases;
- *return future redundancy*
  dealing with constraints that follow a recursive call; and
- *multiple future redundancy*
  dealing with the subsumption of multiple constraints at each step.

We then describe optimizations, in the form of source-to-source transformations, taking advantage of these properties. Thus our contribution may be viewed in two ways, respectively from the viewpoint of programming methodology and language implementation:

- We identify a potentially important programming technique as being "safe" in the sense that the resulting programs do not suffer from the inefficiency that had been expected.
- We identify a procedure-level optimizing transformation for programs that can result in dramatic performance improvement, and have developed a good understanding of why and how the transformation is applicable in real programs.

Finally, we note that the optimizations described can often only be applied if certain basic assumptions can be made about the instantiation status or even value range of certain program variables. We show how multiple specialization can be used to establish the validity of these assumptions, broadening the applicability of the optimizations.

The remainder of this paper is organized as follows. We examine the connection between static program structure, calling patterns and the run-time subsumption of linear inequalities in Section 2. We make some preliminary definitions in Section 3. In Section 4, we redefine the original notion of future redundancy in our context, and restate the corresponding optimization and its correctness. Then in Section 5 we introduce the notion of partial future redundancy, and its corresponding optimization, illustrate it and again establish its correctness. Return future redundancy is introduced in the same manner in Section 6, and multiple future redundancy in Section 7. The importance and utility of multiple specialization for utilizing these optimizations in practice are described in Section 8. Section 9 gives details of performance improvement for a number of examples. Finally, in the appendices we give an introduction to CLP($\mathcal{R}$), provide outlines of proofs of main results in the paper, and discuss example programs in detail.

2 Programs, Queries and Redundant Inequalities

We begin by developing an understanding of a class of programs and queries that lead to linear inequalities being systematically subsumed at runtime. A very wide range of programs can lead to a series of subsumptions at each derivation step, with potential performance improvement if these constraints were handled differently. However, we seek to develop broadly applicable compiler optimizations that can more-or-less be guaranteed to produce an improvement when applied, and so will have to restrict our scope to programs that guarantee a well-behaved sequence of mutually redundant constraints.

We seek programs and queries that repeatedly establish a non-ground linear inequality associated with each call that is always subsumed by constraints resulting from the immediately following call. This ensures that simply testing the subsumed inequality rather than adding it to the constraint solver will not affect the control flow. Now let us consider some very simple ways to generate such a sequence of constraints. Perhaps the most obvious is the following loop, called with the second argument free.

```
p(X, Y, Z) :-
        X > 1,
        NY = Y + 20,
        NY < 10000,
        p(X - 1, NY, Z).
```

Each of the inequalities on `NY` will be subsumed by the following one in conjunction with intervening equations. That is,

$$NY' < 10000 \wedge NY' = Y + 20 \wedge Y = NY \Rightarrow NY < 10000,$$

where we use NY' for the second instance of NY. Thus we would like to simply test the consistency of the inequality $NY < 10000$ with the solver state, without *adding* it to the solver state. We must check, however, that doing so could not affect program behavior. A recursive call always follows the constraint immediately. If the recursive call uses the same rule, the constraint is immediately subsumed. However, if there are other rules for p/3, the computation could proceed erroneously. Thus, we require that *each* of the other rules for p/3 also has, before the first subsequent call, a conjunction of constraints that subsumes `NY < 10000`. This was defined as the *future redundancy* condition by Jørgensen *et al* [3]. It requires, in particular, that every base case also results in the constraint being subsumed. For example, the rule

```
p(0, W, W) :-
        W < 5000.
```

would be acceptable, as $NY = W \wedge W < 5000 \Rightarrow NY < 10000$, as would

```
p(0, 0, 0).
```

since $NY = 0 \Rightarrow NY < 10000$, while the rule

```
p(0, W, W).
```

would not. The future redundancy condition captures correctly an important class of constraints that can be tested without being added to the constraint solver. However, if we study the class of linear recursive procedures we will see that there are other important cases of constraints being systematically subsumed, perhaps with exceptions at boundary conditions, that are not captured by this definition.

First, consider constraints that will be subsumed by a call to any recursive rule, but not to some base cases. We will call these partially future redundant. A successful, finite computation using a linear recursive procedure makes use of exactly one of the base cases of that procedure exactly once, although, of course, many may have been used multiple times in backtracking. This means that at most one instance of any partially future redundant constraint was tested when indeed it should have been added to the constraint set. This was the last instance before the use of the base case. All other instances of the constraint actually *were* subsumed, so it was appropriate for them to be tested without being added. Consider, from before, the example:

```
p(0, W, W).
p(X, Y, Z) :-
        X > 1,
        NY = Y + 20,
        NY < 10000,
        p(T - 1, NY, Z).
```

If called with the first argument ground and second argument free, every use of the recursive rule will establish an inequality constraint that will be subsumed by the following use of that rule, except for the last use before the base case. Thus, for example, the query `?- p(10, U, V)` should result in the answer constraint

$$V = U + 200 \wedge V < 10000$$

as the last inequality must be added to the solver. As we shall see in Section 5, this is easy to achieve by a source to source transformation, even for more general cases where different recursive rules establish different inequalities, and some of the base cases subsume these inequalities while others do not. In the appendices, we will see that partially future redundant constraints are important. We will also see how programs can be transformed systematically to obtain the performance improvement of future redundancy, while still behaving correctly.

Second, consider the following procedure:

```
p(0, W, W).
p(X, YN, Z) :-
        X > 1,
        p(X - 1, Y, Z),
        NY = Y + 20,
        NY < 10000.
```

When called with the second argument free and third argument ground, a chain of inequalities is established on NY. Each of these is subsumed immediately after the return from the body of the rule, except for that in the outer most use of the rule. In this particular example, the subgoals could simply be reordered to place the inequality before the recursive call, but there exist more complicated examples where that is not true. The associated optimization is described in Section 6.

