Quantifier elimination in automatic loop parallelization

Armin Größlinger, Martin Griebl, Christian Lengauer

Department of Mathematics and Informatics, University of Passau, Germany

Abstract

We present an application of quantifier elimination techniques in the automatic parallelization of nested loop programs. The technical goal is to simplify affine inequalities whose coefficients may be unevaluated symbolic constants. The values of these so-called structure parameters are determined at run time and reflect the problem size. Our purpose here is to make the research community of quantifier elimination, in a tutorial style, aware of our application domain –loop parallelization– and to highlight the rôle of quantifier elimination, as opposed to alternative techniques, in this domain. Technically, we focus on the elimination method of Weispfenning.

Key words: loop parallelization, quantifier elimination

1 Introduction

Loop parallelization is a problem which occurs predominantly in high-performance computing. It is applied to highly iterative, compute-intensive matrix algorithms in the fields of linear algebra, numerical computation, image, signal and text processing, computational physics, chemistry and biology, etc. The algorithms are written as sequential programs and parallelism is a device to speed up the computation dramatically, possibly using hundreds or thousands of processors.

The *polytope model* [Len93,Fea96] is a powerful geometric model for the parallelization of nested loop programs. Since its inception, it has been extended from polytopes to polyhedra, and is often also called the *polyhedron model*.

Email address: {groessli,griebl,lengauer}@fmi.uni-passau.de (Armin Größlinger, Martin Griebl, Christian Lengauer).

```
DO i=0,n
  DO j=0,n
      C(i+j) = C(i+j) + A(i) * B(j)
```

Fig. 1. A sequential source code for the polynomial product

The parallelization takes place in three steps:

1. **Modelling:** The sequential input program, typically but not necessarily written in Fortran or C, is transformed to a model-based description.
2. **Parallelization:** The parallelization converts the source model to a target model.
3. **Code generation:** The code generation transforms the target model to an executable (parallel) program.

The following sections describe these steps in more detail and show how traditional methods of the polyhedron model solve the different mathematical problems which arise – provided they can be described with linear formulas. Then, we illustrate, in a tutorial style, how the application of the quantifier elimination method by Weispfenning [Wei88,LW93,Wei94b,Wei97] can help to overcome this proviso.

1.1 Modelling

The steps of a loop nest, iterating over an array structure, are laid out in a multi-dimensional integer coordinate system, with one dimension per loop. The set of points which result is called the *index space*. The borders of the index space are given by the bounds of the loops. Thus, the index space is described by a set of affine inequalities – a standard description of a polyhedron.

Directed edges between the points in the index space represent the dependences between the loop iterations. There is a vibrant research area concerned with the technology for generating these dependences automatically from the source program [Fea91,CG99,Ami04].

Example As an example, let us look at the core part of the simple sequential algorithm for the polynomial product (Figure 1). This code computes the coefficients of the product (in array C) from the coefficients of the two factors (in arrays A and B). Both factors have $n+1$ coefficients; n is called a *structure parameter*. Dependences arise from the accesses to elements of the array C. Several iterations of the loop nest access the same memory location, since the expression $i+j$ has the same value for different instances of i and j. Thus, all iterations (i_0, j_0) and (i_1, j_1) which satisfy the memory equation $i_1+j_1 = i_0+j_0$ are in mutual dependence.

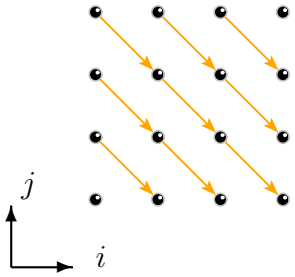


Fig. 2. Dependence structure of the polynomial product

In order to keep the representation simple, we disregard all dependences that are a transitive combination of other dependences and consider only the rest: the so-called *spanning dependences*. Starting from an iteration (i_0, j_0), we compute those iterations (i_1, j_1) that, first, precede (i_0, j_0), second, satisfy the memory equation $i_1 + j_1 = i_0 + j_0$, and third, are maximal among all solutions (i_1, j_1) w.r.t. the lexicographic order. Solving this system of contraints, we obtain $(i_1, j_1) = (i_0 - 1, j_0 + 1)$.

Figure 2 depicts the index space and the spanning dependences between loop iterations (denoted by arrows from the earlier to the later iteration accessing the same array element); the full set of dependences is the transitive closure of the dependences shown.

Limitations The polyhedron model comes with restrictions:

- The program must be a loop nest whose body consists of statements. The loop nest may be *imperfect*, i.e., not all statements need be in the body of the innermost loop.
- Statements are allowed to be array assignments, conditionals and subprogram calls. In the parallelization, the latter are treated as read and write accesses to all parameters.
- The loop bounds and array index expressions must be affine and may contain other variables which are constant in the loop nest.

1.2 Parallelization

With techniques of linear algebra and linear programming, an automatic, optimizing search yields the best mapping of the loop steps to time and space (processors) with respect to some objective function like the number of parallel execution steps (the most popular choice), the number of communications, a balanced processor load or combinations of these or others. When applied

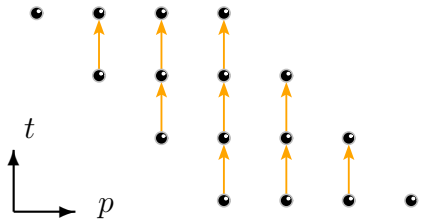


Fig. 3. Transformed index space and dependences of the polynomial product

to the source index space, this so-called *space-time mapping* specifies another polyhedral space, the so-called *target index space*.

