Optimizing Compilation of Linear Arithmetic in a Class of Constraint Logic Programs

Technical Report OSU-CISRC-8/94-TR44

Spiro Michaylov Bill Pippin

Department of Computer and Information Science,
The Ohio State University,
395 Dreese Lab,
2015 Neil Avenue Mall,
Columbus, OH 43210-1277, U.S.A.,
Voice: +1 (614) 292 6377
FAX: +1 (614) 292 2911
{spiro,pippin}@cis.ohio-state.edu.

August 31, 1994

Abstract

A central issue in the optimizing compilation of Constraint Logic Programming (CLP) languages is how to compile away as much general constraint solving as possible. Most such work relies on obtaining mode and type information by global analysis, and uses it to generate specialized code for individual constraints and calls, often with the aid of multiple specialization. Some recent work has augmented these techniques with procedure-level analysis of the inter-relationships between constraints, to detect constraints that subsume other constraints, and variables that cease to be reachable at some point in a computation. In combination, these techniques have been shown to dramatically improve performance for a number of programs.

Here we continue this line of investigation by considering a class of programs that accumulate and simplify systems of linear arithmetic constraints. The programs contain procedures that relate their parameters by an affine transform. For some calling patterns, the procedures repeatedly compose and simplify affine transforms, using virtually the full power of the linear arithmetic constraint solver, and incurring considerable expense. We describe source to source translations that make this composition of transforms explicit, replacing constraint solving with ground (imperative) arithmetic. We demonstrate the translations and present experimental data showing substantial improvements in execution speed and space utilization.

1 Introduction

Many Constraint Logic Programming (CLP) [7] languages incorporate solving of linear arithmetic equations as a basic operational step. These include CLP($\mathcal{R}$) [8], Prolog III [1], CHIP [4] and ECL^{i}PS^e [6]. Programs that frequently require simultaneous linear equations to be checked for satisfiability can cause the solver to use a large amount of space, and may execute slowly. Thus, much effort has recently been put into global program analysis, in the hope that detailed mode and type information can be used to improve performance by reducing the use of the constraint solver.

Global analysis techniques for Constraint Logic Programming have been discussed by Dumortier *et al* [5], García de la Banda and Hermenegildo [2], Jørgensen [9], Marriott and Søndergaard [12], and others. Based on such analysis, considerable work has been done on actual optimizing compilation of CLP languages, by García de la Banda *et al* [3], Jørgensen *et al* [10], Michaylov [15], MacDonald *et al* [11], Marriott and Stuckey [13], and others. Most attempts at optimizing CLP programs depend on a small number of techniques applied at the level of individual procedures in addition to basic global analysis:

- *Multiple specialization*
  Various calling patterns enable different optimizations. This is most fully described by Winsborough in [20].

- *Subgoal reordering*
  Different orders of constraints in the body of a rule may be appropriate for different calling patterns. This was first described for CLP by Marriott and Stuckey in [13], and studied systematically by Michaylov [14].

- *Fold/Unfold transformations*
  These can enable optimizations based on procedures of certain restricted forms.

- *Constraint interactions across rule instances*
  Some of the most useful optimizations rely on observing how various constraint instances throughout the execution of a procedure interact. Some examples include future redundancy [10, 15], dead variable elimination [11] and constraint removal [13].

Many CLP programs dealing with arithmetic, for some classes of queries, establish a network of linear constraints one at a time, calling the equation solver to simplify the network and check its satisfiability at each operational step. However, a large and important subclass of programs establishes this network in such a way that much of this simplification process could be compiled out of the solver, passing it only the simplified set of constraints at the end. Let us consider, as a simple example, the program of Figure 1, which uses Euler's method to solve the ordinary differential equation $y' = t$.

The use of the constraint solver by this program varies depending on the calling pattern. We use `num` to represent a ground arithmetic value and `free` to denote that an argument is completely unconstrained. Consider the following examples:

- θ_1 : `?- solve(num, num, num, num, free).`
  Requires only simple arithmetic evaluation and testing.

- θ_2 : `?- solve(num, free, num, num, num).`
  After suitable re-ordering of constraints, as described in [14], this also requires only simple arithmetic evaluation and testing.

- θ_3 : `?- solve(num, free, num, num, free).`
  Establishes a chain of linear equations linking the two free arguments. In particular

$$\mathtt{YN}_1 = \mathtt{Y}_1 + \mathtt{H} * \mathtt{T}, \quad \ldots, \quad \mathtt{YN}_n = \mathtt{Y}_n + \mathtt{H} * (\mathtt{T} + n * \mathtt{H})$$

  where n is the number of iterations, and the additive term is always ground.

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

Figure 1: Program solving a simple ODE numerically

```
solve(Ti,Yi,H,Tn,Yn) :-
        H1 = H + 1,
        solve1(Ti,1,H,Tn,An,H1),
        Yn = Yi * An.

solve1(Ti,An,H,Tn,An,H1) :- Ti + H >= Tn.
solve1(Ti,Ai,H,Tn,An,H1) :- Ti + H <= Tn,
        Tj = Ti + H,
        Aj = Ai * H1,
        solve1(Tj,Aj,H,Tn,An,H1).
```

Figure 2: Optimized numerical ODE solver

In this paper, we are concerned with optimizing a general class of programs for which the last case is a simple example. Let us consider this case more closely. On termination, the query results in a linear equation between the two variables, Y_0 and Y_f, of the form $Y_f = Y_0 + c$. None of the intermediate variables of the computation are of interest: the only observable of the program is the resulting constraint. At no intermediate stage should the computation fail and backtrack.

An efficiency improvement is possible in this case, as the constraint solver is not aware of the global structure of these constraints, and each time it is called (n times) it must essentially interpret each individual constraint, to determine its relation to others. In this program, the work required at each step could be collected outside the constraint solver, and only the final constraint need be passed to it. Figure 2 shows a transformed version of this program for the calling pattern θ_3, which in fact will also work correctly for any further instantiation of the second and fifth arguments.

Consider the execution of this program for the query pattern θ_3. The auxiliary procedure `solve1` threads an accumulator through its second and fifth arguments, and then the summarizing equation is passed to the solver. This equation is linear, since the auxiliary procedure runs ground and hence the coefficient variable `An` is ground. However, viewed from outside `solve`, the execution is exactly equivalent to that of the original program. The performance improvement can be expected to be dramatic, as `solve1` can be compiled to completely imperative code, and the constraint solver obtains one linear equation per query.

In this paper, we define a series of increasingly general classes of programs for which optimizations closely related to this one are possible. We define the optimizing translations, and consider their effect analytically and empirically. We demonstrate that the optimizations can be useful in practice by establishing the methodological usefulness of the program templates involved, and by studying some real examples. In Section 2 we establish notation and terminology needed to make our presentation easier. Then Section 3 introduces three core classes of programs together with their associated optimizations, with examples. The effect of inequality constraints on the core program classes is considered in Section 4. Then in Section 5 we give empirical evidence of the usefulness of the optimizations, consider the practical aspects of using these translations in an optimizing compiler in Section 6, and conclude in Section 7. We collect some technical details related to affine transformations, and proofs of correctness for those propositions of Sections 3 and 4, in Appendix A; and discuss some example applications in detail, in Appendix B.

2 CLP Procedures as Affine Transforms

Throughout this paper we will need to systematically refer to classes of programs and queries that establish networks of linear equations and inequalities in certain very specific ways. Furthermore, we will need to formally define translations on these classes of programs, and reason about the relationship between the original and translated program. To make the presentation easier to follow we need to establish a certain way to view these programs as well as terminology and notation for describing them. While the material is broadly applicable to all CLP languages with linear arithmetic, the presentation is influenced by the use of CLP($\mathcal{R}$) for the practical aspects of this research.

2.1 Programs, Procedures and Calling Patterns

Our primary concern is with optimizing sets of rules that define a single predicate, with respect to a given allowed query pattern (or calling pattern). We will call such a set of rules (a predicate definition) together with a calling pattern a procedure. In this view, a predicate definition represents many different procedures, and thus we expect that the predicate definition is optimized by multiple specialization. That is, we expect it to be optimized by separately optimizing each of the many procedures it defines, on the assumption that different optimizations will be applicable to each procedure. In particular, the optimizations discussed in this paper will only be applicable to a given predicate definition for certain calling patterns, if at all.

Definition 2.1 (Procedure) *A procedure $\langle p/n, \theta\rangle$ is a set of rules defining an n-ary predicate denoted by the symbol p, with calling pattern $\theta = \langle \alpha_1, \ldots, \alpha_n\rangle$ where each α_i is an element of Insts $\times$ Types, where Insts* $= \{\texttt{ground}, \texttt{free}, \texttt{unknown}\}$ *and Types* $= \{\texttt{arith}, \texttt{herbrand}, \texttt{unknown}\}$.

To further simplify the presentation, we will tend to denote a calling pattern as a sample query, with the arguments being single tokens such as `num` for $\langle$`ground`, `arith`$\rangle$, `arith` for $\langle$`unknown`, `arith`$\rangle$ and `free` for $\langle$`free`, `arith`$\rangle$, as we will mostly be concerned with specifying the instantiation status of arithmetic variables.

We should also note that we are mostly concerned with deterministic procedures that employ direct recursion, although similar optimizations may be applied to a generalized class of procedures with mutual recursion.

2.2 Affine Transforms

A common feature of all the procedures we will consider, irrespective of their recursion pattern, is that they establish a special kind of relationship between two subsets of their argument variables: an affine transform. Consider the following simple procedure:

```
p(X, Y, U, V) :-
        U - 2 * X = 7,
        V = 3 * X + 4 * Y.
```

After a small amount of re-arrangement we can see that the relationship established between the four argument variables is

$$\begin{bmatrix} \mathtt{U} \\ \mathtt{V} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 3 & 4 \end{bmatrix} \begin{bmatrix} \mathtt{X} \\ \mathtt{Y} \end{bmatrix} + \begin{bmatrix} -7 \\ 0 \end{bmatrix}$$

and so the entire rule may be represented as

$$p(\tilde{Z}, \tilde{W}) \text{ :- } \tilde{W} = \mathrm{A}\tilde{Z} + \tilde{\mathrm{a}}.$$

where

$$\mathrm{A} = \begin{bmatrix} 2 & 0 \\ 3 & 4 \end{bmatrix} \qquad \tilde{\mathrm{a}} = \begin{bmatrix} -7 \\ 0 \end{bmatrix}$$

and $\tilde{Z}$ denotes the two variables `X` and `Y`, and $\tilde{W}$ likewise denotes `U` and `V`.

The relationship between $\tilde{Z}$ and $\tilde{W}$ is called an affine transform. We will denote an affine transform T as a pair $\langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$ and write

$$p(\tilde{Z}, \tilde{W}) \text{ :- } \tilde{W} = \mathrm{T}(\tilde{Z}).$$

This notation will make much of our presentation vastly simpler. Now consider a program of the form

$$\begin{array}{rcl} p(0, \tilde{X}, \tilde{Y}) & \text{:-} & \tilde{Y} = \mathrm{T}_1(\tilde{X}). \\ p(N, \tilde{X}, \tilde{Y}) & \text{:-} & N > 0, \tilde{Z} = \mathrm{T}_2(\tilde{X}), p(N-1, \tilde{Z}, \tilde{Y}). \end{array}$$

for a query of the form ?- $p(n, \tilde{U}, \tilde{V})$. We see that with this query the program computes the affine transform

$$\tilde{V} = \mathrm{T}_1(\underbrace{\mathrm{T}_2(\mathrm{T}_2(\ldots(\mathrm{T}_2(\mathrm{T}_2}_{n}(\tilde{U})))\ldots)))$$

Composition, inverse and various other algebraic operations on affine transforms, together with their properties, are collected in Appendix A. If we write $\circ$ for composition of transforms we can see that this is equivalent to computing

$$\tilde{V} = (\mathrm{T}_1 \circ \underbrace{\mathrm{T}_2 \circ \cdots \circ \mathrm{T}_2}_{n})(\tilde{U})$$

by first computing the composition of all the transforms, which itself is affine, and then applying it. This is essentially the optimization demonstrated in Section 1. The procedure `p/3` above repeatedly applies an affine transform to a tuple of non-ground arithmetic variables, which are related to the free variables $\tilde{U}$ by the composition of those transforms applied so far. Such an application is an expensive operation, involving the full power of the linear equation solver. On the other hand, explicitly composing affine transforms is a ground arithmetic operation, which can easily be compiled down to imperative code. While the theoretical complexity of the two computations above is the same, the second one can be implemented vastly more efficiently within a CLP system.

