Scanning Index Sets with Polynomial Bounds Using Cylindrical Algebraic Decomposition

Armin Größlinger

Department of Informatics and Mathematics, University of Passau
armin.groesslinger@uni-passau.de

Technical Report, Number MIP-0803
Department of Informatics and Mathematics
University of Passau, Germany
June 2008

Scanning Index Sets with Polynomial Bounds Using Cylindrical Algebraic Decomposition

Armin Größlinger

University of Passau
Department of Informatics and Mathematics
armin.groesslinger@uni-passau.de

Abstract. Automatic, model-based program transformation relies on the ability to generate code from a model description of the program. In the context of automatic parallelisation, cache optimisation and similar transformations, the task is to generate loop nests which enumerate the iteration points within given domains. Several approaches to code generation from polyhedral descriptions of iteration sets have been proposed and are in use. We present an approach to generating loop nests for index sets with arbitrary polynomials as bounds using cylindrical algebraic decomposition. The generated loops are efficient in the sense that no integer superset is enumerated. We also state where this technique is useful, i.e., where non-linearities in the loop bounds arise in loop program transformations and show some examples for our approach with polyhedral and non-polyhedral input.

1 Introduction

Optimisation of program execution is an ubiquitous challenge in computer science. Programs have to be adapted to exploit parallelism, use caches efficiently or save power on new architectures. Using a model representation of the programs in question, one can facilitate an automatic or semi-automatic transformation process. The model represents the characteristics of the program at an abstract level, modelling, e.g., the execution order of dependent operations or the memory accesses of the program. The transformation is performed in this model which permits an optimising search for the best transformation to achieve the desired target program, e.g., infusing a maximum of parallelism or a minimum of cache misses. After the model of a program has been transformed, corresponding code which can execute on the target architecture has to be generated from the model.

One successful example of such a model is the so-called polyhedron model (previously called the polytope model) of loop programs [Len93] in which loops are modelled by polyhedra. It has been used successfully to infuse parallelism and enhance cache behaviour. The code generation step has long been a subject of study, cf. Section 3. But not all useful transformations of programs can be expressed by linear algebra on polyhedra which have linear bounds; there has been an increasing demand for non-linear transformations.

We drop the restriction to polyhedral index sets, allowing arbitrary semi-algebraic sets, i.e., index sets bounded by polynomials in the variables and parameters. We give an example demonstrating the limitations of the polyhedron model and illustrating the idea of our code generation technique in Section 2. After discussing related work in Section 3, we give a precise formulation of the problem we solve and our main result, its solution, in Section 4. Section 5 presents some examples for polyhedral and non-polyhedral inputs. Section 6 discusses possible future improvements of the code generation algorithm and Section 7 concludes.

2 Introductory Examples

We introduce the problem of code generation by means of a simple example in Section 2.1. A reader who is familiar with polyhedral code generation may want to skip to Section 2.2, which illustrates the additional challenges for code generation in our more general case.

2.1 Introduction to Code Generation

When we speak of code generation, we aim at generating loops that enumerate so-called index sets and execute statements (loop bodies) for each enumerated point. For example, let us generate code for two statements T_1 and T_2, where T_1 is executed at every point in $D_1 = \{x \,|\, 2 \leq x \leq 8\}$ and T_2 at every point in $D_2 = \{x \,|\, 2 \leq x \leq p\}$. The sets D_1 and D_2 are called the index sets of T_1 and T_2, respectively. Since T_1 is to be executed for $x = 2, \ldots, 8$ and T_2 for $2, \ldots, p$ (for $p \in \mathbb{Z}$), we have to generate loops with the index variable x which enumerate the respective x-values and execute T_1 and T_2 at the respective index points. Unfortunately, enumerating the x-values for the two statements independently, as in the following sequence of loops:

```
for (x=2; x<=8; x++)
  T1;
for (x=2; x<=p; x++)
  T2;
```

is not the solution we desire, because the enumeration of the index points has to respect the ordering on the index variable x. For example, the execution of T_2 for $x = 2$ may happen before or after T_1 for $x = 2$, but it must happen before any execution of T_1 or T_2 for $x \geq 3$. The generation of correct code is complicated by the fact that we do not know the value of the upper bound p at code generation time, so the emitted code must work for all possible (integral) values of p. Figure 1 shows three possible codes which enumerate the index sets correctly. The figure illustrates that there is tradeoff between code size and efficiency of the generated code. The code in Figure 1(a) specifies the evaluation of two conditionals (in the `if` statements) in every iteration of the loop. The codes in Figures 1(b) and 1(c) have no overhead for evaluating conditions inside