Methods based on the polyhedron model have been implemented in various prototypical preprocessors. One for C with the message-passing library MPI is our own loop restructurer LooPo [GL96]. These systems use a number of well known *schedulers* (which compute temporal distributions) [Fea92a,Fea92b] and *allocators* (which compute spatial distributions) [Fea94,DR95] for the optimized search of a space-time mapping. In other words, for each computation of each statement in the loop nest, the scheduler determines its logical time step and the allocator its logical processor. The scheduler must ensure that the execution date increases in the direction of each dependence; this constraint can be formulated as a finite system of inequalities that the scheduler has to solve. The allocator tries to optimize locality by putting computations, which access the same memory cell, onto the same logical processor; this is done by a heuristic search.

Example For the polynomial product (Figures 1 and 2), the top-most row of iterations in Figure 2 ($j = n$) can be executed first and simultaneously. Once it is computed, all iterations of the second row can be computed, and so on. Thus, a possible time mapping is $t(i, j) = n - j$. This result is also found by solving the scheduler's constraint system. (Another solution is $t'(i, j) = i$.)

Since the allocator tries to put dependent computations on the same processor, one possible choice of time mapping $t(i, j)$ and space mapping $p(i, j)$ is:

$$\begin{aligned} t(i, j) &= n - j \\ p(i, j) &= i + j \end{aligned} \tag{1}$$

The target space and the dependences in the target space derived from these mappings are depicted in Figure 3.

Limitations The only restriction w.r.t. the parallelization phase is that the space-time mapping must be affine.

```
DO PAR p=0,2*n
  DO t=max(0,n-p),min(n,2*n-p)
      C(p) = C(p) + A(t+p-n) * B(n-t)
```

Fig. 4. A parallel target code for the polynomial product

Recent extensions allow mild violations of this affinity requirement. Essentially, they permit a constant number of breaks in the affinity (e.g., a time mapping like $t''(i,j) = \mathit{if}\ i < n-j\ \mathit{then}\ i\ \mathit{else}\ n-j$ for our running example).

1.3 Code generation

The biggest challenge is to convert the solutions found in the model into efficient code. Significant headway has been made recently on how to avoid frequent run-time tests which guide control through the various parts of the iteration space [QRW00,Bas04].

The result of the code generation is a loop nest whose loops are sequential or parallel depending on whether the corresponding axis in the target coordinate system represents time or space.

While the axes of the target coordinate system are unordered, the loops in the target loop nest must be ordered. Different orders can be chosen. If the outer loop is sequential, one obtains *synchronous* parallelism, i.e., all processes iterate following the same global clock. If the outer loop is parallel, one obtains *asynchronous* parallelism, i.e., each process iterates following its own, local clock. However, the bound expressions of every loop may only contain iteration variables of the outer loops. To enforce this in the code generation, one must use a process of variable elimination, which is in essence a quantifier elimination problem. Care must be taken that the system resulting from the variable elimination has a syntactic form suitable for code generation (see Section 2 for more detail). In simple cases, e.g., for loop nests with only one statement, Fourier-Motzkin elimination can be used, but the general case requires more advanced techniques [QRW00,Bas04].

Example From the space-time mapping, an asynchronous target program can be synthesized, which is shown in Figure 4. To generate the asynchronous target program, the target index space (Figure 3) described by the inequalities:

$$0 \le t \le n$$
$$n - t \le p \le 2 \cdot n - t$$

has to be expressed differently. To generate code, i.e., for-loops which enumerate every integral point in the target index space, where p is the variable of

the outer loop and t the variable of the inner loop, we have to express the target space as follows:

$$\begin{aligned} 0 \leq p \leq 2 \cdot n \\ 0 \leq t \leq n \\ n - p \leq t \leq 2 \cdot n - p \end{aligned} \tag{2}$$

to derive the bounds of the for-loops from the system (compare system (2) to the loop bounds in Figure 4).

1.4 An extension: granularity control

Methods based on the polyhedron model are elegant and work well. However, as presented so far, the resulting parallelism is usually too fine-grained to be efficient. An extension to the model-based parallelization phase, called *tiling*, solves this granularity problem [ABRY01,GFL04]. It partitions the index space by covering it with equally shaped and sized polyhedra, mostly parallelepipeds. Every coordinate of a point in the index space is divided into two components: the coordinate of the tile, and the coordinate of the local offset within the tile.

There is one limitation of the descriptive power of the current polyhedron model, which is hindering the applicability of the tiling technique. If the target program is not to be compiled for a fixed number of processors (which also fixes the size of the tiles), but for an unknown, i.e., parametric number of processors, the size of the tiles depends on parameters. Such a tiling can only be described by an inequality system in which the respective parameters appear as coefficients of variables. Unfortunately, the tools available for solving the linear algebra and linear optimization problems do not handle this case; they require coefficients which are numeric constants.

We have developed techniques for handling parametric coefficients in the polyhedron model. Our application is parametric tiling but, in principle, these techniques can also be used to generalize algorithms in other areas of the polyhedron model, e.g., the dependence analysis.