Finally, let us consider what the resulting program should look like:

$$p(N, \tilde{X}, \tilde{Y}) \text{ :- } p'(N, T_I, \mathcal{T}), \tilde{Y} = \mathcal{T}(\tilde{X}).$$

$$\begin{array}{rcl} p'(0, \mathcal{U}, \mathcal{V}) & \text{:-} & \mathcal{V} = \mathrm{T}_1 \circ \mathcal{U}. \\ p'(N, \mathcal{U}, \mathcal{V}) & \text{:-} & N > 0, \mathcal{W} = \mathrm{T}_2 \circ \mathcal{U}, p'(N-1, \mathcal{W}, \mathcal{V}). \end{array}$$

Here we use the calligraphic font as a shorthand for the variables that represent an affine transform, but they can be thought of as "transform-valued variables". This is analogous to our use of the tilde to denote the variables being transformed, while thinking of them as a "vector-valued variable". In subsequent sections we will see a number of procedures and transformations on them described in this way, differing essentially in the recursion pattern involved.

3 Systems of Equalities

Our objective is to describe formally the classes of programs that can be optimized, and then describe the optimizations as source to source translations; the translations then allow us to reduce solver overhead by replacing constraint tests with goals that can be solved by direct evaluation and assignment. The transformed programs can then be further optimized using the "solver bypass" techniques described in [10]. Furthermore, these optimizations are intended for use with other optimizations, such as the reordering of subgoals within rules [14], and future redundancy [10, 15]. In fact, applicability of the optimizations described here is a good criterion for those reordering translations, as many programs will need reordering before these are applicable.

In this section, we describe progressively more general versions formally. We begin with linear recursion, where there is at most one recursive call in any rule of a procedure, and consider alternate ways to translate linear recursive procedures. We then define two classes of multiply recursive procedures, and show how the various translations for linear recursive procedures can be generalized to such procedures. Proofs of correctness for those translations are given in Appendix A.

In the following, we use the notation $\tilde{X}$ for a vector of program variables, T for an affine transform in a source program, and $\mathcal{X}$ for a transform added to an optimized program. In all cases these are shorthands for a finite number of program terms, not an addition to the language.

For a procedure P, of predicate p, having some rule $p(\tilde{X}, \tilde{Y}, \tilde{Z})$:- $G_1, \cdots, G_n$, the following conventions are used in defining procedure classes.

- In the head,
  - $\tilde{X}$ and $\tilde{Y}$ are vectors of numeric terms that are of interest, and
  - $\tilde{Z}$ is the set of all other arguments.
- In the body, a goal G_i may be:
  - a recursive call to P; or
  - a constraint involving a ground affine transform T, and any two of $\tilde{X}$, $\tilde{Y}$, $\tilde{X}'$, or $\tilde{Y}'$, where $\tilde{X}'$ and $\tilde{Y}'$ are also parameters in a recursive call to P; or
  - any other goal that does not cause a recursive call to P, or involve terms from $\tilde{X}$, $\tilde{Y}$, or any $\tilde{X}'$ or $\tilde{Y}'$.

For any G_i, goals of the first two types are explicitly mentioned in definitions, so that in a body $G_1, \cdots, G_n$ all other goals are of the third type. Definitions describe rules for recursive and base cases, and in each definition, it should be understood that at least one rule of each type must occur, and that no other form of rule is allowed.

3.1 Linear Recursion

The simplest procedure class of interest has exactly one recursive goal in the body of a recursive rule.

Definition 3.1 (Linear Recursive Transforming Procedure) *Let P be a set of rules defining predicate p and calling pattern θ, of form*

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

- *base cases*
  G_k *is* $\tilde{Y} = \mathrm{T}(\tilde{X})$, *for* $1 \le k \le n$
- *recursive cases*
  For some $k, l, m : 1 \le k < l < m \le n$
  1. G_k *is* $\tilde{U} = \mathrm{T}_1(\tilde{X})$
  2. G_l *is* $p(\tilde{U}, \tilde{V}, \tilde{Z})$
  3. G_m *is* $\tilde{Y} = \mathrm{T}_2(\tilde{V})$

For terms of interest, and depending on the calling pattern θ, a linear recursive procedure will either perform ground computation, or accumulate equality constraints in the solver. For θ with arguments corresponding to $\tilde{X}$ and $\tilde{Y}$ free, linear recursive transforming procedures build a chain of constraints between $\tilde{X}$ and $\tilde{Y}$ using the affine transforms. We can use the associativity of these transforms to define equivalent procedures with simplified constraints, so that the transforms are composed and then applied to $\tilde{X}$. The simplicity of the recursion pattern enables us to compose these transforms in two different ways.

The first, Translation 3.1, performs ground computation by threading transforms from the initial call to the base case and back again, and the next, Translation 3.2, gathers transform terms starting at the base case and finishing at the initial call. In both cases we replace the application of affine transforms to possibly free terms with composition of those transforms, which requires only ground computation. We refer to these as *thread* and *gather* translations, respectively.

Translation 3.1 (Linear Threaded) *Let $P = \langle p, \theta \rangle$ be a linear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{X}, \mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*
  G_k *is* $\mathcal{Y} = \mathrm{T} \circ \mathcal{X}$
- *recursive cases*
  1. G_k *is* $\mathcal{U} = \mathrm{T}_1 \circ \mathcal{X}$
  2. G_l *is* $p'(\mathcal{U}, \mathcal{V}, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = \mathrm{T}_2 \circ \mathcal{V}$

Additionally, a wrapping rule of the form

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } p'(\mathrm{T_I}, \mathcal{Y}, \tilde{Z}), \quad \tilde{Y} = \mathcal{Y}(\tilde{X}).$$

is added.

Note that either the pre or post-recursion transform may be the identity, $\mathrm{T_I} = \langle \mathrm{I}, \tilde{\mathbf{0}} \rangle$. Note also that without loss of generality we have assumed variables only in heads and calls.

Translation 3.2 (Linear Gather) *Let $P = \langle p, \theta \rangle$ be a linear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*
  G_k *is* $\mathcal{Y} = \mathrm{T}$
- *recursive cases*
  1. G_k *is deleted*
  2. G_l *is* $p'(\mathcal{X}, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = \mathrm{T}_2 \circ \mathcal{X} \circ \mathrm{T}_1$

and the wrapping rule is:

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } p'(\mathcal{Y}, \tilde{Z}), \quad \tilde{Y} = \mathcal{Y}(\tilde{X}).$$

Translation 3.3 (Inverted Linear Gather) *Let $P = \langle p, \theta \rangle$ be a linear transforming procedure. If the transform inverses $\mathcal{Y}$ exist for both base and recursive cases, an equivalent procedure is obtained by replacing each rule by:*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*
  G_k *is* $\mathcal{Y} = \mathrm{T}^{-1}$
- *recursive cases*
  1. G_k *is deleted*
  2. G_l *is* $p'(\mathcal{X}, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = (\mathrm{T}_2 \circ \mathcal{X}^{-1} \circ \mathrm{T}_1)^{-1}$

and the wrapping rule is:

$$p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } p'(\mathcal{Y}, \tilde{Z}), \quad \tilde{Y} = \mathcal{Y}^{-1}(\tilde{X}).$$

Note that this and some future translations rely on some affine transforms being invertible. In practice, this will restrict the applicability of the translations, or just result in triggering multiple specialization.

Proposition 3.1 (Linear Recursive Translations) *Let* $P = \langle p, \theta \rangle$ *be a linear transforming procedure. The result of applying Translations 3.1, 3.2 or 3.3 to P preserves success, failure, length of computation, and correctness.*

```
fib(N,F) :- N > 0, f(N,_,F).            % original wrapper
fib(0,1).

f(N,Z,Y) :- N > 0,                      % original recursive procedure
        Z = Y + X,
        f(N-1,Y,X).
f(0,1,1).

fib(N,F) :- N > 0, f(N,1,1,1,F).        % wrapper for optimized procedure
fib(0,1).

f(N,I,A,B,F) :- I < N,                  % optimized recursive procedure
        f(N,I+1,B,A+B,F).
f(N,N,_,F,F).
```

Figure 3: Linear Recursive Fibonacci Number Relations

Example 3.1 (Linear Gather Optimization) *Two versions of* `fib(N,F)` *are given in Figure 3. The first improves over the naive doubly recursive version of* `fib(N,F)` *by passing an accumulator pair to an auxiliary procedure* `f/3` *that is linear recursive. It can be improved still further using Translation 3.2.*

For the query pattern `fib(25,N)`, `f/3` *accumulates a chain of constraints,* $F_n = F_{n-1} + F_{n-2}, F_{n-1} = F_{n-2} + F_{n-3}, \ldots, F_1 = 1, F_0 = 1$.

Let

$$T_0 = \left\langle \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\rangle \quad \textit{and} \quad T_f = \left\langle \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\rangle.$$

For the base and recursive cases, `f/3` *computes*

$$\begin{aligned} [F_0, F_1]^T &= T_0([F_0, F_1]^T) \\ [F_1, F_2]^T &= T_f([F_0, F_1]^T) \end{aligned}$$

Applying Translation 3.2 to get `f/5`*, the base case simply returns* T_0*, while the recursive cases compose* n *transforms.*

$$\underbrace{T_f \circ \cdots \circ T_f}_{n-1} \circ T_0$$

Note that since T_f *consists of a (matrix,zero vector) pair, composition is equivalent to repeated matrix multiplication, and a compiler may further optimize* `f/5` *to get the closed form solution for the Fibonacci function.*

3.2 Multilinear Recursion

It turns out that we can extend the definition of a linear transforming procedure to situations where some rules make more than one recursive call. A multilinear recursive procedure is similar to the linear recursive procedure of Definition 3.1 except that there are multiple recursive calls to P in a single rule, and parameters to those calls are threaded from one recursive call to the next.

Definition 3.2 (Multilinear Recursive Transforming Procedure) *A multilinear recursive procedure* P *consists of base and recursive case rules of the following form:*

- *base cases*
  G_k *is* $\tilde{Y} = \mathrm{T}(\tilde{X})$, *for* $1 \le k \le n$
- *recursive cases*
  For some $c, m : 0 < k_1 < l_1 < \cdots < k_c < l_c < m < n$
  1. G_{k_1} *is* $\tilde{X}_1 = \mathrm{T}_0(\tilde{X})$, *and* G_{k_i} *is* $\tilde{X}_i = \mathrm{T}_{i-1}(\tilde{Y}_{i-1})$, *for* $2 \le i \le c$
  2. G_{l_i} *is* $p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z})$
  3. G_m *is* $\tilde{Y} = \mathrm{T}_c(\tilde{Y}_c)$

That is, for c recursive calls within a rule, we potentially have $c+1$ affine transforms interleaved between the calls. The multilinear translations for thread and gather are similar to their respective linear translations (3.1 and 3.2). In each case the recursive cases change to reflect the multiple recursive calls, while the base cases and wrapping rule are unchanged.

Translation 3.4 (Multilinear Threaded) *Let* $P = \langle p, \theta \rangle$ *be a multilinear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{X}, \mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*
  G_k *is* $\mathcal{Y} = \mathrm{T}_0 \circ \mathcal{X}$
- *recursive cases*
  1. G_{k_1} *is* $\mathcal{X}_1 = \mathrm{T}_0 \circ \mathcal{X}$, *and* G_{k_i} *is* $\mathcal{X}_i = \mathrm{T}_{i-1} \circ \mathcal{Y}_{i-1}$, *for* $2 \le i \le c$
  2. G_{l_i} *is* $p'(\mathcal{X}_i, \mathcal{Y}_i, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = \mathrm{T}_c \circ \mathcal{X}_c$

and the wrapping rule is unchanged from Translation 3.1.

Translation 3.5 (Multilinear Gather) *Let $P = \langle p, \theta \rangle$ be a multilinear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*
  G_k *is* $\mathcal{Y} = \mathrm{T}$
- *recursive cases*
  1. G_{k_i} *is deleted*
  2. G_{l_i} *is* $p'(\mathcal{Y}_i, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = \mathrm{T}_c \circ \mathcal{Y}_c \circ \cdots \circ \mathrm{T}_1 \circ \mathcal{Y}_1 \circ \mathrm{T}_0$

and the wrapping rule is unchanged from Translation 3.2.

Proposition 3.2 (Multilinear Recursive Translations) *Let $P = \langle p, \theta \rangle$ be a multilinear transforming procedure. The result of applying Translations 3.4 or 3.5 to P preserves success, failure, length of computation, and correctness.*

3.3 Non-linear Recursion

In the multilinear procedure of the previous section the computation was defined as a single linear sequence of transform applications. We can further extend the definition of multiply recursive procedures to include the case where a procedure P adds transform applications, so that computation forms a tree. A nonlinear recursive procedure differs from the linear recursive procedure of Definition 3.1 in that there are multiple recursive calls to P in a single rule, and parameters of those calls are copied from $\tilde{X}$, and summed into $\tilde{Y}$.