Finally, consider the possibility of more than one constraint being subsumed at each call. We can generate a very simple example:

```
p(0, X, Y, X, Y).
p(N, X, Y, U, V) :-
        N > 1,
```

```
X < 100,
Y > 10,
X1 = X + 2,
Y1 = Y - 3,
p(N - 1, X1, Y1, U, V).
```

Such programs can be useful for simulating a system with multiple state variables. It may at first seem that the partially future redundant inequalities on X and Y can be dealt with in isolation, but this is not the case. If we consider a query with the second and third arguments related by a linear constraint, we will see that it is necessary for the two inequalities to be tested for consistency *together*. We will consider this in more detail in Section 7.

Now let us consider the nature of the programs and queries that we have identified. The programs have tended to use inequalities to determine whether a monotonically increasing or decreasing parameter has reached a certain bound. Clearly for certain calling patterns these inequalities would have been executed as ground tests, and redundancy would not be an issue. However, these programs have tended to have sensible alternate uses, and these have resulted in redundancy. We shall see later that it is not unusual for different calling patterns to result in different inequalities in a program being executed as respectively ground tests or non-ground future redundant constraints. Furthermore, our examples have employed only linear recursion, in the sense that rules contain no more than one recursive call. The optimizations presented here can be generalized to handle multiple recursive calls, but it does not seem to be useful: only linear recursive procedures seem to result in large scale, systematic mutual subsumption of inequalities.

In summary, we see that linear recursive procedures are a rich source of mutually redundant linear inequalities for certain kinds of queries, because linear inequalities in the recursive rules tend to be subsumed by others either in the next call or after the next return. We also see that it is not too difficult, in general, to take advantage of this property. In the appendices we will see that such procedures are useful in practice, and in the remaining sections we formally define the appropriate redundancy conditions and present the optimizing transformations.

3 Preliminary Definitions

We will frequently need to describe templates for specific classes of CLP($\mathcal{R}$) procedures and rules. We assume all programs to be in canonical form, in the sense that only variables appear as arguments of predicates in the body or head. We typically refer to head arguments as a vector $\tilde{X}$ of variables, and to those of a recursive call to the procedure as the vector $\tilde{Y}$. We use $\tilde{C}$, $\tilde{D}$, $\tilde{E}$, ... for sequences of constraints and atoms in the body of a rule, ρ for a rule, and π for individual constraints.

Now we define the class of procedures that we wish to optimize.

Definition 1 (Linear Recursive Procedure) *A linear recursive procedure P is a sequence of rules of the form*

$$p(\tilde{X}) \text{ :- } \tilde{A}, \tilde{B}$$

where $\tilde{A}$ is a sequence of constraints, and $\tilde{B}$ is a sequence of atoms and constraints that is either empty or begins with an atom. Each rule is either:

1. a base case:
   None of the atoms may result in a recursive call to P.

2. a recursive case*:*
   Exactly one of the atoms is a recursive call to P but no other atom may result in such a call indirectly.

4 Future Redundancy

Here we present the definition of future redundancy in a somewhat different manner from that in [3]. It is more general in the sense that the redundant constraint need not appear before the first atom in the body.

```
p(X, Y, Z) :-                    p(X, Y, Z) :-
        X <= 0,                          X <= 0,
        Y < 500.                         Y < 500,
        Z = Y.                           Z = Y.
p(X, Y, Z) :-                    p(X, Y, Z) :-
        X > 0,                           X > 0,
        NY = Y + 20,                     NY = Y + 20,
        NY < 1000,                       test(NY < 1000),
        p(X - 1, NY, Z).                 p(X - 1, NY, Z).
```

Figure 1: Future redundancy in a summation loop, and optimized program.

On the other hand, it is restricted to linear recursive procedures. We begin with a minor definition of a term for the constraints that result from a single procedure call and the immediately surrounding constraints in both the calling and called rules.

Definition 2 (Derived Constraint) *Let A be an atom and ρ be the rule H :- $\tilde{C}, \tilde{D}$, where $\tilde{C}$ is a sequence of constraints, and $\tilde{D}$ is a sequence of atoms and constraints. Then $Der(A, \rho) \equiv (A = H) \wedge \tilde{C}$ is their* derived constraint*: the result of reducing subgoal A using rule ρ.*

Definition 3 (Future Redundancy) *Let*

$$p(\tilde{X}) \text{ :- } \tilde{C}, \pi, \tilde{D}, p(\tilde{Y}), \tilde{E}$$

be a rule in a linear recursive procedure P, where $\tilde{C}$ and $\tilde{E}$ are sequences of constraints and atoms, π is a constraint, and $\tilde{D}$ is a sequence of constraints. Constraint π is future redundant *in that rule if, for every rule $\rho \in P$,*

$$Der(p(\tilde{Y}), \rho) \wedge \tilde{D} \Rightarrow \pi$$

As an example, consider the constraint $NY < 1000$ in the simple summation program on the left of Figure 1. For both rules, the derived constraints imply this constraint, as required. Next, we show how to systematically transform such a program so that the collected set of inequality constraints does not grow with each iteration. We begin by extending the operational model to cope with two meta-level predicates on constraints: `test/1` and `add/1`. The former checks that a constraint is consistent with the collected constraint set, but does not add it to that set. The latter assumes that it is consistent and adds it to the collected constraint set,

Definition 4 (Execution of `test/1` and `add/1`) *The CLP($\mathcal{R}$) operational model is augmented with the following two transitions:*

1. *The goal C ?- $test(\pi), \tilde{G}$ reduces to the goal C ?- $\tilde{G}$ if $C \cup \{\pi\}$ is satisfiable.*
2. *The goal C ?- $add(\pi), \tilde{G}$ reduces to the goal $C \cup \{\pi\}$?- $\tilde{G}$.*

Transformation 1 (Future Redundancy Optimization) *For any linear recursive procedure, replace each future redundant constraint π with the atom $test(\pi)$.*

The result of applying this transformation to the program on the left of Figure 1 is shown on the right of that figure. The correctness of this transformation may then be stated as in [3].