Example The target code for the polynomial product shown in Figure 4 has two drawbacks: it requires $2 \cdot n + 1$ processors and the work is unevenly distributed between them. With tiling, we can solve both problems at the same time, i.e., balance the work between the processors by combining smaller into bigger chunks of work, and map several of the $2 \cdot n + 1$ "virtual" processors onto one physical processor.

```
DO PAR r=0,NP-1
  DO t=0,n
     DO s=ceiling(-t/(NP*B) -r/NP +(n-B+1)/(NP*B)),
          floor(-t/(NP*B) -r/NP +(2*n)/(NP*B))
        DO o=max(0,-t-r*B-s*NP*B+n),min(B-1,-t-r*B-s*NP*B+2*n)
           DO p=max(n-t,r*B+s*NP*B+o),min(2*n-t,r*B+s*NP*B+o)
              C(p) = C(p) + A(t+p-n) * B(n-t)
```

Fig. 5. Target code for the polynomial product after tiling

In order to achieve the desired, so-called *block-cyclic* tiling, we express the virtual processor coordinate p by a coordinate triple (r, b, o); Figure 6 depicts the situation for two real processors. A number of adjacent index points are grouped together into a so-called block (in Figure 6, the number is 2). The block's number is the coordinate b. Within each block, an index point has a unique offset, denoted by o. The blocks are distributed across the real processors in cyclic, i.e., alternating fashion. The assigned number of the real processor is coordinate r.

A block-cyclic tiling of a target space can always be imposed by adding the following set of inequalities to the description of the target space:

$$\begin{aligned} 0 &\leq r \leq \mathit{NP} - 1 \\ 0 &\leq o \leq B - 1 \\ p &= b \cdot B + o \\ b &= s \cdot \mathit{NP} + r \end{aligned} \tag{3}$$

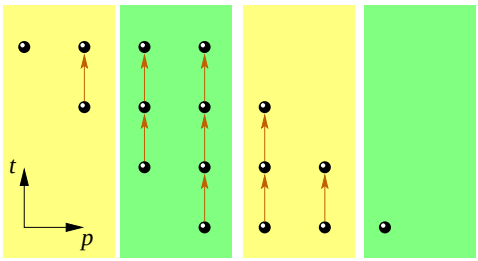
where NP is the number of real processors available and B the block size of the block-cyclic tiling. The last equation (with variable s) specifies the cyclic distribution of blocks across real processors, where s ranges over the cycles.

In order to generate a loop nest, we subject this (non-linear) system to a version of Fourier-Motzkin elimination that can deal with non-linear parameters (cf. applications in Section 2.2.1). The resulting loop code is given in Figure 5.

As desired, this program ensures that a fixed number of processors is used (in Figure 6: $\mathit{NP} = 2$, counted by r and indicated by the shadings), and that the load is balanced (in Figure 6: each of the two physical processors performs two computations per time step).

2 Quantifier elimination in the polyhedron model

Quantifier elimination is used in several places in the polyhedron model. Let us first look at its application in the current polyhedron model, and at the



p:	*0*	*1*	*2*	*3*	*4*	*5*	*6*
r:	*0*	*0*	*1*	*1*	*0*	*0*	*1*
o:	*0*	*1*	*0*	*1*	*0*	*1*	*0*
b:	*0*	*0*	*1*	*1*	*2*	*2*	*3*
s:	*0*	*0*	*0*	*0*	*1*	*1*	*1*

Fig. 6. Transformed index space after a block-cyclic tiling

algorithms used (Section 2.1). Then, let us look at our generalized model, which allows non-linear parameters, and how the need for non-linear quantifier elimination arises.

We can distinguish the following applications on linear inequality systems:

- testing the feasibility,
- projecting,
- computing solutions.

Feasibility testing Testing the feasibility of an inequality system means deciding the formula $\varphi = \exists x_1 \ldots \exists x_n \, (\psi)$, where ψ describes the inequality system, i.e., is a conjunction of inequalities. Sometimes, it suffices to decide whether $\mathbb{R} \vDash \varphi$ holds but, since the variables we analyze in the polyhedron model are integer-valued loop indices, knowing that a solution exists in the reals is often not sufficient. For example, dependence tests can output false dependences when they do not include a check that all the loop indices are integral.

Projecting Projections of inequality systems are mainly needed during code generation. To generate code which enumerates every point in a polyhedron described by ψ, one performs quantifier elimination on the following formulas $\varphi_1, \ldots, \varphi_{n-1}$:

$$\begin{aligned} \varphi_1 &= \exists x_2 \exists x_3 \cdots \exists x_n \, (\psi) \\ \varphi_2 &= \exists x_3 \cdots \exists x_n \, (\psi) \\ &\vdots \\ \varphi_{n-1} &= \exists x_n \, (\psi) \end{aligned}$$

This yields the quantifier-free equivalents $\varphi'_1, \ldots, \varphi'_{n-1}$. Here, φ'_1 is equivalent to a conjunction of atomic formulas describing the lower and upper bounds for the variable x_1, φ'_2 describes the lower and upper bounds for x_2 in dependence of x_1 (the conjuncts only involving x_1 are ignored), and so on. Finally, in the formula ψ, the lower and upper bounds for x_n depend on $x_1, \ldots, x_{n-1}$. The problem with using a quantifier elimination tool to compute $\varphi'_1, \ldots, \varphi'_{n-1}$ is that such tools usually do not guarantee that the results are conjunctions of atomic formulas, i.e., a more or less difficult post-processing effort (e.g., formula simplification) or the reliance on implementation details become necessary.