Definition 3.3 (Simple Nonlinear Recursive Transforming Procedure) *A nonlinear recursive procedure P consists of base and recursive case rules of the following form:*

- *base cases*
  G_k *is* $\tilde{Y} = \mathrm{T}(\tilde{X})$, *for* $1 \leq k \leq n$
- *recursive cases*
  For some $c, m : 1 \leq k_1 < l_1 < \cdots < k_c < l_c < m \leq n$
  1. G_{k_i} *is* $\tilde{X}_i = \mathrm{T}_{1_i}(\tilde{X})$
  2. G_{l_i} *is* $p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z})$
  3. G_m *is* $\tilde{Y} = \sum_{i=1}^{c} \mathrm{T}_{2_i}(\tilde{Y}_i)$

With nonlinear procedures threading is no longer an alternative, and a gather technique is used instead. The nonlinear gather definition differs from the multilinear gather case above in that transform parameters are copied before the recursive call, and combined (gathered) afterwards, rather than passing through a chain of applications. Otherwise the translation is similar to previous gather forms, with the same base case and wrapping rule.

Translation 3.6 (Nonlinear Gather) *Let P be a nonlinear transforming procedure. An equivalent procedure is obtained by replacing each rule by*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

Where:

- *base cases*
  G_k *is* $\mathcal{Y} = \mathrm{T}$
- *recursive cases*
  1. G_{k_i} *is deleted*
  2. G_{l_i} *is* $p'(\mathcal{Y}_i, \tilde{Z})$
  3. G_m *is* $\mathcal{Y} = \sum_{i=1}^{c}(\mathrm{T}_{2_i} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_i})$

and the wrapping rule is unchanged from Translation 3.2.

Previous definitions have allowed multiple base and recursive cases, but have required that transforms be applied to some distinguished vector of terms, say $\tilde{X}$. We can extend our definitions to the case where transforms are applied to both $\tilde{X}$ and $\tilde{Y}$, so that goals $\tilde{Y} = \mathrm{T}(\tilde{X})$ and $\tilde{X} = \mathrm{T}(\tilde{Y})$ are found in different rules of the same procedure.

Definition 3.4 (Asymmetrical Nonlinear Recursive Transforming Procedure) *An asymmetrical nonlinear recursive procedure differs from Definition 3.3 in that the base and recursive rules may have two forms.*

- *base cases*
  For some $k : 1 \le k \le n$

  G_k *is* $\tilde{Y} = \mathrm{T}(\tilde{X})$ *or* G_k *is* $\tilde{X} = \mathrm{T}(\tilde{Y})$

- *recursive cases*
  For some $k, l, m : 1 \le k_1 < l_1 < \cdots < k_c < l_c < m \le n$

1. G_{k_i} *is* $\tilde{X}_i = \mathrm{T}_{1_i}(\tilde{X})$		*1.* G_{k_i} *is* $\tilde{Y}_i = \mathrm{T}_{1_i}(\tilde{Y})$
2. G_{l_i} *is* $p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z})$	*or*	*2.* G_{l_i} *is* $p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z})$
3. G_m *is* $\tilde{Y} = \sum_{i=1}^{c} \mathrm{T}_{2_i}(\tilde{Y}_i)$		*3.* G_m *is* $\tilde{X} = \sum_{i=1}^{c} \mathrm{T}_{2_i}(\tilde{X}_i)$

For purposes of translation, we choose one of the columns above as the preferred form, and designate rules of that form to be *symmetrical*, and rules of the other to be *asymmetrical.*

Translation 3.7 (Asymmetrical Nonlinear Gather) *Let* $P = \langle p, \theta \rangle$ *be an asymmetrical nonlinear transforming procedure, so that there are both symmetrical* ($\Rightarrow$) *and asymmetrical* ($\rightleftharpoons$) *rules; and, w.l.o.g., let the left column of Definition 3.4 be chosen as the symmetrical form. If, in the asymetrical rules below, the transform inverses* $\mathcal{Y}$ *exist, an equivalent procedure is obtained by replacing each rule by:*

$$p'(\mathcal{Y}, \tilde{Z}) \text{ :- } G_1, \cdots, G_n.$$

with goals replaced as follows:

goal	*base* G_k	*recursive* G_{k_i}	G_{l_i}	G_m
$\Rightarrow$	$\mathcal{Y} = \mathrm{T}$	*true*	$p'(\mathcal{Y}_i, \tilde{Z})$	$\mathcal{Y} = \sum_{i=1}^{c}(\mathrm{T}_{2_i} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_i})$
$\rightleftharpoons$	$\mathcal{Y} = \mathrm{T}^{-1}$	*true*	$p'(\mathcal{Y}_i, \tilde{Z})$	$\mathcal{Y} = [\sum_{i=1}^{c}(\mathrm{T}_{2_i} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i})]^{-1}$

The wrapping rule is unchanged from Translation 3.2.

```
analyze(C,V,I) :- C = ser(C1,C2), analyze(C1,V1,I), % original program
                                  analyze(C2,V2,I), V = V1 + V2.
analyze(C,V,I) :- C = par(C1,C2), analyze(C1,V,I1),
                                  analyze(C2,V,I2), I = I1 + I2.
analyze(C,V,I) :- C = r2K, V = I * 2000.

wrapper(C,V,I) :-                                   % optimized program
        analyze(C,R),
        V = I * R.

analyze(C,R) :- C = ser(C1,C2), analyze(C1,R1),
                                analyze(C2,R2), R = R1 + R2.
analyze(C,R) :- C = par(C1,C2), analyze(C1,R1),
                                analyze(C2,R2), S = R1 + R2,
                                                P = R1 * R2, R = P / S.
analyze(C,R) :- C = r2K, R = 2000.
```

Figure 4: Circuit Procedure

Translation 3.7 replaces gather transforms in asymmetrical rules with inverses to ensure that all rules have the same direction.

Proposition 3.3 (Nonlinear Recursive Translations) *Let $P = \langle p, \theta \rangle$ be a nonlinear transforming procedure, and let any transform inverses required by Translation 3.7 exist. The result of applying Translation 3.7, or 3.6 where applicable, to P preserves success, failure, length of computation, and correctness.*

Example 3.2 (Asymmetrical Nonlinear Gather) *Examining Figure 4, and applying Translation 3.7, we see that the transform terms for each of the three cases are as follows:*

$$\begin{aligned} T_{ser} &= \langle R_1 + R_2, \tilde{\mathbf{0}} \rangle \\ T_{par} &= \left\langle \frac{1}{1/R_1 + 1/R_2}, \tilde{\mathbf{0}} \right\rangle \\ &= \left\langle \frac{R_1 R_2}{R_1 + R_2}, \tilde{\mathbf{0}} \right\rangle \\ T_{res} &= \langle R, \tilde{\mathbf{0}} \rangle \end{aligned}$$

4 Systems of Inequalities

So far we have dealt only with chains of linear equations, where the intermediate variables are not involved with any ancillary constraints. However, under rather restricted circumstances, it can also be useful to consider inequalities. First we consider ancillary inequality constraints. We do not go through the tedious details of presenting all of the modified procedure classes and translations, but rather indicate how they can be extended. Then, we consider chains of inequalities analogous to the present chains of equations. Proofs for the propositions in this section are given in Appendix A.

4.1 Ancillary Inequality Constraints

Consider the program in Figure 5. For allowed query pattern `?- p(num, free, free)`, this establishes a chain of equations, with an inequality constraint on each node in the chain, and thus does not conform to

```
p(0, X, X).
p(T, X, Z)
        T > 0, Y = 0.5 * X - 2, Y > 0,
        p(T - 1, Y, Z).
```

Figure 5: Linear Transformation with Bound Test

```
setineq(a, _).
setineq(b, Y) :- Y > 0.

p(0, M, V, M, V, a).
p(T, M, V, M0, V0, b)
        T > 0, M1 = 0.5 * M, V1 = 0.5 * V - 2,
        p(T - 1, M1, V1, M0, V0, _).

p(T, X, Y) :-
        p(T, 1, 0, M, V, Initial),
        Y = M * X + V,
        setineq(Initial, Y).
```

Figure 6: Optimized Linear Transformation with Bound Test

our definition of linear transforming procedure. However, the program can still be optimized. The inequality `Y > 0` is partially future redundant in the sense of [15], and hence need only be tested at each iteration, with only the last one being added to the constraint solver. Furthermore, for the given allowed query pattern, that inequality cannot cause failure at any stage of the recursion, and hence need not even be checked. This is because, for any value of `Y` there is a corresponding value for `X`, as the equation can be rewritten `X = 2 * Y + 4`, and, by partial future redundancy, if `Y > 0` as required, then `X > 0`, and `X` is the `Y` of the previous iteration. These observations give rise to the optimized program in Figure 6. The procedure `setineq` makes sure the inequality is only established if the recursive rules were used.

Formally, we first consider the equivalence of the constraints established.

Proposition 4.1 (Simplified Linked Sequence) *Let $\rhd$ be a binary relation on real numbers from the set $\{<, >, \geq, \leq\}$, extended element wise to vectors of real numbers. Let C be a sequence of constraints $\tilde{X_i} = \mathrm{T}_i(\tilde{X}_{i-1}) \wedge \mathrm{U}_i(\tilde{X_i}) \rhd 0, i = 1, \cdots, n$, such that no other constraints on $\tilde{X}_1 \ldots \tilde{X}_{n-1}$ exist, where, for each $i : 1 < i \leq n$, we have*

$$\tilde{X_i} = \mathrm{T}_i(\tilde{X}_{i-1}) \wedge \mathrm{U}_i(\tilde{X_i}) \rhd 0 \Rightarrow \mathrm{U}_{i-1}(\tilde{X}_{i-1}) \rhd 0 \tag{1}$$

and the T_i and U_i are all ground affine transforms. Then C has exactly the same solutions as

$$\tilde{X_n} = (\mathrm{T}_n \circ \cdots \circ \mathrm{T}_1)(\tilde{X_0}) \wedge \mathrm{U}_n(\tilde{X_n}) \rhd 0.$$

Thus a procedure containing an inequality linked to a variable in a transformation chain can be optimized as before, with the final inequality moved to the wrapper, the others deleted, and some other administrative modifications, as described in [15].

Now, we also consider the problem of early detection of failure. The above translation results in procedures with the same solutions as the original ones. However, for queries that do not have free variables related by the transform, the optimized program may take longer to fail or even go into an infinite loop more often than the original procedure. What we need is a way to actually test the inequality at each stage and then add it at the end (when the base case is reached).

Corollary 4.2 (Early Failure Property) *Let C be a sequence of constraints as defined in Proposition 4.1. Let I be an initial subsequence of k pairs of equality and inequality constraints in C. I is satisfiable if and only if the inequality*

$$\tilde{U}_k((\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)) \rhd 0$$

is satisfiable.

This tells us that it is sufficient, in cases of linked inequalities that are future redundant, to test an inequality that is obtained from the currently accumulated composition of transforms. This is because at each stage k in a threaded transformed procedure, the composition $\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1$ is known. One small complication is that the vector of variables to which the composition of transforms is being applied must be available at each stage of the computation. This translation works correctly (in that it does not delay failure) even when the query arguments that are related by the transform are not free.

This means that a more safe optimization, for non-free queries, results from turning the partially future redundant inequalities into tests. However, in some programs, like the example described above, it is not necessary to test the inequalities even for non-free queries, because the degree of freedom given makes it clear that early failure cannot occur for this calling pattern.

Proposition 4.3 (Satisfiability Property) *Using the definitions in Proposition 4.1, if for each $\tilde{U}_k$ we have $\exists \tilde{X}(\tilde{U}_k(\tilde{X}) \rhd 0)$ satisfiable and $(\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)^{-1}$ exists, the inequality*

$$\tilde{U}_k((\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)) \rhd 0$$

is satisfiable.

This means that for programs with suitably well behaved inequalities, all inequalities but the last one may be removed without delaying failure. As will be shown in Appendix B, knowing that the transformation matrix in this last case is non-singular is such a useful special case that it provides a useful splitting criterion in multiple specialization.

4.2 Chains of Inequalities

Consider a variant of a linear transforming procedure, where instead of collecting a system of equations at each step, a system of inequalities is collected. That is, instead of the transformation $\tilde{U} = \mathrm{T}(\tilde{X})$, we have say $\tilde{U} > \mathrm{T}(\tilde{X})$.