the loops. With the case distinctions on p, the code in Figure 1(c) never executes an empty loop, i.e., when a loop is reached, the upper bound is guaranteed to be greater than or equal to the lower bound. This property comes at the price of an increased code length. The code in Figure 1(b) is shorter and there are no case distinctions in p apart from the loop bounds, but loops may be empty, for example, the last loop is empty for $p \leq 8$. The algorithm we present subsequently produces code without conditionals inside the loops.

```
for (x=2; x<=max(8,p); x++) {
  if (2 <= x && x <= 8)
    T1;
  if (2 <= x && x <= p)
    T2;
}
```

(a) simple code with conditionals

```
for (x=2; x<=min(8,p); x++)
  { T1; T2; }
for (x=max(p+1,2); x<=8; x++)
  T1;
for (x=9; x<=p; x++)
  T2;
```

(b) tricky loop bounds

```
if (p >= 9) {
  for (x=2; x<=8; x++)
    { T1; T2; }
  for (x=9; x<=p; x++)
    T2;
} else if (p == 8) {
  for (x=2; x<=8; x++)
    { T1; T2; }
} else if (p >= 2) {
  for (x=2; x<=p; x++)
    { T1; T2; }
  for (x=p+1; x<=8; x++)
    T1;
} else {
  for (x=2; x<=8; x++)
    T1;
}
```

(c) case distinctions on p

Fig. 1. Three possible codes for $D_1 = \{x \,|\, 2 \leq x \leq 8\}$ and $D_2 = \{x \,|\, 2 \leq x \leq p\}$

In general, the ordering of the operations is determined by the *lexicographic order* of the index set points. For example, in the case of a two-dimensional index set with (x, y) coordinates, the outer loop of the generated code enumerates the x-dimension, and an inner loop enumerates the y-dimension in dependence of x, i.e., for given x, all values y such that (x, y) is in the index set are enumerated. The main task of code generation is to partition the index sets of the statements such that each partition can be scanned by a loop nest. In the above example, a suitable disjoint union of the domains $D_1 \cup D_2 = U_1 \dot\cup U_2 \dot\cup U_3$ is given by the following three sets:

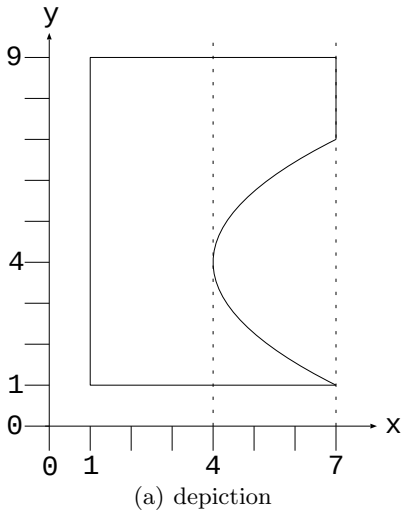
$$
\begin{aligned}
U_1 &= \{x \,|\, 2 \leq x \leq \min(8, p)\} \\
U_2 &= \{x \,|\, \max(p{+}1, 2) \leq x \leq 8\} \\
U_3 &= \{x \,|\, 9 \leq x \leq p\}
\end{aligned}
$$

Note that a statement executes either at every point or not at all in U_i. T_1 executes in $U_1 \dot\cup U_2$ and T_2 executes in $U_1 \dot\cup U_3$. In addition, the sets U_i are convex, which implies that each set can be enumerated by a single `for` loop.

This scheme generalises to the case of n-dimensional polytopes as index sets, i.e., the index sets of the statements can always be represented as a disjoint union of polytopes such that each partition is either a subset of a given index set or disjoint from it and each partition can be enumerated by a single nest of `for` loops. The mathematical reason is that intersection and difference of polytopes can, again, be represented by (a union of) polytopes, and polytopes are convex sets. In the generalisation that we are pursuing, this is not true. Index sets with arbitrary multivariate polynomial bounds can, in general, not be represented as a union of convex sets.

