UNIVERSITÄT PASSAU
Fakultät für Informatik

Code Generation in the Polytope Model with Non-linear Parameters

Bachelorarbeit

Autor:
Philipp Claßen

Betreuer:
Priv. Doz. Dr. Martin Griebl
Lehrstuhl für Programmierung
Universität Passau

Passau, April 17, 2007

Acknowledgements

First of all, I would like to thank Martin Griebl for his many valueable remarks and suggestions, and for proofreading this work.

I also would like to thank Armin Größlinger for explaining the world of quantifier elimination and for his great technical support, especially with the logic tools. Without him, this work would not have been possible.

My final thanks goes to my brother Micki.

Contents

Abstract. The polytope model is a framework for the automatic parallelization of loop nests. Since the model is based on linear programming, all coefficients of the polytopes describing the loops, must be rational. Recently, it has been shown that this restriction can be lifted using quantifier elimination in the real numbers.

In this work, we will demonstrate that the extended polytope model can be applied in small practical examples. We will generalize Quilleré's algorithm [Qui00] for target code generation, thus enabling the use of non-linear parameters.

1. Introduction

As manual parallelization is expensive and error-prone, it seems preferable to have compilers that can automatically transform sequential programs into parallel programs. One way to achieve this is to use the polytope model [Len93] which is very powerful – if it is applicable. That is why much of the research concentrates on extensions to the original model.

One of these restrictions is that the matrices describing the polytopes contain only rational numbers. Thus, these matrices can only describe inequalities that are linear in their variables and parameters. By parameters we mean symbolic constants, which are first known at runtime, but cannot be changed thereafter. Suppose i and j are variables and n is a parameter. The inequality $3i + 4j + 7n + 8 \geq 0$ is (affine) linear and can be described by a rational matrix M:

$$\underbrace{\begin{pmatrix} 3 & 4 & 7 & 8 \end{pmatrix}}_{M} \cdot \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq 0$$

In contrast, the inequality $n \cdot i \geq 0$ cannot be expressed as a rational matrix, because the coefficient of the variable i is not known at compile time. Note that this restriction is not caused by the model itself, but by most of the mathematical tools that are being used. As a consequence, the applicability of some recently developed techniques (e.g., tiling) is severely limited.

In his Diploma thesis [Grö03], Armin Größlinger showed that the polytope model can be extended to allow inequality systems with non-linear parameters, that means parameters appearing as coefficients (e.g., $n \cdot i \geq 0$) or parameters whose total degree[1] is greater or equal than two (e.g., $i + j \geq n^2$). The idea is to generalize existing algorithms by introducing case distinctions on the parameters. As the number of case distinctions quickly explodes, even though most of them are redundant, it is necessary to simplify the results. To decide whether a case distinction is redundant, we use quantifier elimination in the real numbers. Alternatively, it is possible to develop new algorithms that are

[1] The total degree of a monomial $X_1^{i_1} \cdots X_n^{i_n}$ is $i_1 + \ldots + i_n$, for instance X^2Y^3 has a total degree of 5.

directly based on quantifier elimination. This approach is preferable, when the problem can be elegantly expressed using first-order logic.

In this work, we will focus on the target code generation in the extended polytope model, specifically the Quilleré algorithm [Qui00] for loop generation.

1.1. Motivation: Example

Let us illustrate the problem with a simple example:

```
      for i=0..5
S1:      A := ...
      end

      for i=6..10
S2:      A := ...
      end
      for i=0..M-1
S3:      B[i] := B[i-1]
      end

      for i=M..100
S4:      B[i] := B[i-M]
      end
```

Note that this program contains only (affine) linear expressions as loop bounds and array indices. So each loop can be expressed by polytopes represented as matrices with constant entries. So far, everything corresponds well with the polytope model.

Taking a closer look at the example, only the last loop can be parallelized. It is possible to execute M computations simultaneously. Note that M is a parameter, which is first known at run time. As the first three loops have to be sequential and S_4 has to be scheduled after S_3, a suitable schedule would be:

$$\begin{aligned}
&t_1, t_2, t_3, t_4 :: \mathbb{Z} \to \mathbb{Z}\\
&t_1(i) = i\\
&t_2(i) = i\\
&t_3(i) = i\\
&t_4(i) = \left\lfloor \frac{i}{M} \right\rfloor + M - 1 = \left\lfloor \frac{i}{M} + \frac{(M-1)\cdot M}{M} \right\rfloor = \left\lfloor \frac{i + M^2 - M}{M} \right\rfloor
\end{aligned}$$

To get rid of the $\lfloor . \rfloor$ expression in t_4, we introduce a new dimension r which stands for the remainder of i divided by M. This leads to a multi-dimensional schedule, where the r dimension can be run in parallel.

$$\begin{aligned}
&t_1, t_2, t_3, t_4 :: \mathbb{Z} \to \mathbb{Z}^2\\
&t_1(i) = (i, 0)\\
&t_2(i) = (i, 0)\\
&t_3(i) = (i, 0)\\
&t_4(i) = (t, r), \text{ so that } i + M^2 - M = M \cdot t + r \wedge 0 \le r \le M - 1
\end{aligned}$$

Finally, we receive these target inequality systems:

$$\begin{array}{lll}
S_1: & 0 \leq t \leq 5 & \wedge \quad r = 0 \\
S_2: & 6 \leq t \leq 10 & \wedge \quad r = 0 \\
S_3: & 0 \leq t \leq M-1 & \wedge \quad r = 0 \\
S_4: & M \leq M \cdot t + r - M^2 + M \leq 100 & \wedge \quad 0 \leq r \leq M-1
\end{array}$$

In the last inequality system S_4, we obtain two non-linear expression, namely $M \cdot t$ and M^2. Thus, we cannot use our current methods to generate code. Although it is possible to avoid this problem by choosing a simpler schedule, for example $t_4(i) = i$, but that means giving away parallelism unnecessarily. Therefore this solution is not satisfactory.

So our simple example is an interesting case study to examine whether the non-linear extensions work for small practical examples and to show where the current problems are. But first, let us deal with some necessary basics, before we can concentrate on the extensions later.

2. Basics

2.1. Why Non-Linear Parameters?

Apart from our introductory example, there are other important benefits of allowing non-linear parameters, for example:

Parametric Tiling Tiling is a technique that can be used for cache optimizations, gaining parallelism or minimizing communication overhead after the space-time mapping [GFL04]. However, due to the restrictions of the model, its applicability is limited, as the size and shape of the tiles has to be known at compile time.

Otherwise it is often desirable to make the tile shape or size dependent on certain run time parameters like the number of processors, the startup costs for communications and so on. By allowing non-linear parameters, it is possible to lift some of these limitations [Grö03].

Dynamically allocated multidimensional arrays In C/C++ it is not possible to allocate a multidimensional array whose size is not known at compile time. A common practice is to allocate a one-dimensional array instead and do the index resolving manually, for example:

```
// allocate an array of size MxN ...
int *a = malloc(M * N * sizeof(int));
// ... but instead of a[i][j], we have to write
a[i*N + j] = 42;
```

Note that the array index $i * N + j$ is not a linear expression, so we have just left the scope of the polytope model.

There are some recent language extensions avoiding this problem, for example, the new language definition of C [C99] introduced *Variable Length Arrays* (VLA).

However, since most compilers do not support it at the moment, you will still find a lot of code as in the example above.

2.2. Decision Trees

Let us suppose, we are interested whether the loops from our introductory example overlap:

$$\begin{aligned} loop_1 &= \{i \mid 0 \le i \le 5\}, & loop_2 &= \{i \mid 6 \le i \le 10\}, \\ loop_3 &= \{i \mid 0 \le i \le M-1\}, & loop_4 &= \{i \mid M \le i \le 100\} \end{aligned}$$

It is easy to verify that the first two loops do not overlap, but it becomes more complicated when parameters are involved. For example, $loop_4$ overlaps with $loop_1$ if and only if $M \le 5$. That means, in general, the decision has to be postponed until M is known. However, that is not before run time.

An alternative is to build a tree containing the decisions. Every time when a decision must be made, we create a branch in the tree; for each possible case, we create one child associated with the condition that has to be met. Then we continue recursively, but for each case we can assume that its condition holds and use this information for optimization.

The advantage is that it becomes possible to make decisions at compile time, although it requires knowledge only known at run time. However, the resulting decision tree can grow very large if no simplifications are made. This can be problematic for three reasons:

1. The memory usage and computing time for intermediate results is increased *(compile-time overhead)*
2. The tree has to be evaluated when the parameters are known *(run-time overhead)*
3. The tree must not contain infinite paths, otherwise the termination is not guaranteed *(concerns correctness of the compiler)*

Let us return to the problem of determining all loops that overlap with $loop_4$. There are three possible cases:

- If $M \le 5$ then $loop_1$ and $loop_2$ overlap with $loop_4$
- If $6 \le M \le 10$ then only $loop_2$ overlaps with $loop_4$
- If $M > 10$ then no other loop overlaps with $loop_4$

This leads to the decision tree shown in Figure 1. As this tree has been constructed according to our considerations, it is already simplified.