Proposition 4.4 (Simplified Affine Transformation Chain) *Let C be a finite sequence of constraints $\tilde{X}_i \rhd \mathrm{T}_i(\tilde{X}_{i-1}), i = 1, \ldots, n$ such that no other constraints on $\tilde{X}_1 \ldots \tilde{X}_{n-1}$ exist, all of the affine transforms T_i are invertible, and for all T_i, and any $\tilde{X}$ and $\tilde{Y}$, the following ordering and monotonicity conditions apply.*

$$\tilde{X} \rhd \tilde{Y} \Leftrightarrow \mathrm{T}_i(\tilde{X}) \rhd \mathrm{T}_i(\tilde{Y}) \tag{1}$$
$$\mathrm{T}_i(\tilde{X}) \rhd \tilde{X} \tag{2}$$

Then C has exactly the same solutions as $\tilde{X}_n \rhd (\mathrm{T}_n \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)$.

Consider, for example, the program of Figure 7, which can thus be transformed to that of Figure 8, removing the inequalities from the recursion. The simplification is justified because the chain of inequalities $A_n > A_{n-1} + 6, \ldots, A_1 > A_0 + 6$ fits the definition of C in Proposition 4.4, that is, for the relation $>$, and transform $\langle \mathrm{I}, 6 \rangle$, the transform is invertible, and the monotonicity and ordering conditions are met; for all $\tilde{X}, \tilde{Y}$ both $\tilde{X} > \tilde{Y} \Leftrightarrow \langle \mathrm{I}, 6 \rangle(\tilde{X}) > \langle \mathrm{I}, 6 \rangle(\tilde{Y})$ and $\langle \mathrm{I}, 6 \rangle(\tilde{X}) > \tilde{X}$ are satisfied.

It is easy to see that the translations given earlier can be modified appropriately to include ancillary inequalities and inequality chains. Furthermore, one program can collect a number of chains of relations, and, as long as the chains do not share variables, a generalized version of Proposition 4.4 still applies.

```
p(0, Ai, Ao, Bi, Bo) :-
        Ao = Ai + 3,
        Bo > Bi + 4.
p(T, Ai, Ao, Bi, Bo) :-
        T > 0,
        Ak = Ai + 6,
        Bk > Bi + 8,
        p(T - 1, Ak, Ao, Bk, Bo).
```

Figure 7: Linear procedure with a relational chain

```
p'(0, Xe, Ye, Xi, Yi) :-
        Ye = Xe + 3,
        Yi = Xi + 4.
p'(T, Xe, Ye, Xi, Yi) :-
        T > 0,
        p'(T - 1, Xe + 6, Ye, Xi + 8, Yi).

p(T, A1, A2, B1, BO) :-
        p'(T, 0, Ye, 0, Yi),
        A2 = A1 + Ye,
        B2 > B1 + Yi.
```

Figure 8: Optimized procedure with relational chain replaced by single test

5 Empirics

The program translations described in Section 3 can achieve significant time and space savings. These savings are demonstrated in this section for the following programs written in CLP($\mathcal{R}$).

- eulers(Ti,Yi,H,Tf,Yf) :- Yi and Yf are the initial and final values for the ordinary differential equation $\frac{dy}{dt} = y$, solved numerically by Euler's method with a step size of H over the time span Ti to Tf.
- mortgage(P,T,I,R,B) :- B is the balance after T months for a mortgage with interest rate I, repayment rate R, and principal P.
- fib(N,F) :- F is the Nth Fibonacci number. Both the original and optimized procedures are linear recursive.
- analyze(C,V,I) :- The resistive circuit C has voltage drop V for current I. The query uses a ground value for the circuit C consisting of approximately 1000 elements.

The procedures above were compiled using a specially instrumented CLP($\mathcal{R}$) version 1.0, which compiles CLP programs into `CLAM`, or constraint logic arithmetic machine, instructions. This instruction set supports both arithmetic equality and inequality constraints, as well as traditional imperative arithmetic operations for ground numeric terms. Since the current compiler uses constraint instructions even when the query pattern would allow ground arithmetic, the CLP($\mathcal{R}$) compiler was used to produce `CLAM` listings, which were modified by hand and fed into the CLP($\mathcal{R}$) assembler to produce the time and space measurements that follow.

Procedure	Query	Times (s);		Speedup	Nodes	
		original	*optimized*		*original*	*optimized*
Fig. 10, App. B	`eulers(0.0,Yi,0.001,1.0,2.71828)`	.252	.027	(9)	3998	6
Fig. 15, App. B	`mortgage(P,360,1.01,R,B)`	.042	.010	(4)	361	2
Fig. 3, Sec. 3	`fib(25,F)`	.004	.0008	(5)	50	6
Fig. 4, Sec. 3	`analyze(C,V,I)`	.574	.183	(3)	5889	3

The statistics above were gathered on a Sun Sparc ELC with 64 megabytes of memory. Times are in seconds, and space is measured in solver equation records. Both the original and optimized programs use ground arithmetic and tests for loop control variables, and in addition the optimized programs use ground arithmetic to accumulate an affine transform. The time and space savings are due to this replacement of constraint equation solving by ground arithmetic.

The `eulers/5` and `mortgage/5` procedures were optimized by linear threading, Translation 3.1; the `fib/2` procedure, by linear gather, Translation 3.2; and the circuit analysis procedure, `analyze/3`, by Translation 3.7, asymmetrical nonlinear gather.

6 Practical Considerations

The transformations described in this paper are presently being incorporated in a prototype globally optimizing compiler for CLP($\mathcal{R}$). These and most other optimizations are dependent on the availability of good global mode and type analysis. This research augments, rather than replacing, research on global program analysis techniques. However, it may be the case that for such transformations to be fully successful, analyzers will have to be developed in conjunction with a suite of intended transformations, rather than being a completely separate entity.

Furthermore, a number of pragmatic aspects of global compilation, such as controlling multiple specialization, and providing appropriate facilities for programmers to help the compiler via program annotations, need to be explored further before global optimizations of CLP languages can be truly practical. Transforming arithmetic constraints in some approximations to real or rational arithmetic can also be problematic in certain cases. For example, the use of floating point arithmetic in CLP($\mathcal{R}$) is acceptable for many applications, but in some cases programmer control may be needed to avoid transformations that may introduce or aggravate numerical instability.

One relevant area of practical global compilation that is just barely beginning to be investigated is the relationship with programming methodology and program development tools. The optimizations in this paper, like a number of others [9, 11, 13, 15] were developed by either explicitly or implicitly considering the rationale behind the way some class of real programs is structured. That is, they stem from an understanding of programming methodology. Some of the recent work on program development using skeletons and techniques [19, 16, 17] has speculated about the possibility of a well-defined program construction methodology being used to guide optimizing compilation, by associating optimizations with individual skeletons and techniques. In summary, while the potential utility of these and other optimizations is now clear, obtaining the expected benefits reliably in practice can be expected to depend on future research along a number of fronts.

7 Conclusion

We have presented a set of optimizing transformations for a class of Constraint Logic Programs dealing with linear arithmetic. These are dependent on the procedure-level structure of programs, and make use of the inter-relationships between the constraints in a procedure. In conjunction with the traditional form of global mode and type analysis, and other optimizations, these transformations can help make the performance of a range of useful CLP programs approach that of programs in conventional imperative languages.

The optimizations are applicable to a variety of CLP languages that decide linear arithmetic equations and/or inequalities. Being inspired by an easily justifiable programming technique, it is clear that they are applicable to a wide variety of programs spanning a wide range of application areas.

While the optimizations are only now being incorporated into an experimental compiler, our conservative, locally-optimizing hand-compilation of example programs before and after the transformation demonstrate that the improvement in performance is at least substantial, and in some cases dramatic. In conjunction with other optimizations, we believe that these transformations can help to make constraint logic programming a more widely accepted software development tool.

Acknowledgements

This research is supported in part by the National Science Foundation Grant #CCR-9308631. Judith Gardiner helped the authors realize the significance of affine transforms. A number of anonymous referees provided valuable comments on earlier drafts of this paper.

A Affine Transforms

Definition A.1 (Affine Transform) *An affine transform* T *is a pair* $\langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$ *consisting of an* $n \times n$ *matrix* A *and an* n*-vector* $\tilde{\mathrm{a}}$*, where the elements of* A *and* $\tilde{\mathrm{a}}$ *are reals. Such a pair defines a function on* n*-vectors, with application denoted by adjacency. In addition, for transforms* $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$ *and* $\mathrm{T}_b = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$*, binary operations for* $\circ$ *and* $+$ *are defined in terms of matrix multiplication and addition on the elements of these pairs.*

- $\mathrm{T}_a \ \tilde{\mathrm{x}} = \mathrm{A}\tilde{\mathrm{x}} + \tilde{\mathrm{a}}$
- $\mathrm{T}_a + \mathrm{T}_b = \langle \mathrm{A} + \mathrm{B}, \tilde{\mathrm{a}} + \tilde{\mathrm{b}} \rangle$
- $\mathrm{T}_a \circ \mathrm{T}_b = \langle \mathrm{AB}, \mathrm{A}\tilde{\mathrm{b}} + \tilde{\mathrm{a}} \rangle$

Proposition A.1 (Basic Properties) *Let* $\mathcal{A}_n$ *be the set of affine transforms of order* n*, let* $+$ *and* $\circ$ *be defined as above, and define* $\mathrm{T_I} = \langle \mathrm{I}, \tilde{\mathbf{0}} \rangle$ *and* $\mathrm{T_0} = \langle \mathbf{0}, \tilde{\mathbf{0}} \rangle$*, where* I*,* $\mathbf{0}$ *and* $\tilde{\mathbf{0}}$ *are the identity matrix, zero matrix, and zero vector, respectively. Then for any transforms* T_a*,* T_b *and* T_c *from* $\mathcal{A}_n$*, where* $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle$*,* $\mathrm{T}_b = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$*, and* $\mathrm{T}_c = \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle$*, each of the following is true.*

- *The operation* $+$ *has the identity element* $\mathrm{T_0}$*; an additive inverse* $-\mathrm{T}$ *for all transforms* T*; is associative; and commutative.*
- *For the operation* $\circ$*:*
  1. $\mathrm{T}_a \circ \mathrm{T}_I = \mathrm{T}_I \circ \mathrm{T}_a = \mathrm{T}_a$*, so that* T_I *is the identity transform for* $\circ$*; and the inverse* $\mathrm{T}_a^{-1} = \langle \mathrm{A}^{-1}, -\mathrm{A}^{-1}\tilde{\mathrm{a}} \rangle$ *exists if* A^{-1} *exists.*
  2. $(\mathrm{T}_a \circ \mathrm{T}_b) \circ \mathrm{T}_c = \mathrm{T}_a \circ (\mathrm{T}_b \circ \mathrm{T}_c)$*, so that* $\circ$ *is associative.*
  3. $(\mathrm{T}_a + \mathrm{T}_b) \circ \mathrm{T}_c = \mathrm{T}_a \circ \mathrm{T}_c + \mathrm{T}_b \circ \mathrm{T}_c$*, so that the right distributive law holds for* $\circ$*.*

Proof: The properties of $+$ follow directly from its definition and the properties of matrix addition. The properties of $\circ$ follow almost as directly by expanding terms using Definition A.1.

$$
\begin{aligned}
\mathrm{T}_a \circ \mathrm{T}_I &= \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \circ \langle \mathrm{I}, \tilde{\mathbf{0}} \rangle = \langle \mathrm{AI}, \mathrm{A}\tilde{\mathbf{0}} + \tilde{\mathrm{a}} \rangle = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \\
\mathrm{T}_I \circ \mathrm{T}_a &= \langle \mathrm{I}, \tilde{\mathbf{0}} \rangle \circ \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle = \langle \mathrm{IA}, \mathrm{I}\tilde{\mathrm{a}} + \tilde{\mathbf{0}} \rangle = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \\
\mathrm{T}_a \circ \mathrm{T}_a^{-1} &= \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \circ \langle \mathrm{A}^{-1}, -\mathrm{A}^{-1}\tilde{\mathrm{a}} \rangle \\
&= \langle \mathrm{AA}^{-1}, \mathrm{A}(-\mathrm{A}^{-1}\tilde{\mathrm{a}}) + \tilde{\mathrm{a}} \rangle = \mathrm{T}_I
\end{aligned}
$$

$$
\begin{aligned}
(\mathrm{T}_a \circ \mathrm{T}_b) \circ \mathrm{T}_c &= \langle \mathrm{AB}, \mathrm{A}\tilde{\mathrm{b}} + \tilde{\mathrm{a}} \rangle \circ \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle \\
&= \langle \mathrm{ABC}, \mathrm{AB}\tilde{\mathrm{c}} + \mathrm{A}\tilde{\mathrm{b}} + \tilde{\mathrm{a}} \rangle \\
&= \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle \circ (\langle \mathrm{BC}, \mathrm{B}\tilde{\mathrm{c}} + \tilde{\mathrm{b}} \rangle) \\
&= \mathrm{T}_a \circ (\mathrm{T}_b \circ \mathrm{T}_c)
\end{aligned}
$$

$$
\begin{aligned}
(\mathrm{T}_a + \mathrm{T}_b) \circ \mathrm{T}_c &= (\langle \mathrm{A}, \tilde{\mathrm{a}} \rangle + \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle) \circ \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle \\
&= \langle \mathrm{A} + \mathrm{B}, \tilde{\mathrm{a}} + \tilde{\mathrm{b}} \rangle \circ \langle \mathrm{C}, \tilde{\mathrm{c}} \rangle \\
&= \langle \mathrm{AC} + \mathrm{BC}, \mathrm{A}\tilde{\mathrm{c}} + \tilde{\mathrm{a}} + \mathrm{B}\tilde{\mathrm{c}} + \tilde{\mathrm{b}} \rangle \\
&= \mathrm{T}_a \circ \mathrm{T}_c + \mathrm{T}_b \circ \mathrm{T}_c
\end{aligned}
$$