Proposition 1 (Correctness of Future Redundancy Optimization) *There is a one-to-one correspondence between (possibly partial) derivations for any program and that program after the future redundancy transformation, in which derivation length is preserved. Furthermore, in the case of successful derivations, the answer constraints are equivalent, modulo variable renaming.*

```
p(X, Y, Z) :-
        X <= 0,
        Z = Y.
p(X, Y, Z) :-
        X > 0,
        NY = Y + 20,
        NY < 1000,
        p(X - 1, NY, Z).
```

```
p(X, Y, Z) :-
        p(X, Y, Z, a).

p(X, Y, Z, a) :-
        X <= 0,
        Z = Y.
p(X, Y, Z, b) :-
        X <= 0,
        add(Y < 1000),
        Z = Y.
p(X, Y, Z, _) :-
        X > 0,
        NY = Y + 20,
        test(NY < 1000),
        p(X - 1, NY, Z, b).
```

Figure 2: Summation loop that is not quite future redundant, and optimized loop.

In [3], the notion of future redundancy with respect to a given calling pattern is also defined. The definitions in this paper can be modified similarly. We should also note that some other global optimizations, possibly based on multiple specialization, subgoal reordering, or even some unfolding of simple calls may be helpful in exposing some cases of future redundancy. Conversely, the desire to expose such opportunities may motivate the application of some of the other optimizations. We will discuss some aspects of this later.

5 Partial Future Redundancy

Consider the simple summation loop in Figure 2. The first argument controls how many times the number 20 is added to the third argument, and the third argument is used to produce the result, with the requirement that the running total must always be less than 1000. That is, the call $p(N1, N2, T)$ produces $T = N2 + 20 * N1$ provided that $N1 \geq 0$ and $N2 + 20 * N1 < 1000$. The loop is almost future redundant. In the recursive case, we can easily see that $NY + 20 < 1000 \Rightarrow NY < 1000$ gives us the required subsumption. However, the base case provides no such subsumption. If the usual future redundancy optimizations were carried out, the program would be incorrect for certain queries. Consider, for example, ?-$p(10, Y, Z)$, which should succeed with answer constraints $Z = Y + 200, Z < 1000$. In fact, the inequality constraint shown is one of 10 that would be accumulated, but this one subsumes all the others, as expected. The future redundancy optimization would have resulted in no inequalities being added, and thus only the equation would be collected. The tests would all succeed, because of the degree of freedom provided by the uninstantiated second argument. Of course, the query ?-$p(10, Y, Z), Y > 1000$ would then succeed incorrectly.

Now, if we consider this example, we see that it *is* appropriate only to test each of the inequalities, and to add the *last one*. Thus, the program on the left of Figure 2 should be transformed into that on the right. This version of the loop provides most of the efficiency benefits of future redundancy, but maintains correctness by adding the last of the tested inequalities when it bottoms out. However, note that some bureaucracy, including an extra base case, is needed to make sure that it works correctly in the boundary case: when no recursion takes place.

Motivated by the above discussion, we now wish to define the above class of constraints as *partially future redundant.*

Definition 5 (Partial Future Redundancy) *Let*

$$p(\tilde{X}) \text{ :- } \tilde{C}, \pi, \tilde{D}, p(\tilde{Y}), \tilde{E}$$

be a rule in a linear recursive procedure P, where $\tilde{C}$ and $\tilde{E}$ are sequences of constraints and atoms, π is a constraint, and $\tilde{D}$ is a sequence of constraints. Constraint π is partially future redundant *in that rule if, for*

```
p(X, Y, Z) :-                   %% 1st base case
        X <= 0,
        Z = Y.
p(X, Y, Z) :-                   %% 2nd base case
        X = 1,
        Z = Y - 100.
p(X, Y, Z) :-                   %% 1st recursive case
        X > 10,
        NY = Y + 10,
        NY < 2000,              %% partially future redundant
        p(X - 1, NY, Z).
p(X, Y, Z) :-                   %% 2nd recursive case
        X > 1,
        X < 10,
        NY = Y + 5,
        NY < 2003,              %% partially future redundant
        p(X - 1, NY, Z).
```

Figure 3: Loop with two recursive rules

every rule $\rho \in P$ that is a recursive case,

$$Der(p(\tilde{Y}), \rho) \wedge \tilde{D} \Rightarrow \pi.$$

This definition differs only subtly from that of future redundancy: ρ now ranges over all recursive cases in the procedure, rather than all rules. However, the transformation required differs quite substantially. We begin by considering a more complicated example than that of Figure 2. Consider the unoptimized program in Figure 3, with multiple recursive rules and multiple base cases. Note that the constraints $NY < 2000$ and $NY < 2003$ in the recursive rules are partially future redundant. In Figure 4 we show the optimized version of the program. Notice that when a base case is reached it is necessary to determine which of the two constraints was tested last, because they were different. Thus, the number of base cases in the optimized procedure is the product of the number originally and the number of recursive rules plus one. However, no recursive rules are added. An additional argument has been added to keep track of which recursive rule was last used. In those cases where the variables needed when the last inequality is added are not naturally available in the base case, they can also be passed along in this argument. Consider, for example, the two rules shown in Figure 5. The variable NY is not easily available in the base case, and must thus be passed through. The optimized version is shown immediately below.

We should also consider the possibility that there will be recursive rules that do not contain a partially future redundant constraint. This case is dealt with easily: the non-specialized version of the base case is used.

We now describe the optimizing transformation for linear recursive procedures containing partially future redundant constraints.

Transformation 2 (Partial Future Redundancy Optimization) *A linear recursive procedure p/n containing partial future redundant constraints becomes $p/(n+1)$, where the extra argument is used for passing around administrative information. We assume the existence of a set of unique constants not already present in the program, denoted by variously subscripted c.*

1. *Add as a wrapper the new rule $p(\tilde{X})$:- $p(\tilde{X}, c_0)$.*

2. *For every base case: add as the $(n+1)th$ head argument the term c_0. Call the resulting rules the original base cases.*