In contrast, the decision tree shown in Figure 2 is systematically constructed. Note that five of its eight leaves cannot be reached, because of inconsistent conditions. For example, $M \le 5$ implies that $M > 10$ is always false, so a complete subtree can be cut.

In general, it can be difficult to decide whether a given condition holds. But in our context, most decision problems can be formulated as first-order formulas in the real numbers. These formulas can be decided using quantifier elimination [Grö03].

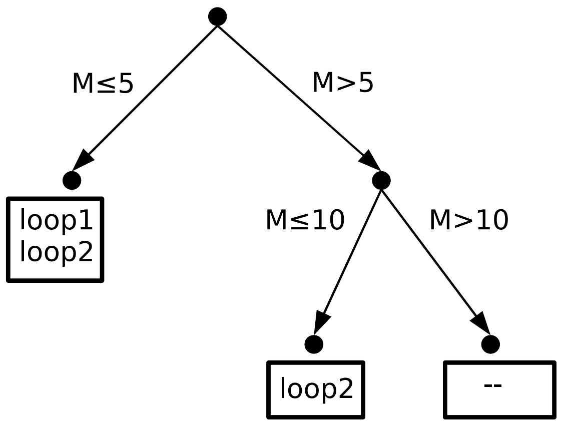


Figure 1: A decision tree showing all loops that overlap with $loop_4$

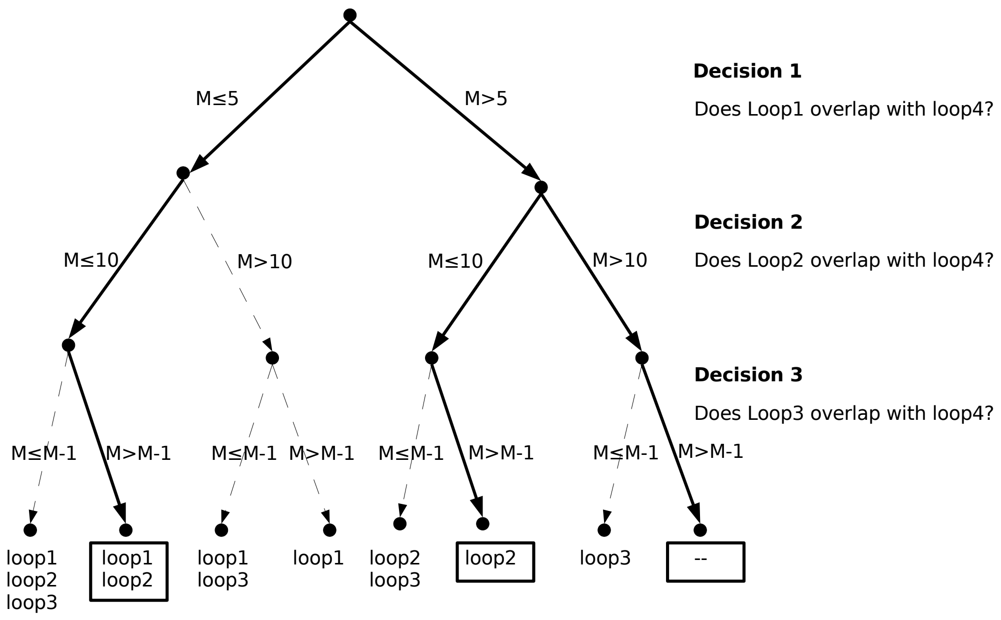


Figure 2: A systematically constructed decision tree. Subtrees that can be cut, are drawn as dashed lines.

Let us suppose we want to verify that the subtree guarded by the conditions $M \leq 5$ and $M > 10$ cannot be reached. First, we set up the corresponding formula:

$$\Psi : \exists M \; (M \leq 5 \implies M > 10)$$

Then we use quantifier elimination to get an equivalent quantifier-free formula that can be decided:[2]

$$\Phi : \bot, \text{ where } \mathbb{R} \models \Psi \iff \Phi$$

The result is that no such M exists, so it is correct to remove the subtree. Finally, when all inconsistent conditions are removed, we receive the simplified tree shown in Figure 1.

The simplification of decision trees is described in detail in the literature [Grö03, section 4.2.2].

2.3. Quilleré's Algorithm for Linear Parameters

The final step in the polytope model is always code generation, that means generating loops which enumerate all integral points inside the transformed polyhedra in the lexicographic order. This technique is called *scanning*. As, in general, we do not have a single polyhedron, but rather multiple polyhedra, it is more precisely known as the *multiple polyhedra scanning problem*. One sophisticated algorithm to solve this problem, has been proposed by Quilleré [Qui00].

The idea is to partition the union of polyhedra recursively into disjoint regions that can be scanned using imperfectly nested loops. We start with the outermost dimension – which is equivalent to the outermost loop – and proceed inwards. For each new dimension, we take the following steps:

1. Project the polyhedra onto the outermost dimensions.
2. Partition these projections into disjoint polyhedra.
3. Compute a topological order that respects the lexicographic order for the partitions from step 2.
4. Generate target code recursively which are loops that scan the sorted partitions from step 3.

Let us illustrate the algorithm with a small example. Figure 3 shows a program consisting of two statements S_1 and S_2 with the following associated polyhedra:

$$\begin{array}{ll} \{i,j \mid 0 \leq i \leq N \wedge 0 \leq j \leq 2\} & :: S_1 \\ \{i,j \mid i \leq N \qquad \wedge 0 \leq j \leq i-2\} & :: S_2 \end{array}$$

One run of the algorithm is illustrated in Figure 4: First, we project S_1 and S_2 on the outermost dimension i and separate these projections into disjoint regions. We obtain one region $L1$ containing S_1, but not S_2, and another region $L2$ that contains both S_1 and S_2. $L1$ has to be scanned before $L2$.

[2] Notation: *false* is denoted by $\bot$

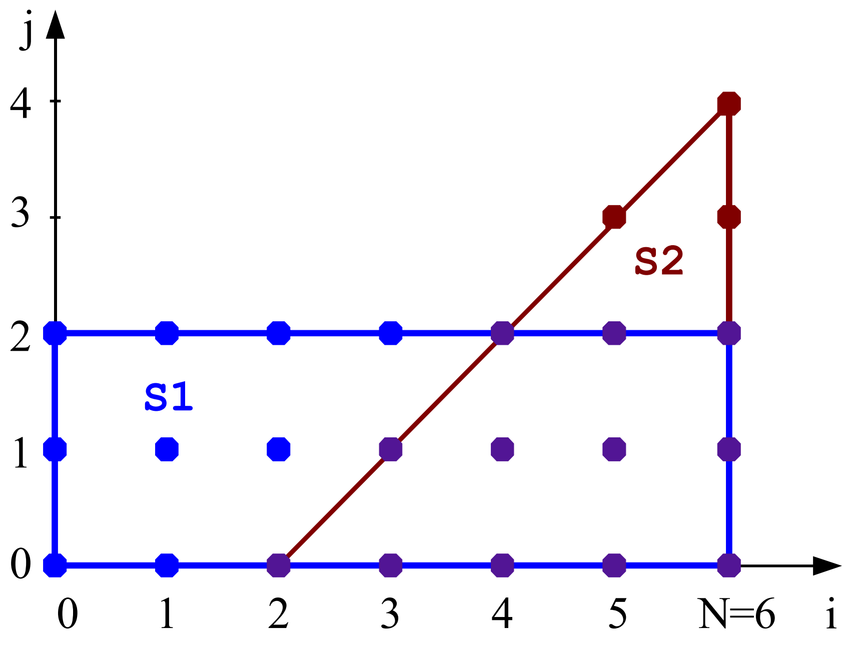


Figure 3: Quilleré example ($N = 6$)