2.2 Non-linearity and Non-convexity

Let $D = \{(x, y) \mid 1 \leq x \leq 7, 1 \leq y \leq 9, (y-4)^2 + 12 - 3x \geq 0\}$ be a non-convex index set. D is depicted in Figure 2(a). It is non-convex due to the parabolic piece of the border. Code generators for the polyhedron model treat non-convexities arising from differences of polytopes by representing the domain as a finite union of convex domains, but this is not possible here. D cannot be represented as a finite union of convex sets. Instead, the code generation has to handle non-convexity directly in the general code generation procedure. All loop bounds that are needed to enumerate the domain correctly are roots of (multivariate) polynomials. For example, the roots of $(y-4)^2 + 12 - 3x$, namely $4 \pm \sqrt{3x - 12}$, are bounds of respective inner loops in the code shown in Figure 2(b). Our main result (cf. Theorem 13) states that the needed polynomials and their roots can be computed, if the index sets are described by polynomial inequalities.



(a) depiction

```
for (x=1; x<=4; x++)
  for (y=1; y<=9; y++)
    T1(x,y);
for (x=5; x<=7; x++) {
  for (y=1; y<=⌊4−√(3x − 12)⌋; y++)
    T1(x,y);
  for (y=⌈4+√(3x − 12)⌉; y<=9; y++)
    T1(x,y);
}
```
(b) generated code to scan the domain

Fig. 2. Non-convex domain $\{(x, y) \mid 1 \leq x \leq 7, 1 \leq y \leq 9, (y-4)^2 + 12 - 3x \geq 0\}$

We are aware of three frequent sources of non-linearities:

1. The source program contains non-linear loop bounds. This is the case, for example, in the sieve of Eratosthenes (cf. Section 5.2). The outermost loop has a non-linear bound and can be written as a `for` loop in C-like languages as `for (i=2; i*i<=n; i++)`.
2. The source program has non-constant strides. Before transformations are applied to the program model, the loop strides are normalised to unit strides, such that every integral point in the iteration domain represents an execution of the loop body. E.g., the loop `for (j=0; j<=n; j+=i)` is normalised to `for (k=0; k*i<=n; k++)`, replacing `j` by `k*i` in the loop body. Normalisation is a necessary step in automatic loop program transformation, since code generators usually generate code which scans every integral point in the index sets (cf. the definition of the code generation problem, Definition 5).
3. A non-linear transformation can be applied to the program. For example, it has been shown that non-linear schedules can substantially improve the performance of solving affine recurrence equations over linear schedules. An example is presented in Section 5.2.

All these cases could not be handled by a general procedure so far, since no code could be generated in the presence of non-linearities. We illustrate here the feasibility of code generation for non-linear, non-convex domains, although the efficiency of the code generation procedure has to be improved to be applicable to bigger examples (cf. Section 5).

3 Related Work

The problem of generating code from polyhedral descriptions has been studied for about two decades. Early work concentrated on code generation for a single statement [Iri88]. After seminal steps in this area [AI91], solutions were developed successively for the case with several statements and unions of polyhedra as index sets [CF93,Wet95,KPR95,QRW00,Bas04].

Code generation for a single statement has been solved for more general cases. In our own previous work [GGL04], we have shown that code can be generated for a polyhedral index set which may depend on non-linear parameters (i.e., the inequalities describing the index set may contain products between a variable and polynomials in the parameters) using quantifier elimination in the reals. Recently, an efficient method for generating code for a tiled index sets of one statement with parametric parallelepiped tiles has been presented [RKRS07].

Our definition of the code generation problem (Definition 5) requires the dimensionality of the domains to agree and does not mention so-called scattering functions (i.e., the index sets to enumerate are given as affine images of polytopes) as are supported by CLooG [Bas04], for example. But this is no principal restriction, because scattering functions (even non-invertible ones) and variations in dimensionality can be encoded in the general definition – possibly losing efficiency both in the generation of the code and in the execution of the

generated code. Improving the algorithm for such special cases is on our future agenda.

4 Code Generation for Semi-Algebraic Index Sets

Having seen some introductory examples, let us now state precisely for which index sets we can generate code and that the generated code is efficient in the sense that it does not enumerate an integer superset of the given domains. We start by giving the definitions we need for our main theorem and the algorithm.