□

Note that left distributivity does not hold, since for $\mathrm{T}_a \circ (\mathrm{T}_b + \mathrm{T}_c)$ and $\mathrm{T}_a \circ \mathrm{T}_b + \mathrm{T}_a \circ \mathrm{T}_c$, $\langle \mathrm{AB} + \mathrm{AC}, \mathrm{A}\tilde{\mathrm{b}} + \mathrm{A}\tilde{\mathrm{c}} + \tilde{\mathrm{a}} \rangle \neq \langle \mathrm{AB} + \mathrm{AC}, \mathrm{A}\tilde{\mathrm{b}} + \mathrm{A}\tilde{\mathrm{c}} + 2\tilde{\mathrm{a}} \rangle$ for $\tilde{\mathrm{a}} \neq \tilde{\mathbf{0}}$.

Proposition A.2 (Composition) *For affine transforms* $\mathrm{T}_k \ldots \mathrm{T}_1$ *applied to a vector* $\tilde{\mathrm{x}}$*, the transforms may be combined freely by* $\circ$ *prior to application.*

$$\mathrm{T}_k \ldots \mathrm{T}_1 \ \tilde{\mathrm{x}} = (\mathrm{T}_k \circ \ldots \circ \mathrm{T}_1) \ \tilde{\mathrm{x}}$$

Proof: The base case of no applications is vacuously true. For the inductive case, consider k applications where we know by the inductive assumption that the proposition is true for $k-1$ applications. Let $\mathrm{T}_k = \langle \mathrm{B}, \tilde{\mathrm{b}} \rangle$, and let $\mathrm{T}_a = \langle \mathrm{A}, \tilde{\mathrm{a}} \rangle = \mathrm{T}_{k-1} \circ \ldots \circ \mathrm{T}_1$. T_a exists since affine transforms are closed under the $\circ$ operation. Then for the inductive case:

$$\begin{aligned} \mathrm{T}_k \mathrm{T}_a \ \tilde{\mathrm{x}} &= \mathrm{T}_k(\mathrm{A}\tilde{\mathrm{x}} + \tilde{\mathrm{a}}) \\ &= \mathrm{B}\mathrm{A}\tilde{\mathrm{x}} + \mathrm{B}\tilde{\mathrm{a}} + \tilde{\mathrm{b}} \\ &= \langle \mathrm{B}\mathrm{A}, \mathrm{B}\tilde{\mathrm{a}} + \tilde{\mathrm{b}} \rangle \ \tilde{\mathrm{x}} \\ &= \mathrm{T}_k \circ \mathrm{T}_a \ \tilde{\mathrm{x}} \end{aligned}$$

The compositions may be performed in any order since $\circ$ is associative. □

Proposition A.3 (Application is Equivalent to Composition Followed by Projection) *An application of a transform* T *to a vector* $\tilde{\mathrm{x}}$ *can be replaced with the projection of the composition of* T *and* $\langle \mathbf{0}, \tilde{\mathrm{x}} \rangle$.

$$\mathrm{T} \ \tilde{\mathrm{x}} = \mathrm{T} \circ \langle \mathbf{0}, \tilde{\mathrm{x}} \rangle \ \tilde{\mathbf{0}}$$

Proof: Immediate from the definitions of application and composition. Note that application of an affine transform to the vector $\tilde{\mathbf{0}}$ gives the vector half of the transform pair. □

We wish to show the correctness of the translations given in Section 3, which, when given a recursively transforming procedure for predicate p, give a translated recursive procedure p', and wrapper to call it.

Search trees will be useful in correctness proofs for the translations. Recall that a search tree is a tree whose root is a query, and whose other nodes are also conjuncts of goals, such that child nodes are reductions of parent nodes. Such trees have branches whenever a procedure goal with multiple clauses is reduced. We will assume that search tree branches are in clause order, and that goals are selected for reduction left-to-right, so that goal reduction in a search tree corresponds to that used in Prolog. Leaf nodes are either [] or *fail*, corresponding to successful reductions of the query or failure, respectively. Reduction paths may also be infinite, corresponding to non-terminating computation. Branch length in a tree will be measured by the number of nodes where a call to p (or p') is reduced, and those nodes will be said to be *procedure* nodes.

Definition A.2 (Isomorphic Search Trees) *Let two search trees S_p and S_q have root queries $p(\ldots)$ and $q(\ldots)$, and let p and q be recursive transforming procedures. The trees S_p and S_q are* isomorphic *if they have the following characteristics:*

- Similar structure: *In S_p, procedure nodes, and the branches from those nodes, are one-to-one and onto the procedure nodes and branches of S_q.*
- Identical leaves: *In S_p, leaves and their labels are one-to-one and onto those of S_q.*

The notion of search tree isomorphism is both less and more restrictive than equivalence, since for success, failure, length of computation, and correctness, isomorphism may not preserve correctness, while equivalence may not preserve computation length.

Proposition 3.1 (Linear Recursive Translations) *Let $P = \langle p, \theta \rangle$ be a linear transforming procedure. The result of applying Translations 3.1, 3.2 or 3.3 to P preserves success, failure, length of computation, and correctness.*

Proof: Let S_p and S_t be search trees for the original procedure and its translated form. Given isomorphism, we can show that correctness is preserved, since by Propositions A.1 and A.2, the forms below are equivalent.

$$\begin{aligned} \tilde{Y} &= \mathrm{T}_2 \ldots \mathrm{T}_2 \ \mathrm{T} \ \mathrm{T}_1 \ldots \mathrm{T}_1 \ \tilde{X} && (\textit{original}) \\ \tilde{Y} &= (\mathrm{T}_2 \circ \ldots \circ (\mathrm{T}_2 \circ (\mathrm{T} \circ (\mathrm{T}_1 \circ \ldots \circ \mathrm{T}_1)))) \ \tilde{X} && (\textit{thread}) \\ \tilde{Y} &= (\mathrm{T}_2 \circ \ldots \circ (\mathrm{T}_2 \circ \mathrm{T} \circ \mathrm{T}_1) \circ \ldots \circ \mathrm{T}_1) \ \tilde{X} && (\textit{gather}) \\ \tilde{Y} &= ((\mathrm{T}_2 \circ \ldots \circ ((\mathrm{T}_2 \circ (\mathrm{T}^{-1})^{-1} \circ \mathrm{T}_1)^{-1})^{-1} \circ \ldots \circ \mathrm{T}_1)^{-1})^{-1} \ \tilde{X} && (\textit{inverse gather}) \end{aligned}$$

Proving isomorphism amounts to showing that a goal of S_p is reduced only when the corresponding goal of S_t would be reduced. The proof is by induction.

For the base case, a search tree with one branch of depth 1, success would lead to the following goals being reduced in S_p and S_t. Three cases are shown for S_t, one for each translation, and ellipses are used for constraints involving terms of $\tilde{Z}$. Trivial reductions unifying variables in calls with variables in heads are not shown.

$$
\begin{array}{rlllllll}
S_p: & p(\tilde{X},\tilde{Y},\tilde{Z}) & & & \ldots, & \tilde{Y}=\mathrm{T}(\tilde{X}), & \ldots, . & \\
S_t: & p(\tilde{X},\tilde{Y},\tilde{Z}) & p'(\mathrm{T_I},\mathcal{Y},\tilde{Z})\ , & \mathcal{X}=\mathrm{T_I}, & \ldots, & \mathcal{Y}=\mathrm{T}\circ\mathcal{X}, & \ldots, & \tilde{Y}=\mathcal{Y}(\tilde{X})\ . \\
 & p(\tilde{X},\tilde{Y},\tilde{Z}) & p'(\mathcal{Y},\tilde{Z})\ , & & \ldots, & \mathcal{Y}=\mathrm{T}, & \ldots, & \tilde{Y}=\mathcal{Y}(\tilde{X})\ . \\
 & p(\tilde{X},\tilde{Y},\tilde{Z}) & p'(\mathcal{Y},\tilde{Z})\ , & & \ldots, & \mathcal{Y}=\mathrm{T}^{-1}, & \ldots, & \tilde{Y}=\mathcal{Y}^{-1}(\tilde{X})\ .
\end{array}
$$

Note that those constraints involving affine transforms necessarily succeed if the left hand side is unbound, since affine transform applications and compositions are closed over their arguments, and the inverses are given to exist; such constraints will be said to be *functional.*

For each of the three cases, in the equations below, (1) follows since we are given that the base rule is chosen for reduction; (2) is by Propositions A.1 and A.2, and the fact that the constraints are functional; and (3) is by definition of the respective translation (thread, gather, or inverse gather) for the base case rule of p. The notation "I" is used to indicate the identity transform, since the form $\mathrm{T_I}$ is too easily confused with T_1, which occurs in the recursive rules.

Thread:

$$
\begin{array}{rcllr}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{Y}=\mathrm{T}(\tilde{X}) & & (1)\\
 & \Leftrightarrow & \mathcal{X}=\mathrm{I}, & & \\
 & & \mathcal{Y}=\mathrm{T}\circ\mathcal{X}, & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (2)\\
 & \Leftrightarrow & p'(\mathrm{I},\mathcal{Y},\tilde{Z}), & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (3)
\end{array}
$$

Gather:

$$
\begin{array}{rcllr}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{Y}=\mathrm{T}(\tilde{X}) & & (1)\\
 & \Leftrightarrow & \mathcal{Y}=\mathrm{T}, & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (2)\\
 & \Leftrightarrow & p'(\mathcal{Y},\tilde{Z}), & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (3)
\end{array}
$$

Inverse Gather:

$$
\begin{array}{rcllr}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{Y}=\mathrm{T}(\tilde{X}) & & (1)\\
 & \Leftrightarrow & \mathcal{Y}=\mathrm{T}^{-1}, & \tilde{Y}=\mathcal{Y}^{-1}(\tilde{X}) & (2)\\
 & \Leftrightarrow & p'(\mathcal{Y},\tilde{Z}), & \tilde{Y}=\mathcal{Y}^{-1}(\tilde{X}) & (3)
\end{array}
$$

In the inductive case, we know that in order to increase the depth of a search tree, a goal of p or p' must be reduced using a recursive clause. Schematic forms for the recursive rule of p, and the three rules for p' resulting from translation, are given below.

$$
\begin{array}{rllllllll}
S_p: & p(\tilde{X},\tilde{Y},\tilde{Z}) \text{ :- } \ldots, & \tilde{U}=\mathrm{T}_1(\tilde{X}), & \ldots, & p(\tilde{U},\tilde{V},\tilde{Z}), & \ldots, & \tilde{Y}=\mathrm{T}_2(\tilde{V}), & \ldots, . \\
S_t: & p'(\mathrm{T_I},\mathcal{Y},\tilde{Z}) \text{ :- } \ldots, & \mathcal{U}=\mathrm{T}_1\circ\mathcal{X}, & \ldots, & p'(\mathcal{U},\mathcal{V},\tilde{Z}), & \ldots, & \mathcal{Y}=\mathrm{T}_2\circ\mathcal{V}, & \ldots, . \\
 & p'(\mathcal{Y},\tilde{Z}) \text{ :- } \ldots, & & & p'(\mathcal{X},\tilde{Z}), & \ldots, & \mathcal{Y}=\mathrm{T}_2\circ\mathcal{X}\circ\mathrm{T}_1, & \ldots, . \\
 & p'(\mathcal{Y},\tilde{Z}) \text{ :- } \ldots, & & & p'(\mathcal{X},\tilde{Z}), & \ldots, & \mathcal{Y}=(\mathrm{T}_2\circ\mathcal{X}^{-1}\circ\mathrm{T}_1)^{-1}, & \ldots, .
\end{array}
$$

For each of the three cases, in the equations below, (1) follows since a recursive rule must be used to reduce the top level call to p; (2) is by the inductive hypothesis; (3) and (4) are by Propositions A.1 and A.2, and the fact that the transform constraints are functional; and (5) is by definition of the respective translation (thread, gather, or inverse gather) for the recursive rule of p.