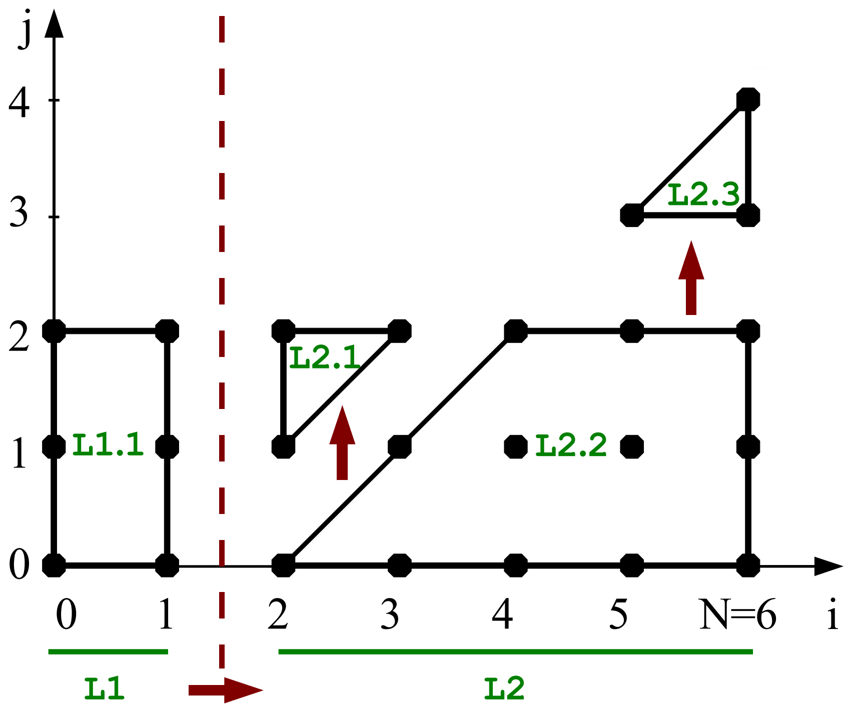


Figure 4: Partitions with dependencies

Proceeding with the next dimension, we recursively generate loops for $L1$ and $L2$. Since $L1$ contains only one statement, it is more illustrative to examine the recursive call for $L2$:

We can skip the projection to the outermost two dimensions, as we are considering the full space. The partitioning of $L2$ results in the regions $L2.1$ (S_1), $L2.2$ (S_1, S_2) and $L2.3$ (S_2). Next, we determine a textual order for these partitions, where the first dimension i is fixed by the outer loop specified via $L2$. By testing each combination, we get two constraints:

- $L2.2 \dashrightarrow L2.1$ (i.e., $L2.2$ is scanned before $L2.1$)
  Consider $i = 2$ fixed, then $(2, 0) \in L2.2$ must be scanned before $(2, 1) \in L2.1$.
- $L2.2 \dashrightarrow L2.3$
  Analogous for $i = 5$: $(5, 2) \in L2.2$ has to precede $(5, 3) \in L2.3$.
- $L2.1$ and $L2.3$ are incomparable
  This is no constraint; both $L2.1 \dashrightarrow L2.3$ and $L2.3 \dashrightarrow L2.1$ are possible.

We choose the order $L2.2 \dashrightarrow L2.1 \dashrightarrow L2.3$. Note that $L2.2 \dashrightarrow L2.3 \dashrightarrow L2.1$ would be legal, too.

Figure 5 summarizes the intermediate results so far. It contains the hierarchy of sorted partitions including their associated statements. The final step is to generate the target code which is shown in Figure 6. In our example, this is fairly straightforward.

```
L1: {i | 0 <= i <= 1, i <= N} ::
  L1.1: {i,j | 0 <= j <= 2} ::
    S1;


L2: {i | 2 <= i <= N } ::
  L2.2: {i,j | 0 <= j <= 2, j <= i-2} ::
    S1;
    S2;

  L2.1: {i,j | 1 <= j <= 2, i-1 <= j} ::
    S1;

  L2.3: {i,j | 3 <= j <= N, j <= i-2} ::
    S2;
```

$\Longrightarrow$

```
for i=0..min(1, N)
  for j=0..2
    S1;
  end
end
for i=2..N
  for j=0..min(2, i-2)
    S1;
    S2;
  end
  for j=max(1, i-1)..2
    S1;
  end
  for j=3..min(N, i-2)
    S2;
  end
end
```

Figure 5: Sorted partitions

Figure 6: Target code

A general approach is to solve the loop descriptions for the associated loop variable. The resulting lower and upper bounds define the loop bounds one-to-one. Additional constraints must be included as additional guards. The body of the loop is a composition of all its associated statements. Consider this small example:

$$L_i : \{i \mid p - i \geq 0 \wedge i - 2 \geq 0 \wedge p = 2i + 1 \wedge p \geq 3\} :: S$$

is equivalent to

$$L_i : \{i \mid 2 \leq i \leq p \wedge p = 2i + 1 \wedge p \geq 3\} :: S$$

which can be transformed into [3]

```
for i=2..p
  if p == 2*i + 1 && p >= 3 then
    S;
  end
end
```

Quilleré describes several extensions of the algorithm, for example:

- Elimination of redundant loop bounds implied by the context (i.e., by its surrounding loops)
- Elimination of guards (at the expense of increased code size)
- Support for non-unimodular mappings

3. Necessary Non-linear Extensions

In this section, we demonstrate how the basic steps of the Quilleré algorithm can be generalized to allow non-linear parameters. We start with the *projection* phase, which can be solved using quantifier elimination. After this, we continue by extending the *partitioning* step. We will see that decision trees can be often avoided and we will discuss the assets and drawbacks of this approach. Finally, we present a generalized algorithm for *topological sorting*, which turns out to be the most expensive operation.

But first, let us agree on some conventions used in this section:

- A *domain* is a finite union of polyhedra.
- Our implementation contains a module `Domain.hs`, which provides basic operations on domains with non-linear parameters (e.g., intersection, union, complement, disjoint union). These operations are needed for the extensions discussed in this section.

  Because there is a danger of confusion, we will use typewriter font ("`Domain`") when we refer to the data type in our implementation and a normal font ("domain") when we refer to a finite union of polyhedra.

[3] As the guard $p \geq 3$ does not depend on i, it is also possible to move it before the `for` loop. Note that this is not possible for the other guard $p = 2 \cdot i + 1$.

3.1. Projection

Given a polyhedron P with n dimensions, we want to project P onto the first d dimensions (where $d < n$):

$$\text{proj}_{\{1,\dots,d\}}(P) = \{(c_1, \dots, c_d) \mid \exists c_{d+1} \cdots \exists c_n \;\; ((c_1, \dots, c_n) \in P)\}$$

Even though the result is a polyhedron for every choice of its parameters, these parameters will generally introduce case distinctions. One way to deal with them, is to use decision trees. For example, by using an extended version of Fourier-Motzkin elimination [Grö03, section 4.2.1], it is possible to generate a decision tree where all leaves are single polyhedra.

The alternative is to relax the restriction that the result must be a polyhedron, but to allow domains (with possibly empty polytopes in the description) instead. Note that the above definition of the projection of a polyhedron can be directly expressed as first-order formula. After applying quantifier elimination we obtain a formula describing the result of the projection.

Since the task of quantifier elimination is to find an arbitrary quantifier-free formula that is equivalent to the input formula, it will generally not return a conjunction of inequalities. That means, the formula does not describe a single polyhedron, but a domain. No decision tree is generated, as the case distinctions are "hidden" inside the formula.

The conversion between a domain represented as a first-order formula and a system of inequalities will be discussed in section 4.1. Note that every domain can be represented as a first-order formula, but the reverse direction is not always possible.

However, we should keep clearly in mind that the final code generation accepts only polyhedra. That means, we can use domains for intermediate results as long as the input of the code generation are polyhedra. For this reason, the overview shown in Figure 7, contains an additional step, called *disjoint union*, which follows the partitioning. It will be discussed in section 3.3.

Remarks:

- Since each domain can be represented as a formula, the projection is not only legal for polyhedra, but also for domains. Therefore, the input systems of our extended Quilleré algorithm can be domains as well.

Example:

Consider S_2 from the previous Quilleré example (shown in Figure 3):

$$\{i, j \mid i \le N \wedge 0 \le j \le i - 2\} :: S_2$$

Let us project onto the i-dimension of S_2 using quantifier elimination. First, we formulate the problem using first-order logic. As we want to project onto the dimension i and

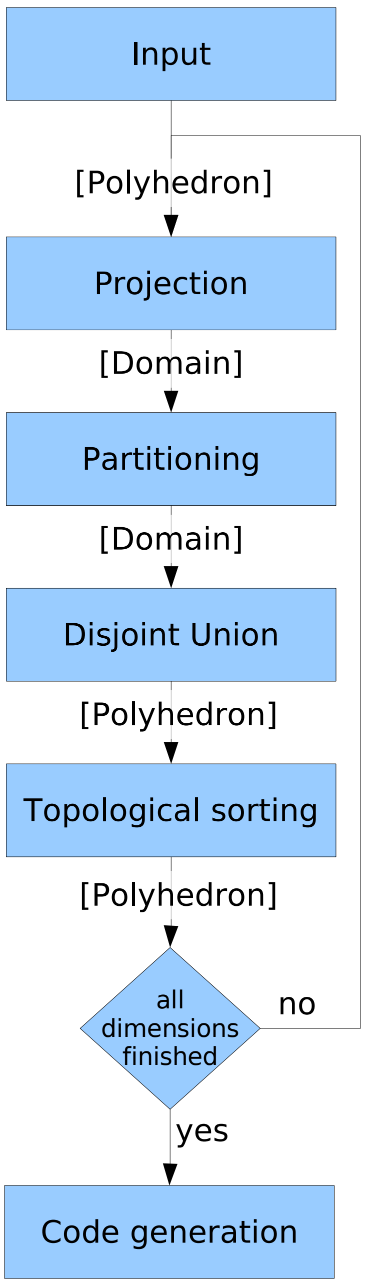


Figure 7: Illustrates the input and output of each basic step of the extended Quilleré algorithm. The square brackets denote lists (i.e., [Polyhedron] is a list of polyhedra). Only the topological sorting and the code generation will increase the decision tree.