4.1 Definitions

We start by defining the code generation problem and its prerequisites.

Definition 1. The inputs for the code generator depend on some *structural parameters* (simply *parameters* for short) $\boldsymbol{p} = (p_1, \ldots, p_k) \in \mathbb{Z}$ ($k \in \mathbb{N}$), i.e., the input (and hence also the output) of the code generator is parametrised in $\boldsymbol{p}$ and the generated code must work for all choices of $\boldsymbol{p}$. The possible values for $\boldsymbol{p}$ are usually restricted to $\boldsymbol{p} \in C \subseteq \mathbb{Z}^k$, the *context* of the problem.

Definition 2. A *statement* $T(\boldsymbol{x})$ is a piece of code (in a given programming language) which depends on a number of variables $\boldsymbol{x} = (x_1, \ldots, x_n)$ ($n \in \mathbb{N}$).

We need not specify statements $T(x)$ more concretely, because we are only concerned with the generation of code for scanning the index sets of the statements, i.e., *loops*, and the statements themselves have no influence on the structure of the generated loops. The loops are determined by the index sets of the statements.

Definition 3. An *index set* (also called *domain*) of a statement $T(\boldsymbol{x})$ is a set $D(\boldsymbol{p}) \subseteq \mathbb{R}^n$ containing all the integral values for $\boldsymbol{x}$ for which $T(\boldsymbol{x})$ shall execute. One instance (i.e., an execution of $T(\boldsymbol{x})$ for a given $\boldsymbol{x}$) is called an *operation*. An index set is called *bounded*, if $D(\boldsymbol{p})$ is bounded for every $\boldsymbol{p} \in C$.

We consider only bounded index sets, because unbounded index sets cannot be enumerated by proper `for` loops with a finite lower and upper bound. Our code generation algorithm works, in principle, also for unbounded index sets; only outputting code with proper `for` loops is, obviously, impossible then.

The order of the operations is defined by the lexicographic order.

Definition 4. Let $\boldsymbol{x} = (x_1, \ldots, x_n)$, $\boldsymbol{y} = (y_1, \ldots, y_n) \in \mathbb{R}^n$. Then $\boldsymbol{x}$ is called *lexicographically less than* $\boldsymbol{y}$, written as $\boldsymbol{x} \prec \boldsymbol{y}$, if and only if there exists an $r \in \{0, \ldots, n-1\}$ such that $(\bigwedge_{i=1}^{r} x_i = y_i) \wedge x_{r+1} < y_{r+1}$ holds.

We can now define the code generation problem.

Definition 5. Let $k, n, m \in \mathbb{N}$, $C \subseteq \mathbb{Z}^k$. Given statements $T_1(\boldsymbol{x}), \ldots, T_m(\boldsymbol{x})$ and domains $D_1(\boldsymbol{p}), \ldots, D_m(\boldsymbol{p}) \subseteq \mathbb{Z}^n$, the problem of *code generation* is to generate a program P which, for any given $\boldsymbol{p} \in C$, executes all (and no more) operations $T_i(\boldsymbol{x})$ with $\boldsymbol{x} \in D_i(\boldsymbol{p}) \cap \mathbb{Z}^n$ for $1 \leq i \leq m$, such that $T_i(\boldsymbol{x})$ is executed before $T_j(\boldsymbol{y})$ if $\boldsymbol{x} \prec \boldsymbol{y}$ for $1 \leq j \leq m$, $\boldsymbol{y} \in D_j(\boldsymbol{p}) \cap \mathbb{Z}^n$.

4.2 Code Generation as Cylindrical Decomposition

A program solving the code generation problem has to enumerate the points of the domains of the statements in lexicographic order. This implies that the outermost dimension is enumerated by one or a sequence of several loops and, inside every such loop, the next dimension is enumerated in dependence of the outer dimension, etc. The concept of a loop nest that scans a union of domains lexicographically is captured by the following definition.