3. *For every recursive rule ρ:*

```
p(X, Y, Z) :-                 %% wrapper
        p(X, Y, Z, a).        %% set up for boundary case

p(X, Y, Z, a) :-              %% 1st base, no recursion (a)
        X <= 0,
        Z = Y.
p(X, Y, Z, b) :-              %% 1st base, 1st recursive (b)
        X <= 0,
        Z = Y,
        add(Y < 2000).        %% add without testing
p(X, Y, Z, c) :-              %% 1st base, 2nd recursive (c)
        X <= 0,
        Z = Y,
        add(Y < 2003).        %% add without testing
p(X, Y, Z, a) :-              %% 2nd base, no recursion (a)
        X = 1,
        Z = Y - 100.
p(X, Y, Z, b) :-              %% 2nd base, 1st recursive (b)
        X = 1,
        Z = Y - 100,
        add(Y < 2000).        %% add without testing
p(X, Y, Z, c) :-              %% 2nd base, 2nd recursive (c)
        X = 1,
        Z = Y - 100,
        add(Y < 2003).        %% add without testing
p(X, Y, Z, _) :-              %% 1st recursive: ignore special arg
        X > 10,
        NY = Y + 10,
        test(NY < 2000),      %% partially future redundant
        p(X - 1, NY, Z, b).   %% set special argument
p(X, Y, Z, _) :-              %% 2nd recursive: ignore special arg
        X > 1,
        X < 10,
        NY = Y + 5,
        test(NY < 2003),      %% partially future redundant
        p(X - 1, NY, Z, c).   %% set special argument
```

Figure 4: Optimized loop with two recursive rules

```
p(0, _, Y, Y).
p(N, W, Y, Z) :-
        NY = N + 10,
        NY < 1000,
        p(N - 1, W + N, NY + W, Z).

p(N, W, Y, Z) :-
        p(N, W, Y, Z, a(_)).

p(0, _, Y, Y, a(_)).
p(0, _, Y, Y, b(ONY)) :-
        add(ONY < 1000).
p(N, W, Y, Z, _) :-
        NY = N + 10,
        test(NY < 1000),
        p(N - 1, W + N, NY + W, Z, b(NY)).
```

Figure 5: Partial future redundancy with variable passed for base case

(a) *Add as $(n+1)$th head argument the anonymous variable (_).*

(b) *For the recursive call:*

  i. *If there is no associated partially future redundant constraint, add c_0 as the $(n+1)$th argument. Otherwise, add as the $(n+1)$th argument the term $c_\rho(\tilde{V}_\rho)$, where $\tilde{V}_\rho$ consists of the variables in the partially future redundant constraint corresponding to that call.*

  ii. *If there is a partially future redundant constraint π_ρ, replace it with test(π_ρ), and add, immediately after each original base case, a new version with $(n+1)$th head argument $c_\rho(\tilde{W})$ where $\tilde{W}$ is a unique renaming of $\tilde{V}_\rho$, and with added subgoal add$(\pi_{\tilde{W}})$ before any atoms in the rule.*

If the original procedure had r recursive rules, with a total of s recursive calls with associated partially future redundant constraints, and b base cases, the transformed procedure has one wrapping rule, r recursive rules, and $b(s+1)$ base cases. However, only b base cases have any chance of unifying with any given call to the procedure, because of the effect of the $(n+1)$th argument. Hence, some relatively minor local optimizations such as indexing can be used to ensure that the overheads of the transformed program are not excessive, provided that more than a few iterations will be taken.

Proposition 2 (Correctness of Partial Future Redundancy Optimization) *As before.*

Thus we have a powerful tool for optimizing procedures containing this more general class of redundant inequality constraint.

6 Return Future Redundancy

We now turn our attention to those recursive procedures where the inequality is located after the recursive call. In most programs, such inequalities can reasonably be moved ahead of the call, but there are cases where this would be unreasonable: for example, if the inequality was thus made nonlinear. Such inequalities can be considered, in a sense, future redundant if they are subsumed immediately after the return from a call in which they are encountered. The summation loop in Figure 6 is a contrived example for illustration, as this ordering of constraints is not really necessary. Each of the inequalities on Z is subsumed by the one that follows it chronologically. Now of course this situation is analogous to partial future redundancy, since the

```
p(X, Y, Z) :-
        X <= 0,
        Z = Y.
p(X, Y, Z) :-
        X > 0,
        p(X - 1, Y, T),
        Z = T + 20,
        Z < 1000.              %% return future redundant constraint
```

Figure 6: Summation loop version that is return future redundant

```
p1(X, Y, Z, a) :-              %% extra arg identifies case
        X <= 0,
        Z = Y.                 %% no redundant constr in base case
p1(X, Y, Z, b) :-              %% extra arg identifies case
        X > 0,
        p1(X - 1, Y, T, _),    %% no need to determine which case:
        Z = T + 20,            %%    will now be subsumed
        test(Z < 1000).        %% only test redundant constraint

addlast(a, Z).                 %% no redundant constr in base case
addlast(b, Z) :-               %% X < 1000 redundant in recursive
        add(Z < 1000).         %%    case

p(X, Y, Z) :-                  %% wrapper:
        p1(X - 1, Y, T, U),    %%
        addlast(U, Z).         %% constraint in outermost rule
                               %%    not subsumed
```

Figure 7: Optimized return future redundant summation loop

last instance of the constraint will most likely *not* be subsumed. Hence, the optimized form of this program is as shown in Figure 7.

Additionally, we should note that an important special case of such redundancy is where there are no atoms after any recursive call (a weak form of tail recursion). Then it is debatable whether it is worth even testing the redundant constraint: the summarizing constraint could simply be placed in the wrapper. The overhead of testing the constraints would be saved, with the additional cost of having to return all the way before failing when failure should have been triggered by a redundant constraint at some intermediate step.

In the following we define a notion of derivation step corresponding to that of having collected additional constraints after returning from a procedure call, but ignoring one constraint in the call (π), which corresponds to the redundant constraint. Note that all other constraints of the called and calling rule are used.

Definition 6 (Return-Derived Constraint) *Let ρ_π and ρ_r be among the rules defining a relation p, such that:*

1. *ρ_π contains a constraint π in its body after the last atom, and has head arguments $\tilde{Y}$;*
2. *$\tilde{C}_\pi$ is the sequence of all constraints in the body of ρ_π other than π;*
3. *in rule ρ_r,*
   (a) *$\tilde{C}$ is the sequence of constraints in ρ_r preceding the recursive call;*
   (b) *the arguments of the recursive call are $\tilde{X}$,*
   (c) *$\tilde{D}$ is the sequence of all constraints after the recursive call.*

Then, the return-derived constraint of rules ρ_π and ρ_r is:

$$Ret(\rho_\pi, \rho_r) = \tilde{C}_\pi \wedge (\tilde{X} = \tilde{Y}) \wedge \tilde{C} \wedge \tilde{D}.$$

Now using this we may define what it means for a constraint to be made redundant immediately after a call to the rule containing it terminates.