the parameter N, we have to eliminate all *other* dimensions. That means, the existence quantifier has to bind j.

$$\exists j\ (i \le N \land 0 \le j \land j \le i - 2)$$

which is equivalent to

$$2 \le i \le N$$

3.2. Partitioning

Given a list of domains $D_1, \ldots, D_n$ with non-linear parameters, we want to compute a partitioning into disjoint domains $\widetilde{D}_1, \ldots, \widetilde{D}_m$, so that:

(i) $\widetilde{D}_1 \uplus \cdots \uplus \widetilde{D}_m = D_1 \cup \cdots \cup D_n$
(which implies that $\{\widetilde{D}_1, \ldots, \widetilde{D}_m\}$ are pairwise disjoint)

(ii) $\forall i \in \{1, \ldots, m\}\ \forall j \in \{1, \ldots, n\}\ (\widetilde{D}_i \cap D_j \neq \emptyset \implies \widetilde{D}_i \subseteq D_j)$

Each domain $\widetilde{D}_i$ is associated with a list of all domains that are a superset of $\widetilde{D}_i$, independent of the choice of the parameters. To avoid additional case distinctions, we do not require $\widetilde{D}_i$ to be non-empty.

Let us illustrate the idea of the extended algorithm with a simplified version (written in Haskell notation):

```haskell
partition :: [Domain] -> [(Domain, [ID])]
partition ds = rec initUniverse ds
  where
  initUniverse = unionsD ds
  rec universe [] = [(universe, [])]
  rec universe (x:xs) =
    let withX    = universe 'intersectD' x
        withoutX = universe 'complementD' x
    in addID (rec withX xs) ++ rec withoutX xs
```

Explanation:

- Each recursive call of `rec` processes one `Domain x` and partitions the universe, which is initialized with $D_1 \cup \cdots \cup D_n$, into two regions:

  1. `withX` $=$ `universe` $\cap$ `x`
  2. `withoutX` $=$ `universe` $\setminus$ `x`

  These regions are partitioned recursively and the results are concatenated.

- `addID`:
  As `withX` is included in `x`, we need to add the identification (`ID`) of the `Domain x` to the result of the partitioning of `withX`. However, since it does not concern non-linear extensions, the details are not important in this context.

- The algorithm uses three basic operations provided by the `Domain` module:
  - `intersectD :: Domain` $\rightarrow$ `Domain` $\rightarrow$ `Domain`
    Computes the intersection of two domains.
  - `complementD :: Domain` $\rightarrow$ `Domain` $\rightarrow$ `Domain`
    Computes the complement of two domains.
  - `unionsD :: [Domain]` $\rightarrow$ `Domain`
    Computes the union of a non-empty set of domains. This function is a generalization of `unionD :: Domain` $\rightarrow$ `Domain` $\rightarrow$ `Domain`.

  All functions work for domains with non-linear parameters. They can be implemented using the formula representation of the `Domains`, which is described in section 4.1. Let $D_1, \dots D_n$ be `Domains` represented by the formulas $\Phi(D_1), \dots, \Phi(D_n)$:

D_1 `‘intersectD‘` D_2	$\leftrightarrow$	$\Phi(D_1) \wedge \Phi(D_2)$
D_1 `‘complementD‘` D_2	$\leftrightarrow$	$\Phi(D_1) \wedge \neg\Phi(D_2)$
`unionsD` $[D_1, \dots, D_n]$	$\leftrightarrow$	$\Phi(D_1) \vee \dots \vee \Phi(D_n)$

Remarks:

- The algorithm does not generate a decision tree. As a consequence, the domains returned by the algorithm might be empty for certain choices of the parameters. This is not a problem here, as the topological sorting algorithm does not require that all domains are non-empty.
- Further optimizations are possible, for example, `rec` can skip the recursive calls if the universe is already empty. Note that it must be empty for *all* possible choices of the parameters. Alternatively, a decision tree could be used to decide whether the universe is empty.

3.3. Disjoint Union

Before we can proceed with the topological sorting, it is necessary to assure that all domains returned by the partitioning, are *connected*. Let us illustrate the problem and its solution with a small example. Suppose we want to partition two domains D_1 and D_2 shown in Figure 8:

$$D_1 : \{i \mid 1 \leq i \leq n\}, \quad D_2 : \{i \mid 3 \leq i \leq 4\}$$

By applying the partition algorithm from the previous section, we obtain four partitions:

- $A : D_1 \cap D_2$ (containing D_1 and D_2)
- $B : D_1 \setminus D_2$ (containing only D_1)

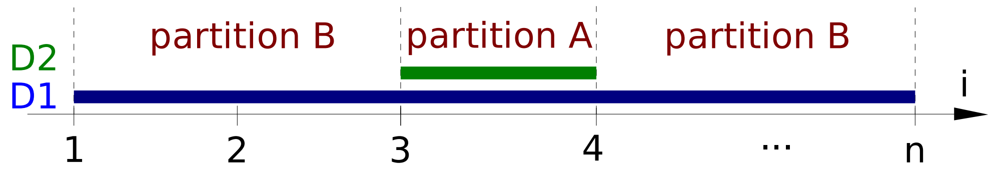


Figure 8: Shows two domains D_1 and D_2, which are partitioned into the domain A and the non-connected domain B (context: $n \geq 5$).

- $C : D_2 \setminus D_1$ (containing only D_2)
- $D : \emptyset$ (containing neither D_1 nor D_2)

Obviously, D can be removed, because it is empty for all choices of the parameters. In this example, we will additionally assume that $n \geq 5$ holds, so C can be removed as well, since the universe becomes $1 \leq i \leq n$, which is D_1. Thus, only A and B remain:

$$A : \{i \mid 3 \leq i \leq 4\}, \quad B : \{i \mid 1 \leq i \leq 2 \ \vee \ 5 \leq i \leq n\}$$

Note that B is not connected. Therefore, it is not possible to determine a legal topological sorting for A and B. We can verify that by refuting all possible orders:

1. $A \dashrightarrow B$ is not legal, since $2 \in B$ must be scanned before $3 \in A$.
2. $B \dashrightarrow A$ is not legal, since $4 \in A$ must be scanned before $5 \in B$.

As only connected domains can be topologically sorted, we have to separate B into a set of connected domains $\{B_1, \ldots, B_r\}$, so that $B = B_1 \uplus \cdots \uplus B_r$. If we additionally demand that all B_i are polyhedra, we benefit in two ways:

1. In contrast to domains, a polyhedron is always connected (independent of the choices of parameters).
2. As pointed out earlier in section 3.1, the final code generation accepts only polyhedra (i.e., the conversion can only be delayed, but not avoided).

The algorithm to compute a disjoint union for polyhedra with non-linear parameters can be found in the literature [Grö03, see `disj` (section 4.3.3)]. It is directly based on quantifier elimination and does not generate a decision tree.

Returning to our example, B can be split into two disjoint polyhedra $B_1 = \{i \mid 1 \leq i \leq 2\}$ and $B_2 = \{i \mid 5 \leq i \leq n\}$, so that $B = B_1 \uplus B_2$. As the second partition A is already a polyhedron, there is no need to separate it.

Now we can compute a topological sorting of the partitions, namely $B_1 \dashrightarrow A \dashrightarrow B_2$. In the next section, we will present the sorting algorithm.

3.4. Topological Sorting

Let $\{D_1, \ldots, D_n\}$ be set of partitions with non-linear parameters satisfying the following preconditions:

(i) Each domain D_i must be connected.

(ii) $\{D_1, \ldots, D_n\}$ must be pairwise disjoint.

Note that the results of the partitioning 3.2, on which the disjoint union algorithm 3.3 has been applied, satisfy both conditions. (i) holds, as each D_i is actually a polyhedron, in particular it is connected. (ii) holds, because the disjoint union algorithm only refined the results of the partitioning algorithm, thus its output $\{D_1, \ldots, D_n\}$ is still a set of partitions, which is also pairwise disjoint.

Before we can present the generalized algorithm for the topological sorting, we have to solve the subproblem of deciding whether D_1 can be sorted before D_2 with respect to the lexicographic order. Note that we consider only the current dimension d, because we assume that the previous dimensions $1, \ldots, d-1$ are already sorted.