Definition 6. A loop nest is called *cylindrical* for $(x_1, \ldots, x_n)$ in context C, if $n = 0$ and it is the empty loop nest (i.e., it consists only of a loop body), or $n \geq 1$ and it is a sequence of $r \in \mathbb{N}$ loops in x_1

```
for (x1 = l1(p); x1 ≤ u1(p); x1++)
  P1;
...
for (x1 = lr(p); x1 ≤ ur(p); x1++)
  Pr;
```

such that all l_i and u_i are continuous functions in $\boldsymbol{p}$, $l_i(\boldsymbol{p}) < u_i(\boldsymbol{p})$ for every $\boldsymbol{p} \in C$, $1 \leq i \leq r$ and $u_i(\boldsymbol{p}) < l_{i+1}(\boldsymbol{p})$ for every $\boldsymbol{p} \in C$, $1 \leq i \leq r-1$ and for every $1 \leq i \leq r$, P_i is cylindrical for $(x_2, \ldots, x_n)$ in context $C \times [l_i(\boldsymbol{p}), u_i(\boldsymbol{p})]$ (note that x_1 becomes a parameter in the subprograms $P_1, \ldots, P_r$).

We call this kind of loop nest *cylindrical*, because the bounds of the loops define a cylindrical decomposition of $\mathbb{R}^n$. We continue by giving the definition of cylindrical decomposition.

Definition 7. A non-empty connected subset of $\mathbb{R}^n$ ($n \in \mathbb{N}$) is called a *region*. For a region R, we define the *cylinder over* R, written as $Z(R)$, as $R \times \mathbb{R}$.

The cylinder over $\mathbb{R}^0 = \{()\}$ is $\mathbb{R}^0 \times \mathbb{R} = \mathbb{R}$.

Definition 8. Let R be a region of $\mathbb{R}^n$. An *f-section of $Z(R)$* is the set

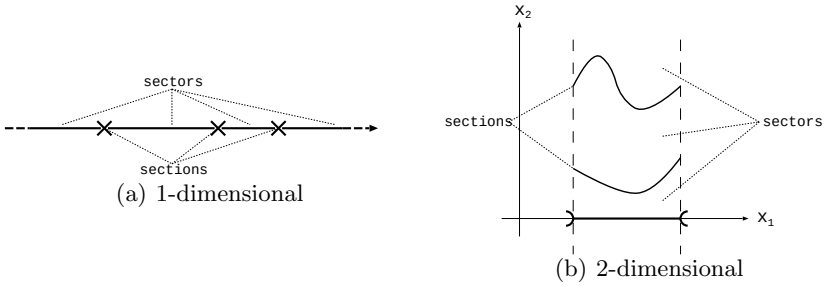
$$\Big\{ \big(a, f(a)\big) \;\Big|\; a \in R \Big\}$$

for a continuous function $f : R \to \mathbb{R}$. An *(f_1, f_2)-sector of $Z(R)$* is the set

$$\Big\{ \big(a, b\big) \;\Big|\; a \in R, b \in \mathbb{R}, f_1(a) < b < f_2(a) \Big\}$$

where $f_1 = -\infty$ or $f_1 : R \to \mathbb{R}$ is continuous, and $f_2 = \infty$ or $f_2 : R \to \mathbb{R}$ is continuous, and $f_1(x) < f_2(x)$ for every $x \in R$.

Obviously, sections and sectors are regions. Figure 3 shows some sections and sectors of $\mathbb{R}^1$ in (a), and some sections and sectors of a cylinder over an interval in $\mathbb{R}^2$ in (b). The sections and sectors shown in Figure 3 also form stacks, as defined by the following definition.



(a) 1-dimensional

(b) 2-dimensional

Fig. 3. Sections and sectors

Definition 9. Let $X \subseteq \mathbb{R}^n$. A *decomposition* of X is a finite collection of pairwise disjoint regions whose union is X. Let R be a region, $r \in \mathbb{N}$, and $f_1, \ldots, f_r : R \to \mathbb{R}$ be continuous functions with $f_1(x) < f_2(x) < \cdots < f_r(x)$ for every $x \in R$. Then $(f_1, \ldots, f_r)$ defines a decomposition of $Z(R)$ consisting of the sets

- f_i-sections of $Z(R)$ for $1 \leq i \leq r$,
- (f_i, f_{i+1})-sectors of $Z(R)$ for $1 \leq i < r$,
- the $(-\infty, f_1)$-sector of $Z(R)$,
- the (f_r, ∞)-sector of $Z(R)$.

Such a decomposition is called a *stack over* R defined by $(f_1, \ldots, f_r)$. In the case of $r = 0$, the decomposition consists only of the $(-\infty, \infty)$-sector of $Z(R)$, i.e., the stack consists of the single region $Z(R)$.