Computing solutions Projecting an inequality system onto some of its dimensions can be viewed as computing the complete solution set of the inequality system. Sometimes, however, it is sufficient to find a single solution of a system, maybe under the constraint that the solution has some properties. For example, in dependence analysis, one can, instead of dealing with all dependences of some equivalence class, choose the lexicographic maximum as a representative [Fea91].

2.1 The current polyhedron model

The current polyhedron model is restricted to inequality systems which can be described by formulas which are linear in the variables *and* in the structure parameters. The reason for this restriction is the fact that subproblems encountered in many algorithms are again linear in the variables and in the structure parameters, so this ensures a kind of "closure" property.

Feasibility testing Various algorithms and implementations thereof exist for testing the feasibility of an inequality in the reals. Most basically, Fourier-Motzkin elimination can decide the existence of a real solution. Because of its doubly exponential complexity, Fourier-Motzkin is usually replaced by other, simply exponential algorithms – most prominently, the Simplex algorithm, Chernikova's algorithm [Che68], or Weispfenning's algorithm for linear inequality systems [Wei94a]. Deciding the existence of integral solutions is algorithmically harder and, therefore, fewer implementations exist. For conjunctions of inequalities, PIP (Parametric Integer Programming) can be used [FCB03]. For more general situations, the Omega test is available [PW92,Pug92], and there are computer algebra systems which handle integral quantifier elimination on linear formulas, e.g., the REDLOG package [DS97] in the computer algebra system REDUCE [Hea04].

Projecting To compute the projection of an inequality system (the formulas $\varphi'_1, \ldots, \varphi'_{n-1}$), one does not use a full quantifier elimination tool, since there are algorithms which have been specifically designed to do so. For example, it is at the heart of Fourier-Motzkin to compute these formulas, or Chernikova's algorithm can be used to compute the projections.

Example For the polynomial product, the parallelization step (using the space-time mapping given in Equation (1)) yields the target index space depicted in Figure 3, which is described by the following inequality system:

$$\begin{aligned} 0 \leq t \leq n \\ n - t \leq p \leq 2 \cdot n - t \end{aligned} \tag{4}$$

To obtain the (untiled) target code, we have to compute the inequalities shown in system (2). As described in Section 2, we can get the bounds for the inner loop index t directly from this inequality system (by solving each inequality for t). The bounds for p can be obtained by computing a quantifier-free equivalent of the following predicate:

$$\exists t \, (0 \leq t \leq n \wedge n - t \leq p \leq 2 \cdot n - t)$$

Using Fourier-Motzkin elimination, we obtain the result $0 \leq p \leq 2 \cdot n$ directly, and it is guaranteed that any result returned is a conjunction of atomic formulas. A full quantifier elimination procedure does not give such a guarantee. For example, REDLOG yields:

$$0 \leq p \leq n \quad \vee \quad n \leq p \leq 2 \cdot n$$

Of course, this is equivalent to the result computed by Fourier-Motzkin, but it requires some suitable simplification procedure to be applied before code, i.e., a loop nest can be generated. For example, the formula simplifier SLFQ [Bro02], which is based on QEPCAD, simplifies REDLOG's result to $0 \leq p \leq 2 \cdot n$. Another way to find an equivalent conjunction is to check for every atomic formula τ in the (negation free) result φ whether $\mathbb{R} \vDash \varphi \rightarrow \tau$ holds, using quantifier elimination. A conjunction equivalent to φ is then given by the conjunction of all atomic formulas τ for which $\mathbb{R} \vDash \varphi \rightarrow \tau$ holds.

Computing solutions To compute a single solution of an inequality system, we usually use the Simplex algorithm with a suitable target function; usually, the lexicographic minimum (as computed by PIP, for example) is the desired solution. Quantifier elimination with answer (which computes solutions for existentially quantifier variables) can also be used, but our experiments showed that, in the linear case, the specialized tools (like PIP) are

faster, sometimes much faster. See Section 2.2.2 for an example application of quantifier elimination with answer.

2.2 The polyhedron model with non-linear parameters

As our example demonstrates, some desired transformations, e.g., tiling the target index space with a parametric tile size, cannot be described in the current polyhedron model, since they require parametric coefficients. This case is not handled by the algorithms mentioned in the last section. We have developed a generalization of the model to allow parameters in the coefficients of variables. The significant difference to the current model is that the most basic operation on equations and inequalities, namely solving for a variable, can, in general, not be performed uniformly for all parameter values, but a case distinction on the sign of the coefficient must be made, which can depend on the parameters. Therefore, algorithms which return a single answer in the current model (e.g., Fourier-Motzkin elimination) need to return a case distinction (with conditions on the parameters) and a separate solution for every case as result.

There are two ways of obtaining algorithms for the generalized model. First, we can look for a way to transform an existing algorithm into an algorithm for the generalized model; this is addressed in Section 2.2.1. Second, we can try to use a full quantifier elimination tool (which is not limited to formulas, which are linear in the variables and parameters, as are the tools mentioned in Section 2.1) to solve some problems; see Section 2.2.2. We review both approaches and refer the reader to our previous publications for the details [GGL04,Grö03].