$D_1 \trianglelefteq D_2$ (i.e., D_1 can be sorted before D_2) if and only if

$$\forall x_1 \cdots \forall x_d \; \forall y_1 \cdots \forall y_d \\ \Big(\big(\underbrace{\Phi(D_1)}_{(x_1,\ldots,x_d)\in D_1} \;\wedge\; \underbrace{\Phi(D_2)}_{(y_1,\ldots,y_d)\in D_2} \wedge x_1 = y_1 \wedge \cdots \wedge x_{d-1} = y_{d-1} \big) \implies x_d \leq y_d \Big)$$

Then it is possible to compute a topological order for the domains D_1 and D_2. We distinguish four cases:

1. $D_1 \trianglelefteq D_2 \wedge \neg(D_2 \trianglelefteq D_1)$ implies $D_1 \dashrightarrow D_2$ (i.e., D_1 must be scanned before D_2)
2. $\neg(D_1 \trianglelefteq D_2) \wedge D_2 \trianglelefteq D_1$ implies $D_2 \dashrightarrow D_1$
3. $D_1 \trianglelefteq D_2 \wedge D_2 \trianglelefteq D_1$ both orders are valid ($D_1 \dashrightarrow D_2$ as well as $D_2 \dashrightarrow D_1$)
4. $\neg(D_1 \trianglelefteq D_2) \wedge \neg(D_2 \trianglelefteq D_1)$ cannot be reached

This leads to the decision tree shown in Figure 9. Now we are ready to extend the algorithm from two domains to n domains $D_1, \ldots, D_n$:

Algorithm (Sketch):

1. Generate a decision tree, whose leaves are directed acyclic graphs $G = (V, E)$:
   - $V = \{1, \ldots, n\}$ (One vertex for each domain $D_1, \ldots, D_n$)
   - $E = \{(i, j) \in V^2 \mid D_i \trianglelefteq D_j \wedge \neg(D_j \trianglelefteq D_i)\}$
     (Add an edge from D_i to D_j if D_i must be scanned before D_j. Note that each call to $\trianglelefteq$ returns a condition, which may lead to a new branch in the tree.)
2. Apply a (standard) topological sorting on each leaf of the decision tree (i.e., on each graph).

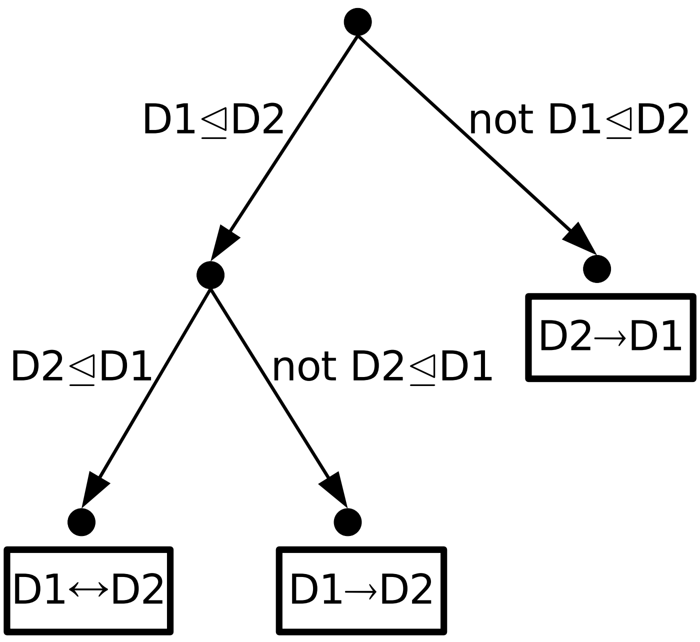


Figure 9: Decision tree to determine a topological order of the domains D_1 and D_2.

Remarks:

- The termination of the algorithm is guaranteed, as there are at most n^2 edges to test and each test can only lead to one additional decision (add edge: yes/no). Thus, the resulting decision tree cannot contain infinite paths.
- As the second step is significantly faster than the first step, it is important to minimize the number of comparisons (i.e., calls to $\trianglelefteq$).
- Some simple optimizations used in our implementation:
  - Combine the test, whether the edge $D_i \dashrightarrow D_j$, $D_j \dashrightarrow D_i$ or none of them should be added (as illustrated in the decision tree from Figure 9).
  - Use the transitivity of the edges to reduce the number of tests.
  - Exclude all edges that would lead to a cycle when added.

Nevertheless, the complexity of the algorithm remains $O(n^2)$ calls of $\trianglelefteq$, where n is the number of domains. The worst case occurs when the resulting graph contains no edges, because none of the above-mentioned optimization will work.

- If the graph contains only a few nodes, it becomes likely that two arbitrary domains are incomparable, that means every order is legal. This could be used to apply a divide-and-conquer approach:

  1. Partition the set of domains $\{D_1, \dots, D_n\}$ into two sets $P_1 = \{D_{i_1}, \dots, D_{i_r}\}$ and $P_2 = \{D_{i_{r+1}}, \dots, D_{i_n}\}$, so that each $X \in P_1$ is incomparable to each $Y \in P_2$, that means $X \trianglelefteq Y$ and $Y \trianglelefteq X$.
  2. Sort P_1 and P_2 independently and concatenate the results.

  For example, if $r = \frac{n}{2}$ then the number of comparisons reduces from n^2 to $2 \cdot r^2$, which is $\frac{n^2}{2}$. It approximates $n \cdot \log n$ if applied recursively (assuming $r = \frac{n}{2}$ on average).

  Since sparse graphs occurred frequently in our tests, future work might examine heuristics to find such a partition efficiently.

3.5. Code Generation

The final step is to generate code for the resulting loop nests. The original approach was to solve each loop description for its associated loop variable, which yields the loop's upper and lower bounds.

However, as the coefficients of the variables are no longer constants, it is more difficult to solve the inequalities, because it is necessary to know the sign of the coefficient. By using a decision tree, we can avoid this problem.

Let us illustrate it with a small example:

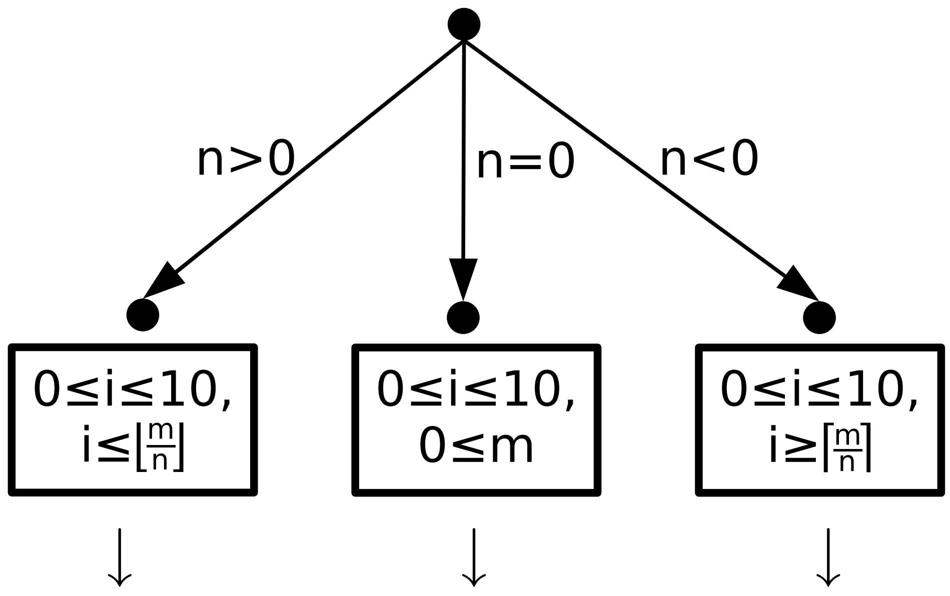
$$L.1 : \{i \mid 0 \leq i \leq 10, n \cdot i \leq m\} :: S$$

To solve $L.1$ for the variable i, we have to divide the second inequality by the parameter n (or substitute n if it is zero). Thus, we have to distinguish three cases ($n > 0$, $n = 0$, $n < 0$) leading to the decision tree shown in Figure 10.

3.6. Example

Let us test our implementation of the extended Quilleré algorithm with the introductory example 1.1, which contains one parameter M. First, we will make no assumptions on M. The result is a decision tree with 12 leaves, where each leaf contains the loop nests for that specific case. However, in practice you often have additional knowledge, for instance, $M \geq 1$, which we will call the *context*. It can be used to reduce the number of leaves and simplify their included inequality systems. This will also be discussed in section 4.2.

Suppose we know that M is an integer between 1 and 100. We obtain the simplified decision tree illustrated in Figure 11.


n>0
n=0
n<0
$0 \le i \le 10,\ i \le \lfloor \frac{m}{n} \rfloor$
$0 \le i \le 10,\ 0 \le m$
$0 \le i \le 10,\ i \ge \lceil \frac{m}{n} \rceil$

```
for i=0..min(10,floor(m/n))
  S;
```

```
if (m >= 0) {
  for i=0..10
    S;
}
```

```
for i=max(0,ceil(m/n)..10)
  S;
```

Figure 10: The decision tree, which resulted from solving the loop description $L.1$ for the variable i. Each of the leaves can be straightforward translated to the target code shown below.