If the decomposition into stacks is made at every level, i.e., also for inner dimensions, the decomposition obtained is called cylindrical.

Definition 10. A decomposition D of $\mathbb{R}^n$ is called *cylindrical*, if either

(1) $n = 1$ and D is a stack over $\mathbb{R}^0$, or
(2) $n > 1$ and there is a cylindrical decomposition D' of $\mathbb{R}^{n-1}$ such that, for each region R of D', D contains a stack over R.

The lower and upper bounds $(l_1, u_1, \ldots, l_r, u_r)$ of the loops in Definition 6 define a decomposition of $\mathbb{R}$ and, since the requirements for the inner dimensions are the same, the loops define a cylindrical decomposition of $\mathbb{R}^n$. Note that the loops do not scan every region of the decomposition; for example, the region between the upper bound of a loop and the lower bound of its successor is not scanned.

The code generation problem can now be reduced to computing a cylindrical decomposition of $\mathbb{R}^n$ such that every region in the decomposition is, for every index set, either a subset of this index set or disjoint from it. The code generated by code generators for the polyhedron model like CLooG is, in fact, cylindrical. The solution to generating code for the more general case with non-linear index set bounds relies on an algorithm for computing a cylindrical decomposition of

$\mathbb{R}^n$. The algorithm cannot, in general, compute a cylindrical decomposition with a minimal set of regions, because it computes a cylindrical *algebraic* decomposition which has the additional restriction that the functions f_i (which define the sections of a stack) are polynomial expressions. Polyhedral code generators generate loop bounds which are maxima or minima of linear expressions for the lower or upper bound of a polytope. Since the minimum or maximum of two expressions is, in general, not a polynomial, the code generated by our algorithm for polyhedral input is longer than the code generated by polyhedral techniques (for an example, cf. Section 5.1). Sections 4.5 and 6 discuss techniques for reducing the amount of generated code by combing two or more loops into a single loop by exploiting properties of the generated bounds.

Definition 11. A set $S \subseteq \mathbb{R}^n$ is called *semi-algebraic*, if it can be defined by a quantifier-free formula of polynomial equalities and inequalities.

A decomposition of $\mathbb{R}^n$ is called *algebraic* if each of its regions is a semi-algebraic set. A *cylindrical algebraic decomposition* (CAD) is a decomposition which is both cylindrical and algebraic.

4.3 The Efficiency of a Solution

The definition of code generation (Definition 5) does not take efficiency into account. For example, all examples shown in Figure 1 are solutions to the code generation problem stated there, but obviously the code in Figure 1(a) is less efficient because it evaluates two conditions in the `if` statements in every iteration of the loop. A different source of inefficiency are inner loops which have empty iteration sets for some iterations of the outer loops, i.e., the outer loops enumerate points (in the outer coordinates) which do not belong to integer solutions of the index sets. Very inefficient code can be generated for a large class of code generation problems. To obtain some solution, it is sufficient to enumerate some arbitrarily large but finite superset of the union of all index sets $\bigcup D_i(\boldsymbol{p})$ and to test for every enumerated point $\boldsymbol{x}$ whether $\boldsymbol{x} \in D_i(\boldsymbol{p})$ and execute $T_i(\boldsymbol{x})$ if that is the case (like in Figure 1(a)).

To capture the notion of an efficient program, which does not enumerate integral values from a superset of the domains, we introduce the concept of a proper scan in $\mathbb{R}$ and $\mathbb{Z}$ for cylindrical loop nests. The idea is to call a loop nest a proper $\mathbb{Z}$-scan, if, for some values enumerated by r outer loops of a loop nest, we can be sure that there is an integral point in the domain that matches the enumerated values and, hence, the body of the loop nest will be executed for the choice of the outer r dimensions. A less strict property, called a proper $\mathbb{R}$-scan, is that there exists any value (maybe with real values for the inner dimensions) that matches the outer dimensions.