Definition 7 (Return Future Redundancy) *Let P be a linear recursive procedure. The constraint π is* return future redundant *in the rule ρ_π of P if, for each recursive rule ρ_r in P, $Ret(\rho_\pi, \rho_r) \Rightarrow \pi$.*

Then we may define a corresponding optimization for linear recursive procedures containing return future redundant constraints. Note that in these definitions there are assumed to be no return future redundant constraints in base cases, as they do not lead to important optimizations.

Transformation 3 (Return Future Redundancy Optimization) *A linear recursive procedure p/n containing return future redundant constraints becomes a wrapper defining p/n, a procedure $p/(n+1)$, where the extra argument is used for passing administrative information, and a tidy-up procedure $addlast/1$, defined as follows:*

1. *Add as a wrapper the new rule $p(\tilde{X})$:- $p(\tilde{X}, Z), addlast(Z)$.*
2. *For every base case: add as the $(n+1)th$ head argument the term c_0.*
3. *Add a rule $addlast(c_0)$ to handle case where only a base case was used, and hence no return future redundant constraints were encountered.*
4. *For every recursive rule ρ:*
   (a) *Add as $(n+1)th$ head argument the term $c_\rho(\tilde{V}_\rho)$, where $\tilde{V}_\rho$ consists of the variables in the return future redundant constraint in the body.*
   (b) *Replace the return future redundant constraint π_ρ by $test(\pi_\rho)$.*

```
p(0, 1, 1).
p(N, X, Y) :-
        N > 0,
        X > 1, Y > 1,            % both are future redundant
        p(N - 1, X - 2, Y - 3).
```

Figure 8: Loop with multiple future redundant constraints

```
p(0, 1, 1).
p(N, X, Y) :-
        N > 0,
        test( ( X > 1, Y > 1) ), % two constraints tested together
        p(N - 1, X - 2, Y - 3).
```

Figure 9: Optimized loop with multiple future redundancy

(c) Add the rule

$$addlast(c_\rho(\tilde{V}_\rho)) \text{ :- } add(\pi_\rho).$$

Fortunately, this optimization does not cause as much code explosion as the previous one, and the same peephole optimizations as before are still applicable. The case where some of the recursive rules do not contain return future redundant constraints is not mentioned explicitly, but can trivially be handled as before.

Proposition 3 (Correctness of Return Future Redundancy Optimization) *As before.*

It is interesting to observe that both partial future redundant and return future redundant constraints may occur in one linear recursive procedure. In that case it is easy to see that the two optimizations may be combined.

7 Multiple Future Redundancy

So far we have only considered cases of (partial) future redundancy where there is at most one redundant constraint associated with each recursive call in a rule. However, it is possible for a rule to contain multiple redundant constraints associated with one recursive call: for example, when different variables are being tested. Consider the program in Figure 8, together with the query

```
?- X + Y < 0, p(10, X, Y)
```

which should immediately lead to failure. If the two redundant constraints are tested separately, failure does not occur until the base case is reached. Hence, we see that it is not sufficient to treat such constraints as we have done so far. This is because we must check the *mutual* satisfiability of the redundant constraints: the first constraint has not yet been subsumed when we test the second. This can be handled by extending the *test*/1 predicate to take conjunctions of constraints, as shown in Figure 9. Then failure is detected before the first recursive call, as desired.

We should note that the example deals with future redundancy, but the technique is directly applicable to both partial future redundancy and return future redundancy. In the case of partial future redundancy, the augmented base cases must also include all of the redundant constraints associated with the appropriate recursive call.

```
p(X, Y, I, Z) :-
        X <= 0,
        Z = Y.
p(X, Y, I, Z) :-
        X > 0,
        NY = Y + I,          %% is I negative, 0 or positive?
        NY < 1000,           %% this is pfr if I is non-negative
        p(X - 1, NY, I, Z).
```

Figure 10: Summation loop needing multiple specialization

8 Multiple Specialization

Now let us consider how the optimizations we have defined would be used in practice by a compiler. We assume that global type and mode analysis is performed, and that some indication of calling patterns at some level has been supplied by the programmer. The optimizations described here are only applicable to certain procedures for certain calling patterns. The usual way to deal with such situations is multiple specialization: generating separate code for different versions of the same procedure, to be selected variously at compile time, link time or run time according to the nature of a call to the procedure. Recursive procedures tend to call themselves back with a pattern similar to that with which they were called, so even run time selection can lead to optimizations. Many factors are considered to determine whether a compiler will split a given procedure in this way, and the applicability of these future redundancy optimizations would become yet another decision criterion.

Multiple specialization has an additional role to play in applying our optimizations. We find that frequently it is not easy to establish future redundancy properties, usually because not enough is known about the status of variables involved. For example, in Figure 10, the constraint $NY < 1000$ is clearly partially future redundant if I is non-negative, but otherwise not. This is because $Y + I < 1000 \rightarrow Y < 1000$ iff $I \geq 0$. Notice, furthermore, that I is unchanged throughout the recursion. Hence, it is quite easy to split the program into two cases outside the recursion. If I is negative, the original version of the program is called (although in fact other optimizations are applicable to that version, as the inequality becomes redundant after the first iteration). Otherwise a version incorporating the partial future redundancy optimization can be called. This is illustrated in Figure 11. Note that this is really only useful if the argument corresponding to I is expected to be ground in the call.

The optimizations described in this paper are being incorporated into a globally optimizing CLP($\mathcal{R}$) compiler currently being developed.