Since the theory of integers with addition and multiplication is undecidable, we cannot, in general, apply the generalization technique presented in Section 2.2.1 to obtain algorithms for integral problems. Constructing such algorithms, which work for the cases usually encountered in the polyhedron model, is a subject of current research.

2.2.1 Generalizing algorithms by program transformation

The current polyhedron model uses a rich set of algorithms. Since some of them have been developed with the polyhedron model in mind, it is desireable to be able to transform these algorithms into algorithms which can handle non-linear parameters. In this section, we present our way to obtain a generalized algorithm by a transformation.

The algorithms used in the polyhedron model work on the coefficients of the

inequalities; in the current model, they work on rational or integral numbers. As a very simple example of an algorithm which is used to manipulate the coefficients, let us consider the following function abs which computes the absolute value of its argument:

$$\begin{aligned} &\text{abs} : \mathbb{Q} \to \mathbb{Q} \\ &\text{abs}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \text{abs}(-x) & \text{if } x < 0 \end{cases} \end{aligned}$$

We use recursion on purpose, since it turns out to be the problematic issue in the transformation.

For every argument x of abs, the conditions $x \geq 0$ and $x < 0$ can be evaluated and it can be determined whether to continue the computation with returning x or with computing $\text{abs}(-x)$. In the generalized polyhedron model, the coefficients of the variables are polynomials or rational functions in the structure parameters. Therefore, in the generalized model, this algorithm has to deal with arguments $f \in \mathbb{Q}[p_1, \ldots, p_m]$. But now, $f \geq 0$ and $f < 0$ cannot be evaluated, since their truth value depends on the parameters $p_1, \ldots, p_m$, in general. The generalized version abs^g of abs, which we are about to construct, has to return a data structure which describes the value of $\text{abs}(f)$ in dependence of the structure parameters. It seems that abs^g can be obtained very simply. Just replace the case distinction of $x \geq 0$ vs. $x < 0$ in the algorithm's *code* by a case distinction in the (generalized) algorithm's *result data structure*.

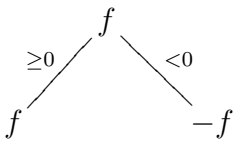
To represent case distinctions which return results of type α as a data structure, we use decision trees, denoted by the data type Tree α. This data type has (among others) the following constructors:

- $\text{Leaf} : \alpha \to \text{Tree } \alpha$ to represent a result of type α as a decision tree.
- $\text{GeCond} : \mathbb{Q}[p_1, \ldots, p_m] \times \text{Tree } \alpha \times \text{Tree } \alpha \to \text{Tree } \alpha$ to represent a binary case distinction. GeCond (g, t_1, t_2) means that, if $g \geq 0$ (for some given values of the parameters), then t_1 represents the result, otherwise t_2.

For example, we would like the absolute value of the polynomial f to be represented by the following expression:

$$\text{GeCond } (f, \text{Leaf } f, \text{Leaf } (-f))$$

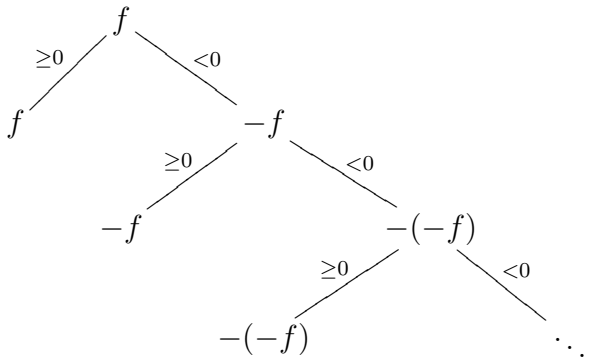
In our depiction of the decision tree, we label each edge with the condition which guards the subtree suspended from it (here, the subtrees are leaves):



Unfortunately, we must do some work to obtain this result. By transferring the case distinction from the code to the result data structure, we obtain the following implementation of abs^g:

$$\begin{aligned}&\text{abs}^g : \mathbb{Q}[p_1, \ldots, p_m] \to \text{Tree } \mathbb{Q}[p_1, \ldots, p_m]\\&\text{abs}^g(f) = \text{GeCond } (f, \text{Leaf } f, \text{abs}^g(-f))\end{aligned}$$

Alas, in the evaluation of $\text{abs}^g(f)$, the recursive call $\text{abs}^g(-f)$ is evaluated repeatedly ad infinitum, which yields the following infinite tree:



We observe that, when $f < 0$ holds, $-f < 0$ cannot hold, but $-f \geq 0$ must hold. I.e., the right child of the root node could be replaced by the leaf $-f$, but our transformed algorithm does not do so!

This example shows that a naïve transformation of algorithms to handle non-linear parameters is not sufficient, since recursive functions can produce infinite decision trees. Only after simplification (i.e., elimination of branches with contradictory conditions), the resulting decision tree may be finite. On the other hand, our transformational approach ensures that the results computed by the generalized algorithm are correct by construction, provided that the algorithm subject to the transformation is correct. Two questions must be answered:

(1) How can the simplification of the decision trees be performed?
(2) Does the simplification *guarantee* that only finite decision trees are constructed?