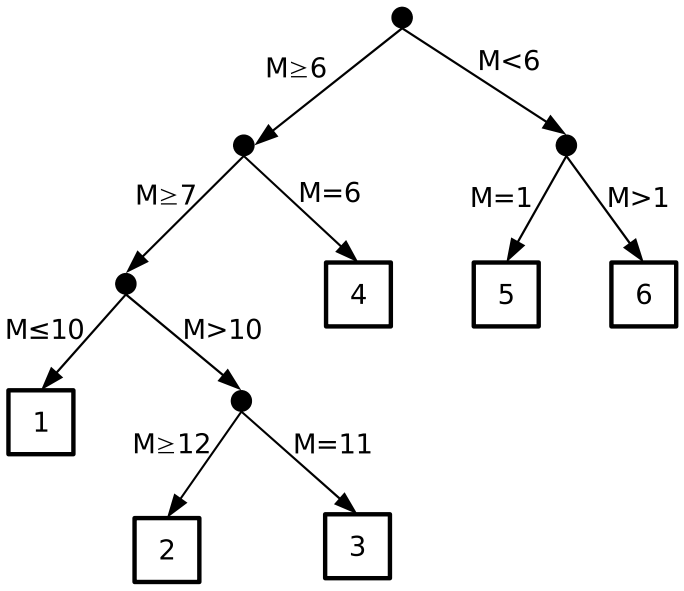


Figure 11: The simplified decision tree for the introductory example (simplified with the context $M \in \{1, \ldots, 100\}$). Each box contains the loop nests resulting from the extended Quilleré algorithm.

Interpretation (of the decision tree shown in Figure 11):

- The decision tree distinguishes six classes (depending on the parameter M):
  - $\boxed{1}$ if $M \in \{7, 8, 9, 10\}$
  - $\boxed{2}$ if $M \in \{12, \ldots, 100\}$
  - $\boxed{3}$ if $M = 11$
  - $\boxed{4}$ if $M = 6$
  - $\boxed{5}$ if $M = 1$
  - $\boxed{6}$ if $M \in \{2, 3, 4, 5\}$
- Let us take a closer look at the content of box $\boxed{1}$. Its inequality systems have been simplified using its associated context $M \in \{7, 8, 9, 10\}$.

```
L1: {t | 0 <= t <= 5} ::
  L1.1: {t,r | r = 0} ::
    S1;
    S3;
```

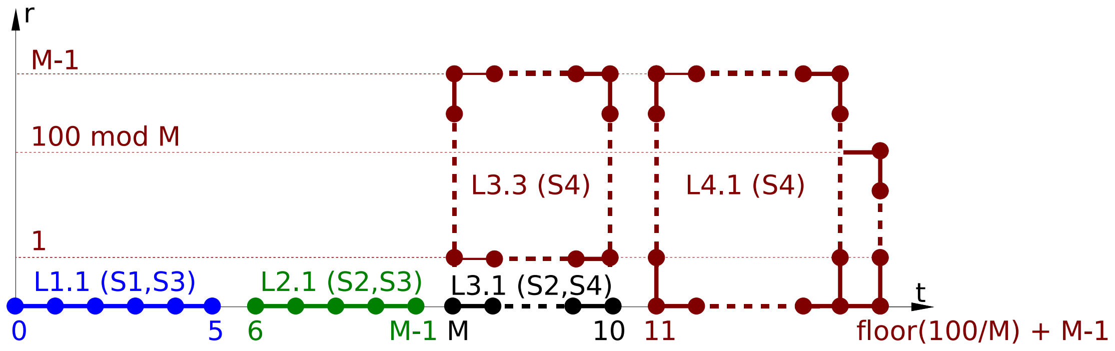


Figure 12: Illustration of the resulting loop nests with their statements for the context $M \in \{7, 8, 9, 10\}$.

```
L2: {t | 6 <= t <= M-1 } ::
  L2.1: {t,r | r = 0} ::
    S2;
    S3;

L3: {t | M <= t <= 10 } ::
  L3.1: {t,r | r = 0} ::
    S2;
    S4;
  L3.2: {t,r | false } ::
    S2;
  L3.3: {t,r | 1 <= r <= M-1 } ::
    S4;

L4: {t | 11 <= t <= floor(100/M) + M-1 } ::
  L4.1: {t,r | 0 <= r <= M-1, r <= M^2 - M + 100 - M*t} ::
    S4;
```

Note that the last inequality $r \le M^2 - M + 100 - M \cdot t$ is only relevant if t is maximal, which is when $t = \lfloor \frac{100}{M} \rfloor + M - 1$. In that case, the inequality could be further simplified:

$$\begin{aligned} r &\le M^2 - M + 100 - M \cdot t \\ &= M^2 - M + 100 - M \cdot \left(\left\lfloor \frac{100}{M} \right\rfloor + M - 1 \right) \\ &= M^2 - M + 100 - M \cdot \left\lfloor \frac{100}{M} \right\rfloor - M^2 + M \\ &= 100 - M \cdot \left\lfloor \frac{100}{M} \right\rfloor \\ &= 100 \mod M \end{aligned}$$

The complete target code is shown in Appendix B. However, it is less optimized than the loop descriptions in this section, which were manually simplified to improve the readability. Thus, many of the guards and bounds in the code could be removed.

4. Implementation

4.1. Simultaneous Matrix/Formula Representation

Each `Domain` stores two equivalent descriptions of their associated domain internally:

1. List of matrices (i.e., inequality systems representing a union of polyhedra[4])
2. Logical formula

It depends on the situation, which representation is more useful. In general, we use formulas for intermediate results, especially when the external logic tools are involved (which is most of the time). But on some occasions, when we need more control, it is preferable to work on the list of matrices. For instance, during the topological sorting, where all domains must be connected, we assure that each domain is a polyhedron (i.e., its list of matrices has only a single entry).

First, let us illustrate how to convert between these two representations. As a convention, we name variables $x_1, \dots, x_n$ and parameters $p_1, \dots, p_m$.

List of matrices → Formula Suppose we want to compute a describing formula $\Phi(D)$ for a given domain D. As a domain is a finite union of polyhedra, we can break down the problem:

$$\Phi(D) = \Phi(P_1 \cup \dots \cup P_r) = \Phi(P_1) \vee \dots \vee \Phi(P_r)$$

Each polyhedron can be described by a conjunction of inequalities. Note that the resulting formula $\Phi(D)$ is in disjunctive normal form (DNF) without negations and contains only weak inequalities.

Formula → List of matrices The opposite direction is more complicated, as formulas are more expressive than our inequality systems, resulting in two necessary restrictions for the input formula Φ:

1. The variables $x_1, \dots, x_n$ may only appear as linear expressions. So all terms in the original formula Φ can be written as $\alpha_0 + \alpha_1 x_1 + \dots + \alpha_n x_n$, where α_i are arbitrary polynomials in the parameters.
   Note that α_i must not contain variables. As a consequence, the terms x_1^2 and $x_1 \cdot x_2$ are invalid, but $p_1^2 p_2^3 x_1 + p_2 x_2 + 3p_2$ is permitted.

[4] We neither require that the polyhedra are pairwise disjoint nor that each polyhedron is non-empty.

2. We consider only integral points. That allows us to eliminate strict inequalities (without introducing additional negations), since $z > 0 \iff z \geq 1$ holds for $z \in \mathbb{Z}$.

   If there are monomials with rational coeffients, we first have to make them integral (by multiplying the inequality with their least common denominator). For example, $\frac{3}{2}x_1 + \frac{4}{3}x_2 > 0$ will be transformed to $9x_1 + 8x_2 \geq 1$.

Returning to the problem, we compute a disjunctive normal form for Φ and eliminate all negations and strict inequalities. Note that this expanded formula is of the same form as $\Phi(D)$ resulting from the previous conversion **List of matrices $\rightarrow$ Formula**. By reversing the steps above, we obtain the associated domain D.

It is possible that the logic tools that we are using, produce formulas, which are not according to the restrictions above, even though they describe a domain. That means there exists an equivalent formula respecting the restrictions.

For example, the formulas $x = 1 \vee x = -1$ and $x^2 = 1$ are equivalent in $\mathbb{R}$, but only the first formula is linear in x. But as the second formula is smaller, it may be preferred by the external tools. To solve that problem, we use a tool named SLFQ [SLFQ], which is a simplifier and degree decreaser. It guarantees that the polynomials in the simplified formula are of minimal degree (i.e., a formula with linear variables). In our example, the formula found by SLFQ is $x \geq -1 \wedge x \leq 1 \wedge (x = 1 \vee x = -1)$, which can be further simplified to $x = 1 \vee x = -1$.