9 Empirics

These were performed on a SPARC ELC with 64 MB of memory, using an instrumented variant of Version 1.1 of IBM's CLP($\mathcal{R}$) compiler. They demonstrate just how dramatic this set of optimizations can be when applicable. For each program and query we see the CPU time before and after the optimization, and the number of solver nodes before and after the optimization, with factors of improvement. Note that only source-to-source transformations were used for the optimization: the $test/1$ predicate was simulated using separate rules involving cuts and failure, much like using double negation-as-failure.

```
p(X, Y, I, Z) :-                %% 1st case of split
        I >= 0,
        p0(X, Y, I, Z).         %% call pfr version
p(X, Y, I, Z) :-                %% 2nd case of split
        I < 0,
        p2(X, Y, I, Z).         %% call simply redundant version

p0(X, Y, I, Z) :-               %% pfr wrapper
        p1(X, Y, I, Z, a).

p1(X, Y, I, Z, a) :-            %% pfr special base case
        X <= 0,
        Y < 1000,
        Z = Y.
p1(X, Y, I, Z, b) :-            %% pfr regular base case
        X <= 0,
        add(Y < 1000),
        Z = Y.
p1(X, Y, I, Z, _) :-            %% pfr recursive case
        X > 0,
        NY = Y + I,
        test(NY < 1000),
        p1(X - 1, NY, I, Z, b).

p2(X, Y, I, Z) :-               %% simply redundant version
        X <= 0,
        Z = Y.
p2(X, Y, I, Z) :-
        X > 0,
        NY = Y + I,
        NY < 1000,              %% only needed once
        p3(X - 1, NY, I, Z).    %% no inequality after 1st iteration

p3(X, Y, I, Z) :-               %% no inequality version
        X <= 0,
        Z = Y.
p3(X, Y, I, Z) :-
        X > 0,
        NY = Y + I,
        p3(X - 1, NY, I, Z).    %% recurse locally
```

Figure 11: Summation loop multiply specialized

Program	Query	Time (sec) Before	After	(Factor)	Solver Space Before	After	(Factor)
Tree	Search	5.71	1.11	(5)	52622	2142	(25)
Simulation (UV)		0.38	0.08	(5)	3564	246	(14)
Simulation (MV)		0.75	0.14	(5)	7044	408	(17)
Mortgage	1	8.34	0.43	(21)	66781	1082	(62)
	2	6.89	0.46	(15)	64617	1429	(45)

The programs and queries used were as follows:

- *Tree*
  A binary search tree lookup, as described in Appendix C, where each node traversed contributes to a cost relative to an initial cost, and the cost is bounded above. There are two recursive rules and one base case.

- *Simulation*
  These programs and queries simulate the behavior of a continuous system numerically, as described in Appendix D. The state variables are bounded above and increase monotonically. The first program has a univariate state, while the second has multivariate state. There is one recursive rule and one base case, and the second program contains two partially future redundant constraints, which must be tested together.

- *Mortgage*
  This is a compound interest calculation program that has become a popular CLP($\mathcal{R}$) benchmark. The program and its optimization are described in detail in Appendix E, and the two queries described there were used. The first results in partial future redundancy, while the second results in future redundancy. The program has one base case and one recursive rule.

For these results, the rules of the program were re-ordered to isolate the direct effects of the optimization. This tends to double or triple the improvements in performance.

Conclusion

We have introduced two important optimizations removing redundant linear inequalities at compile time. It is clear that, where applicable, these optimizations will cause significant performance improvement. We have also made considerable progress towards expanding their applicability. We expect that the long-term effectiveness of these optimizations depends on further improved understanding of CLP($\mathcal{R}$) programming methodology and applications, as well as improved techniques for controlling multiple specialization in practical compilers.

Acknowledgments

The author is grateful to Nathan Loofbourrow, Bill Pippin and a number of anonymous referees for comments on earlier versions of this paper.

A An Overview of CLP($\mathcal{R}$)

Real constants and real variables are both *arithmetic terms.* If t, t_1 and t_2 are arithmetic terms, then so are $(t_1 + t_2)$, $(t_1 - t_2)$, $(t_1 * t_2)$, (t_1/t_2), $abs(t)$, $sin(t)$, $cos(t)$ and $pow(t_1, t_2)$. Uninterpreted constants and functors are like those in PROLOG. Uninterpreted constants and arithmetic terms are *terms*, and so is any variable. If f is an n-ary uninterpreted functor, $n \geq 0$, and $t_1, ..., t_n$ are terms, then $f(t_1, ..., t_n)$ is a term. If t_1 and t_2 are arithmetic terms, then $t_1 = t_2$, $t_1 < t_2$ and $t_1 \leq t_2$ are all arithmetic *constraints.* If at least one of t_1 and t_2 is not an arithmetic term, then only the expression $t_1 = t_2$ is a constraint.

An *atom* is of the form $p(t_1, t_2, \ldots, t_n)$ where p is a predicate symbol distinct from $=$, $<$, and $\leq$, and $t_1, ..., t_n$ are terms. A CLP($\mathcal{R}$) program is defined to be a finite collection of *rules* of the form $A_0 :- \alpha_1, \alpha_2, \ldots, \alpha_k$ where each α_i, $0 \leq i \leq k$, is either a constraint or an atom, and A_0 is an atom. A CLP($\mathcal{R}$) *goal* is of the form $?-\ C \mid \alpha_1, \alpha_2, \ldots, \alpha_k$ where C is a conjunction of constraints and each $\alpha_i, 1 \leq i \leq k$, is either a constraint or an atom. A *derivation step* for the above CLP($\mathcal{R}$) goal selects the first atom or constraints α_1. If α_1 is a constraint c, then if $c \wedge C$ is satisfiable the new goal is $?-\ C \wedge c \mid \alpha_2, \ldots, \alpha_k$. If the constraints are unsatisfiable then execution "backtracks" to a point where an alternate choice of matching rule is available and proceeds using this rule. If α_1 is an atom $p(s_1, \ldots, s_n)$ and there is a rule in the program of the form

$$p(t_1, \ldots, t_n) :- \beta_1, \beta_2, \ldots, \beta_l.$$

The resulting new goal is of the form

$$?-\ C \mid s_1 = t_1, \ldots, s_n = t_n, \beta_1, \ldots, \beta_l, \alpha_1, \ldots, \alpha_k.$$

Ideally the constraint solver would detect whenever the constraints are unsatisfiable. In practice, because of the intractability of detecting unsatisfiability of nonlinear constraints, our implementation of CLP($\mathcal{R}$) only detects the satisfiability/unsatisfiability of all the non-arithmetic and linear arithmetic constraints in the goal, including nonlinear constraints that are effectively linear in the context of the non-arithmetic and linear constraints.