The answer to Question 1 is simple. Since the coefficients of the variables in the generalized polyhedron model are polynomials or rational functions in the parameters, and the algorithms manipulate the coefficients using addition, subtraction, multiplication, and division as algebraic operations and use the natural ordering relation, all the conditions in the decision trees can be written as $f \rho\, 0$ where $f \in \mathbb{Q}[p_1, \ldots, p_m]$ and $\rho \in \{=, \neq, <, \leq, \geq, >\}$. Thus, real quantifier elimination can be used to decide the feasibility of the conjunction of the conditions guarding a branch (in the reals). We use REDLOG [DS97] (a quantifier elimination package for the computer algebra system REDUCE [Hea04]),

which implements Weispfenning's elimination-by-virtual-substitution method [Wei88,LW93,Wei97], and QEPCAD [Bro04], which implements quantifier elimination through partial cylindrical algebraic decomposition. By performing a top-down search for contradictions this way, we avoid running into the infinite branch in the abs^g example, since the infinite branch is cut off when $f < 0 \wedge -f < 0$ is found to be infeasible. Using real quantifier elimination, although the structure parameters are from $\mathbb{Z}$, turns out not to be a problem, since the structure of the problems (like tiling) implies that quantifier elimination in the reals is sufficient. For example, we have proved that tiling the index space of one statement with congruent parallelepipeds as tiles yields a target code which requires no case distinctions in the parameters at all [GGL04].

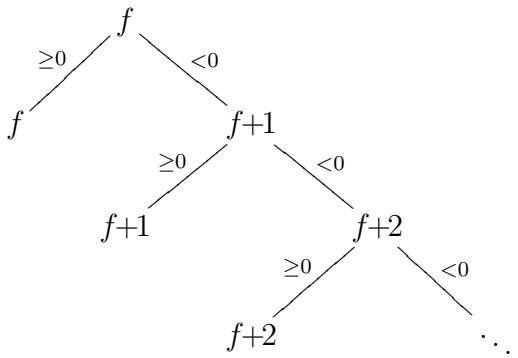
Question 2 is more difficult to answer. There are algorithms which produce infinite decision trees even with the best possible top-down tree simplification. For example, the following function:

$$\begin{aligned}&\text{pos} : \mathbb{Q} \to \mathbb{Q}\\ &\text{pos}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \text{pos}(x+1) & \text{if } x < 0 \end{cases}\end{aligned}$$

is generalized to:

$$\begin{aligned}&\text{pos}^g : \mathbb{Q}[p_1, \ldots, p_m] \to \text{Tree } \mathbb{Q}[p_1, \ldots, p_m]\\ &\text{pos}^g(f) = \text{GeCond } (f, \text{Leaf } f, \text{pos}^g(f+1))\end{aligned}$$

which produces (for some input f) the following decision tree:



But every finite prefix of the infinite branch is consistent. So algorithm pos cannot be generalized by our transformation – even with simplification.

Termination (i.e., the construction of a finite decision tree) is guaranteed if every infinite branch has a finite prefix which is inconsistent. This condition ensures that searching top-down for contradictions and cutting off branches with infeasible conditions eliminates the infinite branch. Note that it does not suffice that, for any given values of the parameters, only a finite prefix of the tree is used. In pos^g, this is the case – yet the tree cannot be made finite.

Applications We have used this transformation to generalize Fourier-Motzkin elimination and the Simplex algorithm to non-linear parameters. For both algorithms, one can prove that top-down simplification ensures that only finite trees are generated. This gives us the ability to solve each of our three main applications (feasibility testing, projection, computing a solution).

Proving the termination of Fourier-Motzkin is rather simple. Since the number of case distinctions is bounded by the number of variables and inequalities in the input system, Fourier-Motzkin produces finite trees even without simplification. But simplification is still necessary to reduce the –in principle– exponential size of the decision trees to trees with few or no case distinctions in practical applications. E.g., for the tiling example, it turns out that no case distinction is ever needed.

The trees produced by the generalized Simplex are infinite, because they must also describe cyclic computations. But we can prove that the use of Bland's minimal index rule for choosing pivots [Bla77] ensures that top-down tree simplification cuts off every infinite branch. Assume that there is an infinite branch labelled with the conditions $(\varphi_i)_{i\in\mathbb{N}}$ such that no prefix $(\varphi_i)_{0\leq i\leq k}$ ($k \in \mathbb{N}$) is inconsistent. Since Simplex together with Bland's rule specifies only a finite number of case distinctions for every pivoting step and the number of pivoting steps performed in an acyclic computation is bounded by the number of variables and inequalities in the input system, the branch must describe a cyclic Simplex computation. But this is a contradiction, since the pivots are chosen using Bland's rule and, therefore, no consistent prefix of the branch can contain a cycle. Thus, no infinite branch with only consistent finite prefixes can exist.

Let us summarize our transformational approach to obtaining algorithms for the generalized polyhedron model:

- Start with an existing algorithm for the current polyhedron model.
- Replace every case distinction in the algorithm's code by a case distinction in the result data structure (by replacing the code for the case distinction by an application of a suitable decision tree constructor like GeCond).
- Change the result type from α to Tree α in the type signature.

To make the principle clearer, we have omitted some technical details [Grö03].

Example An application of our generalized Fourier-Motzkin implementation (which uses our generalized Simplex algorithm to eliminate redundant bounds) to the conjunction of systems (3) and (4) yields a system of projections which correspond directly to the loop bounds of the code in Figure 5. No case distinctions are needed to represent the result.