Definition 12. Let P be a cylindrical loop nest solving the code generation problem for some domains and statements *without* `if` *statements inside its loops*. P is said to perform a *proper* $\mathbb{R}$*-scan* or a *proper* $\mathbb{Z}$*-scan*, respectively, of the domains if for every loop nest

```
for (x_1 = l_1(p); x_1 ≤ u_1(p); x_1++)
  ...
      for (x_n = l_n(p, x_1, ..., x_{n-1}); x_n ≤ u_n(p, x_1, ..., x_{n-1}); x_n++)
        T_j(x_1, ..., x_n);
```

surrounding an occurrence of T_j in P, for every $0 \leq r \leq n-1$, and $X = \mathbb{R}$ or $X = \mathbb{Z}$, respectively, the following condition holds:

$$\boldsymbol{p} \in C \wedge \bigwedge_{i=1}^{r} \big(x_i \in \mathbb{Z} \wedge l_i(\boldsymbol{p}, x_1, \ldots, x_{i-1}) \leq x_i \leq u_i(\boldsymbol{p}, x_1, \ldots, x_{i-1})\big)$$
$$\Longrightarrow \exists x_{r+1}, \ldots, x_n \in X : (x_1, \ldots, x_n) \in D_j(\boldsymbol{p})$$

Note that, since the program is cylindrical, it is impossible for two distinct parts of the program to enumerate the same values $(x_1, \ldots, x_r)$ for the outer dimensions (this would violate the cylindricality of the program).

It is desirable to have programs which perform proper $\mathbb{Z}$-scans, because this guarantees that every iteration an outer loop performs will lead to at least one execution of the body. In contrast, the proper $\mathbb{R}$-scan property only guarantees that the program does not enumerate integer points from a superset of the domains of the statements, but outer loops may perform superfluous iterations which have empty inner loops. For example, the program

```
for (x=0; x<=p; x++)
    for (y=ceil(x/(2*p)); y<=1-floor(x/(2*p)); y++)
        T(x,y);
```

performs a proper $\mathbb{R}$-scan of the domain

$$D(p) = \{(x, y) \,|\, 0 \leq x \leq p, 2x \leq 2py + x \leq 2p\}$$

for $p \geq 0$, because for every $0 \leq x \leq p$ there exists $y \in \mathbb{R}$ such that $2x \leq 2py + x \leq 2p$, for example, $y = 0.5$. Only the iteration $x = 0$ has a non-empty y loop with $y \in \{0, 1\}$ but, for $x \geq 1$, the real y-values of D lie only in the open interval between 0 and 1. Cases like these do occur in practice when two index variables are linked by an equation or a system of equations which has integer "holes" in its solution set, as in the Eratosthenes example (cf. Section 5.2). On the other hand, the problem is rarely caused by inequalities, i.e., inequalities with real solutions but without integral solutions are infrequent in practice.

4.4 Code Generation for Semi-algebraic Sets

Theorem 13. *Let C and D_i be as in Definition 5 where the D_i are bounded index sets. The code generation problem can be solved with a cylindrical loop nest which performs a proper $\mathbb{R}$-scan of the domains if the extended index sets $\widehat{D}_i = \{(\boldsymbol{p}, \boldsymbol{x}) \,|\, \boldsymbol{p} \in C, \boldsymbol{x} \in D_i(\boldsymbol{p})\}$ are semi-algebraic. A solution can be computed algorithmically from the defining formulas with polynomial (in)equalities for $D_1(\boldsymbol{p}), \ldots, D_m(\boldsymbol{p})$ and C from a sign-invariant cylindrical algebraic decomposition of $\mathbb{R}^n$ for the formulas defining the $\widehat{D}_i$ and the algorithm shown in Figure 4. The generated code performs a proper $\mathbb{R}$-scan of the domains.*

Proof. Let Ψ be the set of all polynomials in the formulas defining the $\widehat{D}_i$. A sign-invariant cylindrical algebraic decomposition of $\mathbb{R}^{k+n}$ for Ψ can be computed by well-known algorithms [ACM98]. This yields decompositions R_j of $\mathbb{R}^j$ for $1 \leq j \leq k+n$ with two important properties:

(1) For every $1 \leq i \leq m$ and $S \in R_{k+n}$, either $S \subseteq \widehat{D}_i$ or $S \cap \widehat{D}_i = \varnothing$. This is due to the sign invariance of the decomposition. All $\psi \in \Psi$ have constant sign on S and, therefore, the truth value of the formula defining $\widehat{D}_i$ is constant on S.