A *derivation sequence* is a possibly infinite sequence of goals, starting with an initial goal, wherein there is a derivation step to each goal from the preceding goal. A derivation sequence is *successful* if it is finite and its last goal contains only constraints. The constraints in the last goal of successful derivation sequences are called *answer constraints.*

B Proof Outlines for Main Results

The correctness of all of the optimizations is easy to establish with simple inductive arguments.

Proof of Proposition 1: By induction on derivation length, with cases corresponding to use of the future redundant constraint succeeding or failing. The key is to observe that, by definition of future redundancy, whenever a future redundant constraint is encountered, it will *always* be subsumed at the next derivation step, so the right choice of rule will always be taken. □

Proof of Proposition 2: Much as before. Depends on observing that when a call to a linear recursive procedure succeeds, the base cases have added the appropriate constraints that subsume the omitted ones, and have only done so where constraints were actually omitted. As the constraints added to base cases always represent omitted future redundant ones, failure must correspond to failure in the original procedure. When only a base case is used in the computation, the extra argument ensures that the version that does not add extra constraints is used. □

Proof of Proposition 3: This time the return future redundant constraint is subsumed after a return from the current rule. This happens no matter what the calling rule was. This time the outermost rule (first one used) is the interesting case, and the wrapping rule takes care of the constraint that fails to be subsumed. When only a base case is used, the extra argument ensures that a superfluous constraint is not added. □

C Tree Traversal

This is a binary search tree traversal routine. Traversing nodes in various ways has a cost, that is added to an initial cost on beginning the traversal. The total cost of a search is bounded from above, and increases monotonically. The two recursive rules have different partially future redundant constraints. The interesting queries are of the form

```
?- search(ground-tree, num, num, var, var)
```

so that the bound test is non-ground. The original and unoptimized programs appear in Figures 12 and 13 respectively.

```
search(node(K,D,L,R), K, B, U, U).
search(node(K,D,L,R), C, B, U, V) :-
        C < K,
        U < B,
        search(L, C, B, U+1, V).
search(node(K,D,L,R), C, B, U, V) :-
        C > K,
        U < B,
        search(R, C, B, U+2, V).
```

Figure 12: Original binary search tree traversal program

```
search(node(K,D,L,R), K, B, U, U, a).
search(node(K,D,L,R), K, B, U, U, b(BB,UU)) :-
        add(UU < BB).
search(node(K,D,L,R), K, B, U, U, c(BB,UU)) :-
        add(UU < BB).
search(node(K,D,L,R), C, B, U, V, _) :-
        C < K,
        test(U < B),
        search(L, C, B, U+1, V, b(B, U)).
search(node(K,D,L,R), C, B, U, V, _) :-
        C > K,
        test(U < B),
        search(R, C, B, U+2, V, c(B, U)).
```

Figure 13: Optimized binary search tree traversal program

D Simulation

These programs and queries simulate the behavior of a continuous system numerically. They can be viewed as numerical solution of ordinary differential equations, with a bound on the value of the dependent (state) variable. The state variables are bounded above and increase monotonically.

The first program, in Figures 14 and 15 respectively, has a state consisting of one variable, which increases monotonically and is bounded above. The second, in Figures 16 and 17, has a state consisting of two variables, which also increase monotonically and are bounced above. In this case, the two bounding constraints are both (multiply) partially future redundant, and thus need to be tested together in the optimized program.

```
solve(T, Y, H, TF, YF) :-
        T + H > TF,
        Y = YF.
solve(T, Y, H, TF, YF) :-
        T + H <= TF,
        Y < 100000,
        DYDT = T,
        TN = T + H,
        YN = Y + H * DYDT,
        solve(TN, YN, H, TF, YF).
```

Figure 14: Original univariate simulation

```
solve(T, Y, H, TF, YF, a) :-
        T + H > TF,
        Y = YF.
solve(T, Y, H, TF, YF, b(OY)) :-
        T + H > TF,
        Y = YF,
        add(OY < 100000).
solve(T, Y, H, TF, YF, _) :-
        T + H <= TF,
        test(Y < 100000),
        DYDT = T,
        TN = T + H,
        YN = Y + H * DYDT,
        solve(TN, YN, H, TF, YF, b(Y)).
```

Figure 15: Optimized univariate simulation

Here, the interesting queries are respectively of the forms

```
?- solve(num, var, num, num, var)
```

and

```
?- solve(num, var, var, num, num, var, var)
```

so that a relationship is being built up between the initial and final values of the state variables, so that the bounds test is non-ground. Of course, the most common use would be expected to ground the initial values of the state variables, and then future redundancy would not be an issue. That is, the queries would be of the forms

```
?- solve(num, num, num, num, var)
```

and

```
?- solve(num, num, num, num, num, var, var)
```

respectively.

```
solve(T, Y1, Y2, H, TF, YF1, YF2) :-
        T + H > TF,
        Y1 = YF1,
        Y2 = YF2.
solve(T, Y1, Y2, H, TF, YF1, YF2) :-
        T + H <= TF,
        Y1 < 100000,
        Y2 < 100000,
        TN = T + H,
        YN1 = Y1 + 2 * H * T,
        YN2 = Y2 + H * T + 1,
        solve(TN, YN1, YN2, H, TF, YF1, YF2).
```

Figure 16: Original multivariate simulation

```
solve(T, Y1, Y2, H, TF, YF1, YF2, a) :-
        T + H > TF,
        Y1 = YF1,
        Y2 = YF2.
solve(T, Y1, Y2, H, TF, YF1, YF2, b(OY1, OY2)) :-
        T + H > TF,
        Y1 = YF1,
        Y2 = YF2,
        add(OY1 < 100000),
        add(OY2 < 100000).
solve(T, Y1, Y2, H, TF, YF1, YF2, _) :-
        T + H <= TF,
        test(Y1 < 100000, Y2 < 100000),
        TN = T + H,
        YN1 = Y1 + 2 * H * T,
        YN2 = Y2 + H * T + 1,
        solve(TN, YN1, YN2, H, TF, YF1, YF2, b(Y1, Y2)).
```