2.2.2 Using quantifier elimination directly to solve some problems

Some problems in the generalized polyhedron model can be formulated as quantifier elimination problems. For example, testing the feasibility of an inequality system ψ is simply asking the quantifier elimination tool for a quantifier-free equivalent of $\exists x_1 \ldots \exists x_n (\psi)$.

Computing the lexicographic minimum of an inequality system is an example in which quantifier elimination *with answer* can be used. Given an inequality system ψ, the following formula:

$$\mu := \psi \wedge \forall y_1 \ldots \forall y_n \left(\psi[x_1 := y_1, \ldots, x_n := y_n] \rightarrow (x_1, \ldots, x_n) \leq_{lex} (y_1, \ldots, y_n) \right)$$

describes that $(x_1, \ldots, x_n)$ is the finite lexicographic minimum of ψ's solution set. Quantifier elimination with answer can now compute a quantifier-free formula (in the parameters) of the form $\bigvee \chi_i$ which is equivalent to $\exists x_1 \ldots \exists x_n (\mu)$ and, for the different cases χ_i, values for $(x_1, \ldots, x_n)$ are computed such that μ holds, i.e., the lexicographic minimum is computed.

In order to compare this approach with the generalization of existing algorithms (Section 2.2.1), we used a generalized Simplex algorithm for computing the lexicographic minimum. It turns out that the generalized Simplex is by orders of magnitude more efficient than the direct encoding described in this section. The reason is that the Simplex algorithm is tuned to its application domain, whereas quantifier elimination cannot make use of domain-specific information.

A second drawback of using quantifier elimination directly is that not all problems in the polyhedron model can be described easily this way. For example, it is not guaranteed that the projection of a polyhedron with non-linear parameters can be described by a conjunction. Consider the following formula:

$$0 \leq x \leq y \quad \wedge \quad p \cdot y \geq 0 \quad \wedge \quad y \leq 10$$

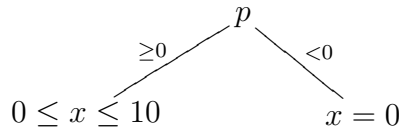
with non-linear parameter p. If we assume nothing about p, REDLOG's answer to the question $\exists y \, (0 \leq x \leq y \wedge p \cdot y \geq 0 \wedge y \leq 10)$ is:

$$(x = 0 \wedge p \neq 0) \quad \vee \quad (0 \leq x \leq 10 \wedge p \geq 0) \tag{5}$$

This formula cannot be written as a conjunction of atomic formulas. To generate code, the conditions on the loop indices must be joined by conjunctions. The given problem can be solved by applying our generalized Fourier-Motzkin to:

$$0 \leq x \leq y \quad \wedge \quad p \cdot y \geq 0 \quad \wedge \quad y \leq 10$$

to eliminate y:

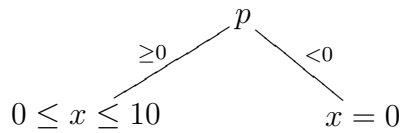


The question now is: can we find the conditions $p \geq 0$ and $p < 0$ and the respective solutions $0 \leq x \leq 10$ and $x = 0$ from the quantifier elimination's answer to $\exists y\,(0 \leq x \leq y \wedge p \cdot y \geq 0 \wedge y \leq 10)$?

In this example, one can find a suitable case distinction by extracting the two conditions $p \neq 0$ and $p \geq 0$ from the two disjuncts of (5). This yields a total of four possible cases:

1.	$p \neq 0 \wedge p \geq 0$:	$x = 0 \vee x \geq 0$
2.	$p \neq 0 \wedge \neg(p \geq 0)$:	$x = 0$
3.	$\neg(p \neq 0) \wedge p \geq 0$:	$x \geq 0$
4.	$\neg(p \neq 0) \wedge \neg(p \geq 0)$:	$x \in \emptyset$

After simplification (Case 4 is impossible and Cases 1 and 3 can be combined since both have $x \geq 0$ as solution) and computing conjunctive equivalents for the formulas describing x in each case, we get, as desired, the following decision tree:



Example Instead of using our generalized Fourier-Motzkin, one can also use REDLOG and SLFQ to compute the tiled target code (Figure 5), because we have proved that the projections are equivalent to conjunctions of atomic formulas [GGL04]. As described in Section 2, for every loop index, REDLOG is used to project the input system onto the respective prefix of the loop indices. The resulting formulas are fed to SLFQ to simplify them to a conjunction of atomic formulas from which the loop bounds can be derived. Unfortunately, 2GB of heap are not enough for SLFQ to simplify the formula describing the innermost loop with the bounds for p. We can also compute the required bounds using the simpler technique of checking, for every atomic formula in REDLOG's result, whether it is implied by the whole formula. In our example, this works for all loop indices, including p.

3 Conclusion

Automatic, model-based parallelization requires powerful mathematical tools, e.g., for optimization, projection or feasibility tests. The traditional limitation in this research area is the linearity of the expression, but this is too restrictive even for problems which are quite simple but relevant in practice.