Thread:

$$
\begin{array}{lllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}),\quad p(\tilde{U},\tilde{V},\tilde{Z}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (1)\\
 & \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}),\quad p'(\mathrm{I},\mathcal{V},\tilde{Z}),\quad \tilde{V}=\mathcal{V}(\tilde{U}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (2)\\
 & \Leftrightarrow & \mathcal{U}=\mathrm{T}_1\circ\mathrm{I},\quad p'(\mathcal{U},\mathcal{V},\tilde{Z}), & \tilde{Y}=\mathrm{T}_2\circ\mathcal{V}(\tilde{X}) & (3)\\
 & \Leftrightarrow & \mathcal{X}=\mathrm{I}, & & \\
 & & \mathcal{U}=\mathrm{T}_1\circ\mathcal{X},\quad p'(\mathcal{U},\mathcal{V},\tilde{Z}),\quad \mathcal{Y}=\mathrm{T}_2\circ\mathcal{V}, & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (4)\\
 & \Leftrightarrow & p'(\mathrm{I},\mathcal{Y},\tilde{Z}), & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (5)
\end{array}
$$

Gather:

$$
\begin{array}{lllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}),\quad p(\tilde{U},\tilde{V},\tilde{Z}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (1)\\
 & \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}),\quad p'(\mathcal{V},\tilde{Z}),\quad \tilde{V}=\mathcal{V}(\tilde{U}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (2)\\
 & \Leftrightarrow & p'(\mathcal{V},\tilde{Z}),\quad \tilde{V}=\mathcal{V}\circ\mathrm{T}_1(\tilde{X}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (3)\\
 & \Leftrightarrow & p'(\mathcal{V},\tilde{Z}),\quad \mathcal{Y}=\mathrm{T}_2\circ\mathcal{V}\circ\mathrm{T}_1, & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (4)\\
 & \Leftrightarrow & p'(\mathcal{Y},\tilde{Z}), & \tilde{Y}=\mathcal{Y}(\tilde{X}) & (5)
\end{array}
$$

Inverse Gather:

$$
\begin{array}{lllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}),\quad p(\tilde{U},\tilde{V},\tilde{Z}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (1)\\
 & \Leftrightarrow & \tilde{U}=\mathrm{T}_1(\tilde{X}),\quad p'(\mathcal{V},\tilde{Z}),\quad \tilde{V}=\mathcal{V}^{-1}(\tilde{U}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (2)\\
 & \Leftrightarrow & p'(\mathcal{V},\tilde{Z}),\quad \tilde{V}=\mathcal{V}^{-1}\circ\mathrm{T}_1(\tilde{X}), & \tilde{Y}=\mathrm{T}_2(\tilde{V}) & (3)\\
 & \Leftrightarrow & p'(\mathcal{V},\tilde{Z}),\quad \mathcal{Y}=(\mathrm{T}_2\circ\mathcal{V}^{-1}\circ\mathrm{T}_1)^{-1}, & \tilde{Y}=\mathcal{Y}^{-1}(\tilde{X}) & (4)\\
 & \Leftrightarrow & p'(\mathcal{Y},\tilde{Z}), & \tilde{Y}=\mathcal{Y}^{-1}(\tilde{X}) & (5)
\end{array}
$$

□

Proposition 3.2 (Multilinear Recursive Translations) *Let $P = \langle p,\theta\rangle$ be a multilinear transforming procedure. The result of applying Translations 3.4 or 3.5 to P preserves success, failure, length of computation, and correctness.*

Proof: (sketch) Generalization from the proof of Proposition 3.1 is straight forward. Again translation gives isomorphic search trees, and again the thread and gather forms compose the same sequence of transforms, with the thread translation composing from right to left, and the gather translation grouping at each base case transform, so that correctness is also preserved. □

For nonlinear transforming procedures, the results of the individual recursive calls within a clause are combined rather than threaded, so that we are comparing summation of vectors for the original procedure with summation of transforms for the translated one.

Proposition A.4 (Summation Distributes Over Application) *The sum of transform applications is equal to the application of the sum of the transforms.*

$$\sum_{i=1}^{n}\mathrm{T}_i(\tilde{X}) = \left(\sum_{i=1}^{n}\mathrm{T}_i\right)(\tilde{X})$$

Proof: Immediate from the the right distributivity of transform composition, Proposition A.1, and the interchangeability of application and composition, Proposition A.3. □

Proposition 3.3 (Nonlinear Recursive Translations) *Let $P = \langle p, \theta \rangle$ be a nonlinear transforming procedure, and let any transform inverses required by Translation 3.7 exist. The result of applying Translation 3.7, or 3.6 where applicable, to P preserves success, failure, length of computation, and correctness.*

Proof: It's sufficient to consider the general case, Translation 3.7.

Let S_p and S_t be search trees for the original procedure and its translated form. Proving isomorphism amounts to showing that a goal of S_p is reduced only when the corresponding goal of S_t would be reduced. The proof is by induction.

Remember that Translation 3.7 applied to asymmetric nonlinear transforming procedures, which may have two types of rules, distinguished by whether transforms are applied to the parameter vectors $\tilde{X}$ or $\tilde{Y}$. Since it's necessary to translate the rules so that compositions are consistently accumulated using a single transform parameter, the form applying transforms to $\tilde{X}$ is arbitrarily chosen to be primary, and instances of the other form are rewritten using inverses.

For the base case, a search tree with one branch of depth 1, there are two cases, depending on whether a base rule instance is symmetrical with the primary form or not. The table below gives the goals that would be reduced in case of success, for both search trees, and for the symmetric and asymmetric forms, with the symmetric case indicated by $\Rightarrow$, and the asymmetric case, by $\rightleftharpoons$. As before, ellipses are used for constraints involving terms of $\tilde{Z}$, and trivial reductions unifying variables in calls with variables in heads are not shown.

$$
\begin{array}{rlllll}
S_p, \Rightarrow & p(\tilde{X}, \tilde{Y}, \tilde{Z})\ , & & \ldots, & \tilde{Y} = \mathrm{T}(\tilde{X}), & \ldots, . \\
\rightleftharpoons & p(\tilde{X}, \tilde{Y}, \tilde{Z})\ , & & \ldots, & \tilde{X} = \mathrm{T}(\tilde{Y}), & \ldots, . \\
S_t, \Rightarrow & p(\tilde{X}, \tilde{Y}, \tilde{Z})\ , & p'(\mathcal{Y}, \tilde{Z})\ , & \ldots, & \mathcal{Y} = \mathrm{T}, & \ldots, \quad \tilde{Y} = \mathcal{Y}(\tilde{X})\ . \\
\rightleftharpoons & p(\tilde{X}, \tilde{Y}, \tilde{Z})\ , & p'(\mathcal{Y}, \tilde{Z})\ , & \ldots, & \mathcal{Y} = \mathrm{T}^{-1}, & \ldots, \quad \tilde{Y} = \mathcal{Y}(\tilde{X})\ .
\end{array}
$$

The proof for the base case is identical either to the linear gather or inverse gather base cases, respectively, since the translations are identical.

For each of the three cases, in the equations below, (1) follows since we are given that the base rule is chosen for reduction; (2) is by Propositions A.1 and A.2, and the fact that the inverse is given to exist; and (3) is by definition of the base case translation for symmetric and asymmetric rules.

$\Rightarrow$

$$
\begin{aligned}
p(\tilde{X}, \tilde{Y}, \tilde{Z}) \quad &\Leftrightarrow \quad \tilde{Y} = \mathrm{T}(\tilde{X}) && (1)\\
&\Leftrightarrow \quad \mathcal{Y} = \mathrm{T}, \qquad \tilde{Y} = \mathcal{Y}(\tilde{X}) && (2)\\
&\Leftrightarrow \quad p'(\mathcal{Y}, \tilde{Z}), \qquad \tilde{Y} = \mathcal{Y}(\tilde{X}) && (3)
\end{aligned}
$$

$\rightleftharpoons$

$$
\begin{aligned}
p(\tilde{X}, \tilde{Y}, \tilde{Z}) \quad &\Leftrightarrow \quad \tilde{X} = \mathrm{T}(\tilde{Y}) && (1)\\
&\Leftrightarrow \quad \mathcal{Y} = \mathrm{T}^{-1}, \qquad \tilde{Y} = \mathcal{Y}(\tilde{X}) && (2)\\
&\Leftrightarrow \quad p'(\mathcal{Y}, \tilde{Z}), \qquad \tilde{Y} = \mathcal{Y}(\tilde{X}) && (3)
\end{aligned}
$$

In the inductive case, we know that in order to increase the depth of a search tree, a goal of p or p' must be reduced using a recursive clause. There are two cases, one each for symmetrical and asymmetrical rules. Schematic forms for the rules of p, and those of p' resulting from translation, are given below. In this case the ellipses stand not only for goals involving $\tilde{Z}$, which are unchanged by translation, but also for constraints indexed by i, with $1 \leq i \leq c$, where there are c recursive calls.

$$
\begin{array}{rllll}
S_p, \Rightarrow & p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } \ldots, & \tilde{X}_i = \mathrm{T}_{1_i}(\tilde{X}), \ldots, & p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z}), \ldots, & \tilde{Y} = \sum \mathrm{T}_{2_i}(\tilde{Y}_i), \ldots . \\
\rightleftharpoons & p(\tilde{X}, \tilde{Y}, \tilde{Z}) \text{ :- } \ldots, & \tilde{Y}_i = \mathrm{T}_{1_i}(\tilde{Y}), \ldots, & p(\tilde{X}_i, \tilde{Y}_i, \tilde{Z}), \ldots, & \tilde{X} = \sum \mathrm{T}_{2_i}(\tilde{X}_i), \ldots . \\
S_t, \Rightarrow & p'(\mathcal{Y}, \tilde{Z}) \text{ :- } \ldots, & & p'(\mathcal{Y}_i, \tilde{Z}), \ldots, & \mathcal{Y} = \sum (\mathrm{T}_{2_i} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_i}), \ldots . \\
\rightleftharpoons & p'(\mathcal{Y}, \tilde{Z}) \text{ :- } \ldots, & & p'(\mathcal{Y}_i, \tilde{Z}), \ldots, & \mathcal{Y} = [\sum (\mathrm{T}_{2_i} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i})]^{-1}, \ldots .
\end{array}
$$

For each case, in the equations below, (1) follows since a recursive rule must be used to reduce the top level call to p; (2) is by the inductive hypothesis; (3) and (4) are by Propositions A.1 and A.2, and the fact that the transform constraints are functional; (5) is by distributivity of summation over application, Proposition A.4; (6) is again by Proposition A.2, and the fact that the constraint defining $\mathcal{Y}$ is functional; for the asymmetrical case, (7) is by Proposition A.1 and the existence of inverses; and for both cases, the last equation is by definition of the respective translation (symmetrical or asymmetrical nonlinear) for the recursive rule of p.