Figure 17: Optimized multivariate simulation

E Mortgage Interest

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

Figure 18: Program for reasoning about mortgage repayments

Q_1 : `?- mortgage(P, 360, 1.01, R, B).`
Answer: `P = 0.0278 * B + 97.22 * R, B + 2.01*R >= 0`
Q_2 : `?- 0 <= B, B <= 1030, mortgage(100000, T, 1.01, 1030, B).`
Answer: `T = 355, B = 385.449`

Figure 19: Two queries for the mortgage program

θ_1 : `?- mortgage(any, num, num, any, any).`
θ_2 : `?- mortgage(num, any, num, num, any).`

Figure 20: Two corresponding query patterns

The role of multiple specialization is illustrated well in this example, the well-known Mortgage program, in Figure 18. We consider two query patterns exemplified in Figure 19 and defined formally in Figure 20. Note that we have chosen calling patterns far more general than those suggested by the respective queries: we wish to capture merely that aspect of the queries that tells us which of the two inequality constraints in the recursive rule would benefit substantially from our trying to establish future redundancy.

The first is a relatively straightforward query establishing a linear equation, but with complicated properties with respect to future redundancy. Clearly the constraint `T > 1` is not of interest as `T` will always be ground. Initially it seems that the constraint `P >= 0` is not in any way future redundant, because of the degrees of freedom in the constraints `P1 = P * I - R` in the recursive case and `B = P * I - R` in the base case. In fact, if interest rate `I` and repayment rate `R` could be guaranteed to be non-negative (and in fact no other query would make sense), the inequality is partially future redundant. Furthermore, if the balance `B` can be guaranteed to be non-negative (also expected) the equality is actually future redundant. These properties are easy to determine automatically, and since `I`, `R` and `B` are passed through the recursion unchanged, it seems reasonable to deal with the situation using multiple specialization. However, we may also notice that full future redundancy buys us little in the way of performance improvement, so it is better to forego the extra code, and just concentrate on partial future redundancy. The optimized program is shown in Figure 21. Notice that the constraints used for multiple specialization could in principle result in thrashing, although in this case the possibility is of little interest because of the semantics of the variables. We should note that the cost of thrashing would actually be lower than that of failing to apply the optimization in general.[1]

The second query seems more complicated, yet this time only the constraint `T > 1` is of interest, and it is clearly future redundant. The resulting program is shown in Figure 22.

```
mortgage(P, T, I, R, B) :-
    R > 0, I > 0,                   %% needed for P >= 0 to be pfr
    mg_rip(P, T, I, R, B).          %% optimized version
mortgage(P, T, I, R, B) :-
    ( R <= 0; I <= 0 )              %% out of luck
    mg_general(P, T, I, R, B).      %% call the general version above

mg_rip(P, T, I, R, B) :-            %% wrapper for optimized version
    mg_rip(P, T, I, R, B, a).       %% old base case first

mg_rip(P, T, I, R, B, _) :-
    T > 1,                          %% ground test
    T1 = T - 1,                     %% ground evaluation & assignment
    test(P >= 0),                   %% partially future redundant
    P1 = P * I - R,                 %% I > 0, R > 0 this version
    mg_rip(P1, T1, I, R, B, f(P)).  %% pass old value of P
mg_rip(P, T, I, R, B, f(OP)) :-     %% general case: get old P
    T = 1,                          %% ground test
    add(OP >= 0),                   %% last pfr constraint added
    B = P * I - R.                  %% I > 0 known, R >= 0 this version
mg_rip(P, T, I, R, B, a) :-         %% kept for no iteration case
    T = 1,                          %%    no constraint added
    B = P * I - R.
```

Figure 21: Mortgage program optimized for θ_1

```
mortgage(P, T, I, R, B) :-
    test(T > 1),                    %% future redundant
    T1 = T - 1,
    P >= 0,                         %% ground test
    P1 = P * I - R,                 %% ground evaluation & assignment
    mortgage(P1, T1, I, R, B).
mortgage(P, T, I, R, B) :-
    T = 1,                          %% subsumes last fr inequality
    B = P * I - R.                  %% ground evaluation & assignment
```

Figure 22: Mortgage program optimized for θ_2

[1] This example is actually incorrect in Jørgensen et al. [3]: on page 422, Q_3 needs the additional constraint $B \geq 0$, which of course will reappear in the answer constraint shown therein.

References

[1] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. An abstract machine for CLP($\mathcal{R}$). In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI), San Francisco*, pages 128–139, June 1992.

[2] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 14(3):339–395, July 1992.

[3] Niels Jørgensen, Kim Marriott, and Spiro Michaylov. Some global compile-time optimizations for CLP($\mathcal{R}$). In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming: Proceedings of the 1991 International Symposium*, pages 420–434, San Diego, CA, October 1991. MIT Press.

[4] Kim Marriott and Harald Søndergaard. Analysis of constraint logic programs. In Saumya Debray and Manuel Hermenegildo, editors, *Proc. of the 1990 North American Conference on Logic Programming*, pages 521–540, Austin, TX, 1990. MIT Press.

[5] Kimbal G. Marriott and Peter J. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. In *Proc. ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Charleston, NC, January 1993.

[6] Andrew D. McDonald, Peter J. Stuckey, and Roland H. C. Yap. Redundancy of variables in CLP($\mathcal{R}$). In *Logic Programming: Proceedings of the 1993 International Symposium*, Vancouver, October 1993. MIT Press.

[7] Spiro Michaylov. *Design and Implementation of Practical Constraint Logic Programming Systems.* PhD thesis, School of Computer Science, Carnegie Mellon University, August 1992. Available as technical report CMU-CS-92-168.

[8] Spiro Michaylov. Optimizing CLP($\mathcal{R}$) programs by reordering subgoals. Technical report, Department of Computer and Information Science, The Ohio State University, May 1994.

[9] William H. Winsborough. Path-dependent reachability analysis for multiple specialization. In *Logic Programming: Proceedings of the North American Conference*, pages 133–153, Cleveland, OH, USA, November 1989. MIT Press.