The basis for the solution presented here is a powerful and, at the same time, efficiently implemented mathematical tool. Quantifier elimination, e.g., the method by Weispfenning as implemented in the REDLOG package of REDUCE, provides us with the necessary functionality. We discovered that generalizing an existing algorithm (like the Simplex algorithm or Fourier-Motzkin elimination) using quantifier elimination to simplify the resulting decision trees usually performs better than expressing the problem as a logical formula and using quantifier elimination alone to solve it. Combining quantifier elimination with existing algorithms enables a much desired feature, namely non-linear parameters. This way, quantifier elimination is making a significant contribution to bringing parallelization in the polyhedron model closer to the realm of practice.

Acknowledgements

This work has been funded in part by exchange grants from the German Academic Exchange Service (DAAD) in its Procope programme, and from the Bavarian-French University Center (BFHZ-CCUFB). Our exchange partners were Paul Feautrier and his group; we thank them for discussions. We also thank the reviewers for their useful comments.

References

[ABRY01] Rumen Andonov, Stefan Balev, Sanjay Rajopadhye, and Nicola Yanev. Optimal semi-oblique tiling. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2001)*, pages 153–162. ACM Press, July 2001. Extended version available as technical report: IRISA, No. 1392, December 2001.

[Ami04] Pierre Amiranoff. *An Automata-Theoretic Modelization of Instancewise Program Analysis: Transducers as Mappings from Instances to Memory Locations.* PhD thesis, Conservatoire National des Arts et Métiers, December 2004.

[Bas04] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proc. 13th Int. Conf. on Parallel Architectures and Compilation Techniques (PACT 2004)*, pages 7–16. IEEE Computer Society Press, September 2004.

[Bla77] Robert G. Bland. New finite pivoting rules for the simplex method. *Mathematics of Operations Research*, 2:103–107, 1977.

[Bro02] Christopher W. Brown. The SLFQ system, 2002. `http://www.cs.usna.edu/~qepcad/SLFQ/Home.html`.

[Bro04] Christopher W. Brown. QEPCAD quantifier elimination tool, 2004. `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`.

[CG99] Jean-François Collard and Martin Griebl. A precise fixpoint reaching definition analysis for arrays. In Larry Carter and Jeanne Ferrante, editors, *Languages and Compilers for Parallel Computing, 12th International Workshop, LCPC'99*, LNCS 1863, pages 286–302. Springer-Verlag, 1999.

[Che68] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6):282–293, 1968.

[DR95] Michèle Dion and Yves Robert. Mapping affine loop nests: New results. In Bob Hertzberger and Giuseppe Serazzi, editors, *High-Performance Computing & Networking (HPCN'95)*, LNCS 919, pages 184–189. Springer-Verlag, 1995.

[DS97] Andreas Dolzmann and Thomas Sturm. REDLOG: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31:2–9, June 1997.

[FCB03] Paul Feautrier, Jean-François Collard, and Cédric Bastoul. Solving systems of affine (in)equalities: PIP's user's guide. Technical report, PRiSM, University of Versailles, 2003.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, 1992.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.

[GFL04] Martin Griebl, Peter Faber, and Christian Lengauer. Space-time mapping and tiling – a helpful combination. *Concurrency and Computation: Practice and Experience*, 16(2–3):221–246, February 2004. Proc. 9th Workshop on Compilers for Parallel Computers (CPC 2001).

[GGL04] Armin Größlinger, Martin Griebl, and Christian Lengauer. Introducing non-linear parameters to the polyhedron model. In Michael Gerndt and Edmond Kereku, editors, *Proc. 11th Workshop on Compilers for Parallel Computers (CPC 2004)*, Research Report Series, pages 1–12. LRR-TUM, Technische Universität München, July 2004.

[GL96] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo. In Michael Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers (CPC'96)*, Konferenzen des Forschungszentrums Jülich 21, pages 311–320. Forschungszentrum Jülich, 1996.

[Grö03] Armin Größlinger. Extending the Polyhedron Model to Inequality Systems with Non-linear Parameters using Quantifier Elimination. Diploma thesis, Universität Passau, September 2003. `http://www.infosun.fmi.uni-passau.de/cl/arbeiten/groesslinger.ps.gz`.

[Hea04] Athony C. Hearn. REDUCE computer algebra system, 2004. `http://www.reduce-algebra.com`.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[LW93] Rüdiger Loos and Volker Weispfenning. Applying linear quantifier elimination. *The Computer Journal*, 36(5):450–462, 1993. Special issue on computational quantifier elimination.

[Pug92] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Comm. ACM*, 35(8):102–114, August 1992.

[PW92] William Pugh and Dave Wonnacott. Eliminating false data dependences using the Omega test. *ACM SIGPLAN Notices*, 27(7):140–151, July 1992. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'92).*

[QRW00] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Programming*, 28(5):469–498, October 2000.

[Wei88] Volker Weispfenning. The complexity of linear problems in fields. *J. Symbolic Computation*, 5(1&2):3–27, February–April 1988.

[Wei94a] Volker Weispfenning. Parametric linear and quadratic optimization by elimination. Technical Report MIP-9404, Universität Passau, April 1994.

[Wei94b] Volker Weispfenning. Quantifier elimination for real algebra—the cubic case. In *Proc. Int. Symp. on Symbolic and Algebraic Computation (ISSAC'94)*, pages 258–263. ACM Press, July 1994.

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra—the quadratic case and beyond. *Applicable Algebra in Engineering Communication and Computing*, 8(2):85–101, February 1997.