Remarks:

- We use lazy evaluation to eliminate the maintenance overhead, that means each representation is only evaluated when it is actually needed. Let us illustrate this idea with an example, which describes a common scenario:

  1. Initialize a `Domain` X_1 with a list of matrices.
  2. Apply a sequence of operations that work only on the formula representation (e.g., partitioning, simplification, projection).

     Let X_n be the final result and $X_2, \ldots, X_{n-1}$ the intermediate results, which are not used elsewhere.
  3. Apply an operation that requires the list of matrices of X_n (e.g., display them).

  X_1 and X_n require that both representations have to be evaluated, but the intermediate results $X_2, \ldots, X_{n-1}$ do not use their list of matrices, consequently these are not evaluated. This behavior is shown in Table 1.

- The computation of a disjunctive normal form can theoretically lead to an exponential growth of the resulting formula. Although we have not observed it in our examples, it is still preferable to avoid switching between different representations too often.

representation	X_1	X_2	$\cdots$	X_{n-1}	X_n
list of matrices	initialization	not evaluated		not evaluated	conversion
	$\downarrow$				$\uparrow$
formula	conversion	$\to$ computation $\to$	$\cdots$	$\to$ computation $\to$	computation

Table 1: Shows which representations of the `Domains` $X_1, \ldots, X_n$ have to be evaluated (either by initialization, conversion or computation). Arrays denote the flow of control.

- Some operations can be efficiently computed in both representations, which can be used as an optimization to save conversions.

  For example, let D_1 and D_2 be two `Domains`, which are represented

  - by their formulas $\Phi(D_1)$ and $\Phi(D_2)$.
  - by their list of inequality systems represented as matrices $\{P_1^{(1)}, \ldots, P_n^{(1)}\}$ and $\{P_1^{(2)}, \ldots, P_m^{(2)}\}$.

  The union $D_1 \cup D_2$ will be simultaneously (but lazy) computed in both representations:

  Formula $\Phi(D_1) \vee \Phi(D_2)$

  List of matrices $\{P_1^{(1)}, \ldots, P_n^{(1)}, P_1^{(2)}, \ldots, P_m^{(2)}\}$

- Each decision tree, whose leaves are `Domains`, can be expressed as a single first-order formula, which can in turn be converted to a `Domain`. Although this is not practical, as the resulting formula grows too big, it explains how decisions can be "hidden" inside a `Domain` (cmp. section 3.1).

4.2. Simplifications

As we have seen in the previous section, formulas are used for intermediate results. It is important to simplify them, since the external logic tools work well on small formulas, but may fail if the formulas grow too big. The quality of the simplification can be improved, when we have additional context information. In this section, we will examine two basic strategies:

1. Simplification of the intermediate results.

2. Simplification of the final decision tree (i.e., reducing its size).

Note that both strategies are not mutually exclusive, but they can be combined. Let us now discuss each strategy on its own, starting with the first one.

Suppose we have a domain D, which is represented by a logical formula $\Phi(D)$, and a context C. This context is represented as a conjunction of logical formulas containing information about the parameters used in D (e.g., $p > 0 \wedge p \leq q$).

Although simplifying the intermediate results involves additional cost, it can help to reduce overall consumed time. But much depends on the simplifier that is being used, so it is difficult to make general statements. The main criterion for a suited simplifier is that it simplifies *fast*, whereas it is less important that it simplifies *smart*.

Simplifiers that are degree decreasers (e.g., SLFQ) should be handled with care, as it is possible that they increase the number of polyhedra in the intermediate results. For instance, let D be a domain that can be described by a single polyhedron P_1 (where x is variable and p is a parameter):

$$\Phi(D) = \Phi(P_1) = \{x \mid x = p \wedge p^2 = 1\}$$

which is equivalent to

$$\Psi = \{x = p \wedge (p = 1 \vee p = -1)\} = \{x \mid (x = 1 \wedge p = 1) \vee (x = -1 \wedge p = -1)\}$$

Although the second formula Ψ could be considered "easier" than the original formula $\Phi(D)$, it is no longer a conjunction. That means the algorithm from the previous section 4.1 would convert it into a domain $\widetilde{D}$ consisting of two polyhedra $\widetilde{P_1}$ and $\widetilde{P_2}$:

$$\widetilde{D} = \widetilde{P_1} \cup \widetilde{P_2}, \text{ where } \widetilde{P_1} = \{x \mid x = 1 \wedge p = 1\} \text{ and } \widetilde{P_2} = \{x \mid x = -1 \wedge p = -1\}$$

As the topological sorting 3.4, which is the most time consuming part of the extended Quilleré, involves $O(n^2)$ comparisons with n being the number of polyhedra, we end up with additional comparisons. It is likely that it also leads to additional leaves in the decision tree. The consequence could be a worse performance than without simplification.

However, if we have the non-empty context $C = \{p > 0\}$, we could simplify the original formula $\Phi(D)$ to $\{x \mid x = 1\}$, which is certainly the best possible representation in that case. Note that this last simplification is not trivial, so you cannot expect that a simplifier, which is optimized for speed, is able to find it.

Nevertheless, it is advantageous to reduce the degree of the polynomials in the formula when it improves the performance of the logic tools. For instance, there are specialized algorithms for quantifier elimination of linear [Wei88] and quadratic formulas [Wei97]. In the linear case, we could even use some functionality of PolyLib [PolyLib], which is a library for linear-parameterized polyhedra and therefore an order of magnitude faster than the general-purpose logic tools.

Next, let us analyze the second strategy, which concerns the simplification of the final decision tree by removing irrelevant subtrees (see section 2.2). From every eliminated subtree, we benefit in two ways:

1. The overall computation time is reduced, since no code has to be generated for the removed subtrees. The time needed for deciding whether a subtree can be cut is not relevant (compared to the total time).

2. The generated target code becomes more efficient, because the decision tree contains less case distinctions, which would otherwise be evaluated at runtime.

Note that only the final decision tree needs to be simplified, which requires that the evaluation of the intermediate decision trees is delayed (*lazy evaluation*).

In contrast to the simplification of the intermediate results, the simplification of the final decision tree is generally not time critical. As each removed subtree will drastically reduce the total time, we should use all available context information to detect all branches that can be eliminated.

Further background information on the simplification of quantifier-free formulas can be found in the literature [DS97].

4.3. Experiments

In this section, we will present the results of our experiments on our introductory example 1.1. To get a better understanding of the impact of the logic tools, we tested the extended Quilleré algorithm with different simplifiers and context information. We ran the experiments on a Dual Core AMD Opteron$^{\text{TM}}$ with 2200 MHz and 2 GB RAM. The results are collected in Table 2, which reads from left to right as follows:

- *context 1* is the context that is available to the simplifier of the intermediate results. It will be ignored if the simplification of the intermediate results is turned off.
- *context 2* is used by the tree simplifier to reduce the size of the final decision tree by eliminating all leaves that are redundant in terms of this context. Note that the tree simplifier should not be confused with the simplifier of the intermediate results.
- The *formula simplifier* that is used to simplify the intermediate and final results:

REDLOG 3.0	In combination with REDUCE 3.8
SLFQ 1.9	In combination with QEPCAD B 1.44
none	Turns off the simplification of intermediate results (but not the simplification of the decision tree)

- The total computation *time*.
- The number of *leaves* of the final decision tree.
- The number of *lines* of target code.
- *expr* counts the number of expressions in the target code, which is the sum of all lower and upper bounds and the number of guards.

The experiments show that the tree simplifier profits from additional context information reducing both the total computation time and the complexity of the target code. It is also important use a formula simplifier, at least once before the code generation. Otherwise the target code contains a lot of redundant lower and upper bounds.