$\Rightarrow$

$$\begin{array}{lllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow \tilde{X}_i = \mathrm{T}_{1_i}(\tilde{X}), & p(\tilde{X}_i,\tilde{Y}_i,\tilde{Z}), & \tilde{Y} = \sum \mathrm{T}_{2_i}(\tilde{Y}_i) & (1) \\
 & \Leftrightarrow \tilde{X}_i = \mathrm{T}_{1_i}(\tilde{X}), & & & \\
 & \quad p'(\mathcal{Y}_i,\tilde{Z}), & \tilde{Y}_i = \mathcal{Y}_i(\tilde{X}_i), & \tilde{Y} = \sum \mathrm{T}_{2_i}(\tilde{Y}_i) & (2) \\
 & \Leftrightarrow p'(\mathcal{Y}_i,\tilde{Z}), & \tilde{Y}_i = \mathcal{Y}_i \circ \mathrm{T}_{1_i}(\tilde{X}), & \tilde{Y} = \sum \mathrm{T}_{2_i}(\tilde{Y}_i) & (3) \\
 & \Leftrightarrow p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{Y} = \sum((\mathrm{T}_{2_i} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_i})(\tilde{X})) & (4) \\
 & \Leftrightarrow p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{Y} = (\sum \mathrm{T}_{2_i} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_i})(\tilde{X}) & (5) \\
 & \Leftrightarrow p'(\mathcal{Y}_i,\tilde{Z}), & & \mathcal{Y} = \sum \mathrm{T}_{2_i} \circ \mathcal{Y}_i \circ \mathrm{T}_{1_i}, \;\; \tilde{Y} = \mathcal{Y}(\tilde{X}) & (6) \\
 & \Leftrightarrow p'(\mathcal{Y},\tilde{Z}), & \tilde{Y} = \mathcal{Y}(\tilde{X}) & & (7)
\end{array}$$

$\rightleftharpoons$

$$\begin{array}{lllll}
p(\tilde{X},\tilde{Y},\tilde{Z}) & \Leftrightarrow \tilde{Y}_i = \mathrm{T}_{1_i}(\tilde{Y}), & p(\tilde{X}_i,\tilde{Y}_i,\tilde{Z}), & \tilde{X} = \sum \mathrm{T}_{2_i}(\tilde{X}_i) & (1) \\
 & \Leftrightarrow \tilde{Y}_i = \mathrm{T}_{1_i}(\tilde{Y}), & & & \\
 & \quad p'(\mathcal{Y}_i,\tilde{Z}), & \tilde{X}_i = \mathcal{Y}_i^{-1}(\tilde{Y}_i), & \tilde{X} = \sum \mathrm{T}_{2_i}(\tilde{X}_i) & (2) \\
 & \Leftrightarrow p'(\mathcal{Y}_i,\tilde{Z}), & \tilde{X}_i = \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i}(\tilde{Y}), & \tilde{X} = \sum \mathrm{T}_{2_i}(\tilde{X}_i) & (3) \\
 & \Leftrightarrow p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{X} = \sum(\mathrm{T}_{2_i} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i}(\tilde{Y})) & (4) \\
 & \Leftrightarrow p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{X} = (\sum \mathrm{T}_{2_i} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i})(\tilde{Y}) & (5) \\
 & \Leftrightarrow p'(\mathcal{Y}_i,\tilde{Z}), & & \tilde{Y} = (\sum \mathrm{T}_{2_i} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i})^{-1}(\tilde{X}) & (6) \\
 & \Leftrightarrow p'(\mathcal{Y}_i,\tilde{Z}), & & \mathcal{Y} = (\sum \mathrm{T}_{2_i} \circ \mathcal{Y}_i^{-1} \circ \mathrm{T}_{1_i})^{-1}, \;\; \tilde{Y} = \mathcal{Y}(\tilde{X}) & (7) \\
 & \Leftrightarrow p'(\mathcal{Y},\tilde{Z}), & \tilde{Y} = \mathcal{Y}(\tilde{X}) & & (8)
\end{array}$$

$\square$

Proposition 4.1 (Simplified Linked Sequence) *Let $\rhd$ be a binary relation on real numbers from the set $\{<,>,\geq,\leq\}$, extended element wise to vectors of real numbers. Let C be a sequence of constraints $\tilde{X}_i = \mathrm{T}_i(\tilde{X}_{i-1}) \wedge \mathrm{U}_i(\tilde{X}_i) \rhd 0, i = 1,\cdots,n$, such that no other constraints on $\tilde{X}_1 \ldots \tilde{X}_{n-1}$ exist, where, for each $i : 1 < i \leq n$, we have*

$$\tilde{X}_i = \mathrm{T}_i(\tilde{X}_{i-1}) \wedge \mathrm{U}_i(\tilde{X}_i) \rhd 0 \Rightarrow \mathrm{U}_{i-1}(\tilde{X}_{i-1}) \rhd 0 \qquad (1)$$

and the T_i and U_i are all ground affine transforms. Then C has exactly the same solutions as

$$\tilde{X}_n = (\mathrm{T}_n \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0) \wedge \mathrm{U}_n(\tilde{X}_n) \rhd 0.$$

Proof:

Since by Proposition A.2, $\tilde{X}_1 = \mathrm{T}_1(\tilde{X}_0), \ldots, \tilde{X}_n = \mathrm{T}_n(\tilde{X}_{n-1})$ and $\tilde{X}_n = (\mathrm{T}_n \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)$ are equivalent, we need only to show that solutions are preserved for the inequalities. In the following, let C' refer to the simplified set of constraints.

$\Rightarrow$ By definition of C we have that $\mathrm{U}_i(\tilde{X}_i) \rhd 0, i = 1, \cdots, n$, which subsumes $\mathrm{U}_n(\tilde{X}_n) \rhd 0$.

$\Leftarrow$ By induction.

The base case of $n = 1$ is vacuously true since C and C' are the same.

For the inductive case, where $n > 1$ and the proposition is true for $n - 1$, the first statement below is by definition of C', the second by assumption, the third by the inductive hypothesis, and the fourth by conjunction of the previous statements.

$$\begin{aligned}
&\mathrm{U}_n(\tilde{X}_n) \rhd 0 && (1)\\
\mathrm{U}_n(\tilde{X}_n) \rhd 0 \quad &\Rightarrow \quad \mathrm{U}_{n-1}(\tilde{X}_{n-1}) \rhd 0 && (2)\\
\mathrm{U}_{n-1}(\tilde{X}_{n-1}) \rhd 0 \quad &\Rightarrow \quad \mathrm{U}_i(\tilde{X}_i) \rhd 0, i = 1, \cdots, n-2 && (3)\\
& \mathrm{U}_i(\tilde{X}_i) \rhd 0, i = 1, \cdots, n && (4)
\end{aligned}$$

□

Corollary 4.2 (Early Failure Property) *Let C be a sequence of constraints as defined in Proposition 4.1. Let I be an initial subsequence of k pairs of equality and inequality constraints in C. I is satisfiable if and only if the inequality*

$$\tilde{U}_k((\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)) \rhd 0$$

is satisfiable.

Proof:

By Proposition 4.1, with n replaced by k throughout. □

Proposition 4.3 (Satisfiability Property) *Using the definitions in Proposition 4.1, if for each $\tilde{U}_k$ we have $\exists \tilde{X}(\tilde{U}_k(\tilde{X}) \rhd 0)$ satisfiable and $(\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)^{-1}$ exists, the inequality*

$$\tilde{U}_k((\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)) \rhd 0$$

is satisfiable.

Proof:

$\tilde{X}_0 = (\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)^{-1} (\tilde{X}_k)$ exists, by the existence of some vector $\tilde{X}_k$ such that $(\tilde{U}_k(\tilde{X}_k) \rhd 0)$, by the existence of the inverse, and the fact that by definition of C, there are no other constraints on $\tilde{X}_0$. In addition, by definition of C we have $\tilde{X}_i = \mathrm{T}_i(\tilde{X}_{i-1}) \wedge \mathrm{U}_i(\tilde{X}_i) \rhd 0 \Rightarrow \mathrm{U}_{i-1}(\tilde{X}_{i-1}) \rhd 0$, so that by a straight forward induction $\tilde{U}_k((\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)) \rhd 0$ must be satisfiable. □

Proposition 4.4 (Simplified Affine Transformation Chain) *Let C be a finite sequence of constraints $\tilde{X}_i \rhd \mathrm{T}_i(\tilde{X}_{i-1}), i = 1, \ldots, n$ such that no other constraints on $\tilde{X}_1 \ldots \tilde{X}_{n-1}$ exist, all of the affine transforms T_i are invertible, and for all T_i, and any $\tilde{X}$ and $\tilde{Y}$, the following ordering and monotonicity conditions apply.*

$$\begin{aligned}
\tilde{X} \rhd \tilde{Y} \Leftrightarrow \mathrm{T}_i(\tilde{X}) \rhd \mathrm{T}_i(\tilde{Y}) && (1)\\
\mathrm{T}_i(\tilde{X}) \rhd \tilde{X} && (2)
\end{aligned}$$

Then C has exactly the same solutions as $\tilde{X}_n \rhd (\mathrm{T}_n \circ \cdots \circ \mathrm{T}_1)(\tilde{X}_0)$.

Proof:

$\Rightarrow$ By induction. Let T_j be $\mathrm{T}_k \circ \cdots \circ \mathrm{T}_1$, and let $\tilde{X}_j = \mathrm{T}_j(\tilde{X}_0)$.

For $k = 1$, where there are no compositions, the proposition is vacuously true; $\tilde{X}_1 \rhd \mathrm{T}_1(\tilde{X}_0)$ has the same solutions as itself.

For the inductive case, the proposition is true for k, and we wish to show that it is true for $k+1$. Then for the inequalities below, 1 follows by definition of C, 2 by the inductive hypothesis, 3 by 1 and the monotonicity condition, 4 follows from 1 and 3 by transitivity, and 5 from 4 by Proposition A.2, composition of transforms.

$$\begin{aligned} \tilde{X}_{k+1} &\rhd \mathrm{T}_{k+1}(\tilde{X}_k) && (1)\\ \tilde{X}_k &\rhd \mathrm{T}_j(\tilde{X}_0) && (2)\\ \mathrm{T}_{k+1}(\tilde{X}_k) &\rhd \mathrm{T}_{k+1}(\mathrm{T}_j(\tilde{X}_0)) && (3)\\ \tilde{X}_{k+1} &\rhd \mathrm{T}_{k+1}(\mathrm{T}_j(\tilde{X}_0)) && (4)\\ \tilde{X}_{k+1} &\rhd \mathrm{T}_{k+1}\circ \mathrm{T}_j(\tilde{X}_0) && (5) \end{aligned}$$

$\Leftarrow$ By induction.

For $i=1$, $\tilde{X}_1 \rhd \mathrm{T}_1(\tilde{X}_0)$ is a solution to itself.

For $i+1$, let $\tilde{X}_s \rhd \mathrm{T}_{i+1}\circ\cdots\circ \mathrm{T}_1(\tilde{X}_0)$ be satisfiable. Also, for brevity, let the result of an application $\mathrm{T}_i\circ\cdots\circ\mathrm{T}_1(\tilde{X}_0)$ be named $\tilde{A}_i$. We need to construct some $\tilde{X}$ such that $\tilde{X}_s \rhd \mathrm{T}_{i+1}(\tilde{X})$, $\tilde{X} \rhd \mathrm{T}_i(\tilde{A}_{i-1})$, and then by the inductive hypothesis the $\tilde{X}$ can be used to construct a solution to the subsequence of constraints $\tilde{X}_i \rhd \mathrm{T}_i(\tilde{X}_{i-1}),\ldots,\tilde{X}_1 \rhd \mathrm{T}_1(\tilde{X}_0)$.

By the invertibility of T_{i+1}, and using $\tilde{X}_s$, we can define $\tilde{X}_\epsilon$.

$$\tilde{X}_\epsilon = (\mathrm{T}_{i+1}^{-1}(\tilde{X}_s)+\tilde{A}_i)/2$$

Since by definition $\tilde{X}_s \rhd \mathrm{T}_{i+1}(\tilde{A}_i)$, then by cancellation of inverses $\mathrm{T}_{i+1}^{-1}(\tilde{X}_s) \rhd \tilde{A}_i$, and by construction $\tilde{X}_\epsilon$ lies between $\mathrm{T}_{i+1}^{-1}(\tilde{X}_s)$ and $\tilde{A}_i$.

$$\mathrm{T}_{i+1}^{-1}(\tilde{X}_s) \rhd \tilde{X}_\epsilon \rhd \tilde{A}_i$$

Then by the ordering assumption $\tilde{X}_s \rhd \mathrm{T}_{i+1}(\tilde{X}_\epsilon)$, and by definition of $\tilde{A}_i$, $\tilde{X}_\epsilon \rhd \mathrm{T}_i(\tilde{A}_{i-1})$, so that $\tilde{X}_\epsilon$ is the desired $\tilde{X}$.

$$\tilde{X}_s \rhd \mathrm{T}_{i+1}(\tilde{X}_\epsilon),\ \tilde{X}_\epsilon \rhd \mathrm{T}_i(\tilde{A}_{i-1})$$

□

Proposition A.5 (Composition Time Complexity) *Let P and Q be defined as below, where the argument vectors and transforms are of order n. In addition, let matrix multiplication of ground terms be done using the ordinary iterative algorithm, and constraint testing be done using an incremental Gaussian solver, where the solver stores constraints in parametric form, so that only related terms need be considered when a test is made.*

$$\begin{aligned} P = \langle p,(arith, arith)\rangle \quad &where \quad p(\tilde{X},\tilde{Y}) \text{ :- } \tilde{Y}=\mathrm{T}(\tilde{X}).\\ Q = \langle q,(numeric, arith)\rangle \quad &where \quad q(\mathcal{X},\mathcal{Y}) \text{ :- } \mathcal{Y}=\mathrm{T}\circ\mathcal{X}. \end{aligned}$$

Then both P and Q have the same time complexity.