Surprisingly, the simplification of the intermediate results is less effective when the simplifiers are called with a non-empty context. The problem is that it depends on the

context 1	context 2	formula simplifier	time	#leaves	#lines	#expr
—	—	none	447 sec	12	468	1753
—	—	Redlog	625 sec	12	468	902
—	—	Slfq	496 sec	12	474	603
—	$M \geq 1$	none	379 sec	10	427	1658
—	$M \geq 1$	Redlog	555 sec	10	427	847
—	$M \geq 1$	Slfq	425 sec	10	431	554
$M \geq 1$	$M \geq 1$	Redlog	635 sec	13	586	1079
$M \geq 1$	$M \geq 1$	Slfq	588 sec	13	562	732
—	$1 \leq M \leq 100$	none	340 sec	9	401	1573
—	$1 \leq M \leq 100$	Redlog	514 sec	9	401	807
—	$1 \leq M \leq 100$	Slfq	380 sec	9	403	521
$1 \leq M \leq 100$	$1 \leq M \leq 100$	Redlog	598 sec	12	512	984
$1 \leq M \leq 100$	$1 \leq M \leq 100$	Slfq	543 sec	12	512	688
—	$M \in \{1, \ldots, 100\}$	none	388 sec	6	258	1013
—	$M \in \{1, \ldots, 100\}$	Redlog	506 sec	6	258	515
—	$M \in \{1, \ldots, 100\}$	Slfq	378 sec	6	260	337
$M \in \{1, \ldots, 100\}$	$M \in \{1, \ldots, 100\}$	Redlog	699 sec	6	274	534
$M \in \{1, \ldots, 100\}$	$M \in \{1, \ldots, 100\}$	Slfq	time-out	—	—	—
—	$M = 7$	none	84 sec	1	45	185
—	$M = 7$	Redlog	115 sec	1	45	95
—	$M = 7$	Slfq	101 sec	1	45	61
$M = 7$	$M = 7$	Redlog	39 sec	1	55	98
$M = 7$	$M = 7$	Slfq	60 sec	1	55	71

Table 2: A summary of experiments with our introductory example. Context 1 is the context information that is available to the simplifier of the intermediate results, while context 2 is used to reduce the size of the final decision tree. Each line shows the elapsed time, the number of leaves of the final decision tree, the number of lines of target code and the number of expressions in this code.

application, whether a formula Φ_1 is "simpler" than an equivalent formula Φ_2. Some sophisticated simplifications can even be harmful if they increase the number of polyhedra (e.g., degree reduction), as the number of case distinctions might increase as well. However, since we used general-purpose formula simplifiers, we cannot conclude that context information is a disadvantage for the simplification of the intermediate results.

Remarks:

- At first glance, a context like $M = 7$ does not seem to be realistic in a scenario with non-linear parameters. However, it might appear deep inside the decision tree, where we have additional context information from the conditions of the tree.

  For example, in order to solve the inequality $i \cdot (M - 7) > 0$ for i, we obtain three new branches: $M > 7$, $M < 7$ and $M = 7$. The latter branch can be simplified under the context $M = 7$ (e.g., by substituting all occurrences of M by 7).

- The context $M \in \{1, \ldots, 100\}$ implies that $M \in \mathbb{Z}$, thus it is more restrictive than $1 \leq M \leq 100$, which requires only that $M \in \mathbb{R}$. Since $M \in \mathbb{Z}$ cannot be expressed by first-order formulas in the real numbers, we used a finite case distinction instead, namely $M = 1 \vee \cdots \vee M = 100$. However, the resulting formula is very big, so it does not work with SLFQ.

  Although this approach is not recommendable for real-work examples, it shows that the results could be improved when the tree simplifier can use the fact that M is an integer. The reason is that decision trees typically contain redundant leaves, which can only be reached if at least one of the parameters is not an integer.[5]

5. Conclusions

This work shows that it is feasible to extend the polytope model to allow inequality systems with non-linear parameters. We demonstrated how an algorithm, which is restricted to linear parameters, can be extended. We gave the Quilleré algorithm as an example, but other components of the polytope framework could also be extended, using the same techniques.

Although our examples are very simple, since they involved only a few variables and parameters, further optimizations are possible to increase the range of application. As pointed out in the introduction, there are two basic approaches to extend algorithms. In this work, we expressed the critical steps of the algorithm directly using first-order logic. Note that we deviated from the original algorithm, as we generate domains in the projection phase, but not polyhedra.[6] From an abstract point of view, this is only an

[5] It is not possible to construct a general tree simplifier that is capable of eliminating all redundant leaves, because it is undecidable whether a given condition, which can be any Diophantine equation of the form $f(p_1, \ldots, p_n) = 0$, has an integral solution (Hilbert's tenth problem).

[6] As pointed out in section 3.1, it is also possible to use an extended version of Fourier-Motzkin for the projection. In that case, we would receive a decision tree of polyhedra instead of a single domain. Although we are closer to the original Quilleré algorithm, it is likely that the final target code will contain more case distinctions.

implementation detail, as we revert to polyhedra after the partitioning. The advantage is that between these two steps we are able to use the full expressiveness of the first-order logic, so we can take more advantage of the logic tools.

The alternative would be to strive for an accurate translation of each step of the linear algorithm and to insert a case distinction, whenever information about the parameters are needed that are unknown at compile time [Grö03, section 4.2]. But as the original algorithm was tuned for linear parameters, it is assumed that some optimizations are no longer effective in the generalized algorithm.

In both cases, the performance depends heavily on the logic tools that are being used. Since they were mainly developed for general purpose use, they cannot take advantage of the specific situation. For instance, the tools could distinguish between variables and parameters to assure that all variables appear only as linear expressions. This would eliminate the need for external degree decreasers. Additionally, a more specialized simplifier could minimize the number of polyhedra by producing formulas that are in disjunctive normal formula, where the number of disjunctions is minimal. As you can expect further improvements in the area of symbolic computing, it seems realistic that the generalized polytope model with non-linear parameters could be applied in future parallelizing compilers.

A. Overview of the most important modules

This section contains only a brief overview of some modules that are used in this work. Further information can be found in the documentation.

A.1. LMath.Logic.Domain

Provides the `Domain` data type, which represents a union of polyhedra with non-linear parameters. Here is a overview of the operations that are frequently used in this work:

- Basic operations
  - Union of two `Domains` (`unionD`)
  - Union of a non-empty set of `Domains` (`unionsD`)
  - Intersection of two `Domains` (`intersectD`)
  - Complement of two `Domains` (`complementD`)
  - Simplification of one `Domain` (`simplifyD`)
- Extended operations
  - Partitioning of a set of `Domains` (`lPartitionD`)
  - Disjoint union of a `Domain` into a set of polyhedra (`disjointUnionD`)
  - Topological sorting of a set of `Domains` (`lTopSortD`)
  - Conversion of a first-order formula to a `Domain` (`formulaToDomain`)
  - Extract the first-order formula from a `Domain` (`domainToFormula`)

A.2. LMath.Logic.DomainTree

Extends the operations defined in `LMath.Logic.Domain`, which mostly work on `Domains`, to expect decision trees of `Domains` as input.

A.3. LMath.Types.DTree.Merge

Provides functions to merge several decision trees, using a given merge operation (e.g., concatenation).

B. Target code

This section contains the target code from example 3.6. The final loop descriptions have been simplified using SLFQ, but there are still redundant guards (`if`-statements) and upper and lower bounds left, which could be removed as well.

```
for (int t = 0; t <= min(M - 1, 5); t++) {
  if (-t + 5 >= 0 && t >= 0) {
    for (int r = 0; r <= 0; r++) {
      S1;
      S3;
    }
  }
}


for (int t = 6; t <= min(M - 1, 10); t++) {
  if (-t + M - 1 >= 0 && t - 6 >= 0) {
    for (int r = 0; r <= 0; r++) {
      S2;
      S3;
    }
  }
}


if (M - 1 >= 0) {
  for (int t = max(ceildiv(M*M - M + 1, M), 6); t <= 10; t++) {
    if (t - M >= 0 && -t + 10 >= 0) {
      for (int r = 0; r <= 0; r++) {
        S2;
        S4;
      }
    }
    if (t*M - M*M + M - 1 >= 0 && -t + M - 1 >= 0 && -t + 10 >= 0) {
      for (int r = 0; r <= 0; r++) {
        S2;
      }
    }
    if (t*M - M*M + M - 1 >= 0 && -t + 10 >= 0) {
      for (int r = max(-t*M + M*M, 1); r <= M - 1; r++) {
        S4;
      }
    }
  }
}
```

```
if (M - 1 >= 0 && -M + 100 >= 0) {
  for (int t = max(ceildiv(M*M - M + 1, M), 11);
       t <= floordiv(M*M - M + 100, M); t++) {
    if (-t*M + M*M - M + 100 >= 0 && t - 11 >= 0) {
      for (int r = 0; r <= min(-t*M + M*M - M + 100, M - 1); r++) {
        S4;
      }
    }
  }
}
```

References

[DS97] Andreas Dolzmann and Thomas Sturm. *Simplification of quantifier-free formulae over ordered fields.* Journal of Symbolic Computation, 24(2):209-231, August 1997.

[GFL04] Martin Griebl, Peter Faber and Christian Lengauer. *Space-time mapping and tiling: a helpful combination.* Concurrency and Computation: Practice and Experience, 16(3):221–246, March 2004.

[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`.

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

[Qui00] F. Quilleré and S. Rajopadhye and D. Wilde. *Generation of efficient nested loops from polyhedra.* International Journal of Parallel Programming, 28(5):469–498, October 2000.

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

[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.

[PolyLib] `http://icps.u-strasbg.fr/polylib/`

[QEPCAD] `http://www.cs.usna.edu/~qepcad/B/QEPCAD.html`

[Redlog] `http://www.fmi.uni-passau.de/~redlog/`

[Reduce] `http://www.zib.de/Symbolik/reduce/`

[SLFQ] `http://www.cs.usna.edu/~qepcad/SLFQ/Home.html`

[C99] *ISO/IEC 9899:1999/Cor.1:2001 (E)*, IO/IEC, 2001.