Proof: For P, the solver is used. Application requires n dot products of n terms each. Since the parametric form that accumulates each dot product is built incrementally, those forms will include from 1 to n related terms, so that testing satisfiability as each term is added involves $\Theta(n)$ terms, for $\Theta(n^3)$ time overall. For Q, since $\mathcal{X}$ is ground, the composition is computed by ordinary matrix multiplication and addition, followed by n independent equality constraints each taking unit time. It follows that both P and Q have the same time complexity for the given model of computation. □

Note that the constant factor for P is larger than for Q if ground multiplications can be performed more quickly than solver satisfiability tests.

B Applications

We now show how these optimizations apply to two groups of application programs. We are concerned with showing how the optimizations apply at different levels of complexity, and with the actual resulting code.

B.1 Ordinary Differential Equations

The numerical solution of systems of ordinary differential equations (ODEs) illustrates constraint elimination in linear recursive relations. Figure 9 defines a solver using Euler's method, with definition of the procedure `dydt(T,Y,D)` determining the equation to be solved. The techniques described are applicable to any system of linear ODEs, that is systems with each member of the form $\frac{dy}{dt} = f_0(t)y + f_1(t)$, where f_0 and f_1 are arbitrary functions in t. Skuppin [18] demonstrates the use of ODE solvers written in CLP($\mathcal{R}$) and Prolog III, and shows the value of unusual query patterns in practical applications.

A solver such as `eulers/5` can be used to solve both initial value and two-point boundary value problems. In the first case we have calling pattern `eulers(num, num, num, num, free)`, and y_n is found by ground computation from y_0. In the typical form of the second case, each of a system of equations is constrained by either the initial or final value, and for those equations with unknown initial value, we have calling pattern `eulers(num, free, num, num, num)`, recursion builds a chain of equality constraints, and this chain collapses to solve for y_0 when the base case is reached.

```
eulers(Ti,Yn,H,Tn,Yn) :- Ti + H >  Tn.
eulers(Ti,Yi,H,Tn,Yn) :- Ti + H <= Tn,
          dydt(Ti,Yi,F1),
          Tj = Ti + H,
          Yj = Yi + H * F1,
          eulers(Tj,Yj,H,Tn,Yn).
```

Figure 9: Simple Linear ODE Solver

```
ode1(Ti,Yi,H,Tn,Yn) :-                 %  -         -      - -
               H1 = H + 1,             %  | 1+H   0   |  , | 0 |
               solve1(Ti,1,H,Tn,An,H1), % |  0   1+H  |    | 0 |
               Yn = Yi * An.           %  -         -      - -

solve1(Ti,An,H,Tn,An,H1) :- Ti + H >= Tn.
solve1(Ti,Ai,H,Tn,An,H1) :- Ti + H <= Tn,
               Tj = Ti + H,
               Aj = Ai * H1,
               solve1(Tj,Aj,H,Tn,An,H1).
```

Figure 10: Optimized Solver for $\frac{dy}{dt} = y$

Three simple linear ODEs are used as examples. Procedures optimized to solve these equations for unknown initial value are given in Figures 10, 11 and 12. Clause bodies for `dydt(T,Y,D)` have been folded into the solver, and a wrapper is used to provide accumulator variables.

In the first example, $\frac{dy}{dt} = y$, `eulers/5` builds a series of simple constraints between y_{i+1} and y_i with additive constants. These additive constants are explicitly summed in the optimized program. For $\frac{dy}{dt} = 3t^2y$ we use a multiplicative accumulator initialized to one in place of the additive accumulator in the first example. Finally, for $\frac{dy}{dt} = y + t$, the calculation is described by the recurrence $y_n = y_0(1+h)^{n-1} + h\sum_{i=0}^{n-1} t_i(1+h)^{n-1-i}$, where the two terms use a multiplicative and additive accumulator, respectively.

In a problem such as $\frac{dy}{dt} = t\,sin(y)$, the derivative is nonlinear in y, constraints would be delayed, and the approach above can not be used to solve for unknown initial value, or to relate initial and final values.

```
ode2(Ti,Yi,H,Tn,Yn) :-
        solve2(Ti,1,H,Tn,Mn),
        Yn = Yi * Mn.

solve2(Ti,Yn,H,Tn,Yn) :- Ti + H >  Tn.
solve2(Ti,Mi,H,Tn,Yn) :- Ti + H <= Tn,
        F1 = Mi * Ti * Ti * 3,
        Tj = Ti + H,
        Mj = Mi + H * F1,
        solve2(Tj,Mj,H,Tn,Yn).
```

Figure 11: Optimized Solver for $\frac{dy}{dt} = 3t^2y$

```
ode3(Ti,Yi,H,Tn,Yn) :-
        solve3(Ti,0,1,H,Tn,An,Mn),
        Yn = Yi * Mn + An.

solve3(Ti,An,Mn,H,Tn,An,Mn) :- Ti + H >  Tn.
solve3(Ti,Ai,Mi,H,Tn,An,Mn) :- Ti + H <= Tn,
        H1 = H + 1,
        Aj = Ai * H1 + Ti * H,
        Mj = Mi * H1,
        Tj = Ti + H,
        solve3(Tj,Aj,Mj,H,Tn,An,Mn).
```

Figure 12: Optimized Solver for $\frac{dy}{dt} = y + t$

Note that ODE solvers based on Runge-Kutta methods can be optimized in the same way as the simple solver given in Figure 9.

B.2 The Mortgage Program

The mortgage program, shown in simplified form in Figure 13, is a frequently used benchmark for CLP optimization techniques since:

1. it allows a large range of meaningful and useful query patterns,
2. the different query patterns give rise to a wide range of run-time behaviors, amenable to a wide range of optimizations, and
3. these phenomena are representative of other real CLP programs, although to our knowledge no other single, useful program has exhibited all of them together.

To defer the discussion of ancillary inequalities, we first consider optimization of the simplified mortgage program. We consider two calling patterns:

- $\theta_1 = \mathtt{mg}(\mathtt{free}, \mathtt{num}, \mathtt{num}, \mathtt{num}, \mathtt{free})$
  Such queries establish a linear equation between the arguments corresponding to `P` and `R` as the answer; an example is given in Figure 14.
- $\theta_2 = \mathtt{mg}(\mathtt{free}, \mathtt{num}, \mathtt{num}, \mathtt{free}, \mathtt{free})$
  Such queries set up a linear equation between `P`, `R` and `B` as an answer. The resulting program is shown in Figure 15, with some additional optimizations based on observing simple invariants.

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

Figure 13: Simplified mortgage program

```
mg(P, T, I, R, B) :-
        mg1(T, I, R, 1, 0, A1, A2),
        B = P*A1 + A2.

mg1(T, I, R, A1, A2, O1, O2) :-
        T > 1,
        T1 = T - 1,
        mg1(T1, I, R, I*A1, I*A2 - R, O1, O2).
mg1(T, I, R, A1, A2, O1, O2) :-
        T = 1,
        O1 = I*A1,
        O2 = I*A2 - R.
```

Figure 14: Simplified mortgage program optimized for calling pattern θ_1

```
mg(P, T, I, R, B) :-
        mg1(T, I, 1, 0, A1, A2),
        B = P*A1 + R*A2.

mg1(T, I, A1, A2, O1, O2) :-
        T > 1,
        T1 = T - 1,
        mg1(T1, I, I*A1, I*A2 - 1, O1, O2).
mg1(T, I, A1, A2, O1, O2) :-
        T = 1,
        O1 = I*A1,
        O2 = I*A2 - 1.
```

Figure 15: Simplified mortgage program optimized for calling pattern θ_2

Now, let us consider the full version of the mortgage program, which has the additional constraint P > 0 in the recursive rule, before the call. This constraint is partially future redundant if R and I can be guaranteed to be positive. These two variables are passed through the program unchanged. Furthermore, in this case the transform is invertible, and hence the inequality not only need not be added at each step, but also need not be tested. In summary, for the pattern θ_1 it is well worth splitting the program along these lines, obtaining the program shown in Figure 16, where the call to gmg uses the general (full) mortgage program.

```
mg(P, T, I, R, B) :-
        I > 0, R > 0,
        smg(P, T, I, R, B).
mg(P, T, I, R, B) :-
        ( I <= 0 ; R <= 0 ),
        gmg(P, T, I, R, B).

setineq(a, _).
setineq(b, P) :- P > 0.

smg(P, T, I, R, B) :-
        mg1(T, I, R, 1, 0, A1, A2, Last),
        B = P*A1 + A2,
        setineq(Last, P).

mg1(T, I, R, A1, A2, O1, O2, b) :-
        T > 1,
        T1 = T - 1,
        mg1(T1, I, R, I*A1, I*A2 - R, O1, O2, _).
mg1(T, I, R, A1, A2, O1, O2, a) :-
        T = 1,
        O1 = I*A1,
        O2 = I*A2 - R.
```

Figure 16: Full mortgage program for calling pattern θ_1

Unfortunately, this technique is not fully applicable to the pattern θ_2, as it specifically requires R to be unknown. However, multiple specialization can still be used to exploit future redundancy, while the inequality on P will still have to be tested at each stage. This is shown in Figure 17.

```
mg(P, T, I, R, B) :-
        I > 0, R > 0,
        smg(P, T, I, R, B).
mg(P, T, I, R, B) :-
        ( I <= 0 ; R <= 0 ),
        gmg(P, T, I, R, B).

smg(P, T, I, R, B) :-
        mg1(T, I, 1, 0, A1, A2),
        B = P*A1 + R*A2.

mg1(P0, R0, T, I, A1, A2, O1, O2) :-
        T > 1,
        T1 = T - 1,
        test(P0*A1 + R0*A2 > 0),
        mg1(P0, R0, T1, I, I*A1, I*A2 - 1, O1, O2).
mg1(P0, R0, T, I, A1, A2, O1, O2) :-
        T = 1,
        O1 = I*A1,
        O2 = I*A2 - 1.
```

Figure 17: Full mortgage program optimized for calling pattern θ_2

References

[1] Alain Colmerauer. An introduction to PROLOG-III. *Communications of the ACM*, 33(7):69–90, July 1990.

[2] M. J. García de la Banda and M. Hermenegildo. A practical approach to the global analysis of constraint logic programs. Technical Report UPM/DIA/CLIP/PRINCE-1/92.1.1, Facultad de Informática, Universidad Politécnica de Madrid, December 1992.

[3] M. J. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in constraint logic programs. Technical Report UPM/DIA/CLIP/PRINCE-1/92.2.1, Facultad de Informática, Universidad Politécnica de Madrid, December 1992.

[4] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems FGCS-88*, pages 693–702, Tokyo, Japan, December 1988.

[5] Veroniek Dumortier, Gerda Janssens, and Maurice Bruynooghe. Detection of free variables in the presence of numeric constraints by means of abstract interpretation. Technical Report CW 145, Derpartment of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A – B-3001 Leuven, Belgium, March 1992.

[6] ECRC. ECL^{i}PS^e users manual. 1993.

[7] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany*, pages 111–119. ACM, January 1987.

[8] 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.

[9] Niels Jørgensen. *Abstract Interpretation of Constraint Logic Programs.* PhD thesis, Roskilde University Center, Denmark, 1992.

[10] 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.

[11] Andrew D. Macdonald, Peter J. Stuckey, and Roland H. C. Yap. Redundancy of variables in CLP($\mathcal{R}$). In *Logic Programming: Proceedings of the 1993 International Symposium*, pages 75–93, Vancouver, October 1993. MIT Press.

[12] 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.

[13] 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.

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

[15] Spiro Michaylov. Repeated redundant inequalities in constraint logic programming. In *First International Conference on Constraints in Computational Logics*, Munich, September 1994. Also appears as Technical Report OSU-CISRC-6/94-TR31, Department of Computer and Information Science, The Ohio State University.

[16] Spiro Michaylov. Skeletons and techniques for the systematic development of constraint logic programs. In *6th IEEE International Conference on Tools with Artificial Intelligence*, New Orleans, November 1994. Also appears as Technical Report OSU-CISRC-6/94-TR30, Department of Computer and Information Science, The Ohio State University.

[17] Spiro Michaylov and Bill Pippin. Optimizing compilation of linear arithmetic in a class of constraint logic programs. In *International Logic Programming Symposium*, Ithaca, NY, November 1994. Also appears as Technical Report OSU-CISRC-8/94-TR44, Department of Computer and Information Science, The Ohio State University.

[18] Reinhard Skuppin. Treating ordinary differential equations in clp. Technical Report FAW-TR-93004, FAW Ulm, 1993.

[19] Leon Sterling and Marc Kirschenbaum. Applying techniques to skeletons. In J.-M. Jacquet, editor, *Constructing Logic Programs*, pages 127–140. MIT Press, 1993.

[20] 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.