(2) Due to the cylindrical nature of the decomposition, the regions of a stack are ordered lexicographically. Let $\boldsymbol{w} \in \mathbb{R}^{j-1}$ and regions $S_1, S_2 \in R_j$, $S_1 \neq S_2$, and define the sets of x_j-coordinates of points "above" $\boldsymbol{w}$ in S_1 and S_2 by

$$A_1 = \{x_j \in \mathbb{R} \,|\, (\boldsymbol{w}, x_j) \in S_1\}$$
$$A_2 = \{x_j \in \mathbb{R} \,|\, (\boldsymbol{w}, x_j) \in S_2\}.$$

Then either $A_1 \cap A_2 = \varnothing$, or A_1 and A_2 are ordered (i.e., all points in A_1 are either less than or greater than all points in A_2).

Code is generated as specified by the algorithm in Figure 4. It is a recursive procedure which, in each step of the recursion, generates the loops for the next dimension. The main code generation function is `code_gen(`S`,`t`,`d`)`, where d is the number of the current dimension, S a sector or section from the decomposition of $\mathbb{R}^{d-1}$ and $t \in S$, a so-called test point which is used (in the base case of the recursion) to test whether a domain $\widehat{D}_i$ is a subset of the current region (for which loops are generated). These properties of S, d and t are an invariant of the recursion.

Code generation starts with $S = \{()\}$, $t = ()$, $d = 1$. Note that S is the only sector of a decomposition of $\mathbb{R}^0$ ($d - 1 = 0$), and $t \in S$ holds. In each step of the recursion, code is generated for the f_i-sections and (f_i, f_{i+1})-sectors over S in the functions `section_code` and `sector_code`, respectively. Note that the code is composed such that the lexicographic ordering is respected due to property (2) of the decompositions. The code generated is different depending on whether the current dimension d is a parameter dimension ($1 \leq d \leq k$) or an index set variable ($k+1 \leq d \leq k+n$). For a parameter dimension, a conditional statement is generated that checks that the actual value of the parameter satisfies the constraint imposed by the current section or sector. For an index set dimension, the code generated is a loop that enumerates the integral points between the two sections of a sector (if code for a sector is generated), or a loop with exactly one iterations if and only if the section has an integral value (for the given values of the outer dimensions). Note that in the course of the recursion a test point $t \in S$ is constructed. The function `rational_between(`a`,`b`)` is used to compute a rational point between a and b.

The base case of the recursion is $d = n + k + 1$, in which no more loops are generated and the loop body is written. If $t \in \widehat{D}_i$ for a domain $\widehat{D}_i$, then $S \subseteq \widehat{D}_i$; otherwise $S \cap \widehat{D}_i = \varnothing$ (due to property (1) of the decomposition). That is, the body of the loop nest generated has to contain exactly the statements T_i for which $t \in \widehat{D}_i$ holds. □

Note that our algorithm generates code which has exactly one lower and one upper bound in each loop generated.

Let us illustrate the relation between the index set, a cylindrical algebraic decomposition and the code generated for the example shown in Figure 5. The roots (i.e., sections) for x defining R_1 are $1, 4, 7$, i.e., we have to handle the cases $x = 1$, $1 < x < 4$, $x = 4$, $4 < x < 7$, and $x = 7$. For $1 \leq x < 4$, the decompositions of $\{1\} \times \mathbb{R}$ and $]1, 4[\times\mathbb{R}$ are given by the roots 1 and 9 for y. For $x = 4$, the roots for y are given by 1, 4, and 9. For $4 < x < 7$, the roots for y are 1, $4 - \sqrt{3x - 12}$, $4 + \sqrt{3x + 12}$, and 9 and, for $x = 7$, we have the roots 1, $4 + \sqrt{3x + 12}$, and 9. The sections and test points (i.e., points which lie on each of the sections and in each of the sectors and are used to test whether the respective region is part of a domain) for the domain are shown in Figure 5(b). The roots correspond directly to the loop bounds of the code in Figure 5(c).

4.5 Improving the Code

The example in Figure 5 shows that the code generated by the algorithm is quite lengthy. The key observation to reducing the code size is that the loops for neighbouring regions (sections and sectors) in a stack often contain the same inner loops and loop bodies. For example, the loops on x for $1 \leq x \leq 1$ and $1 < x < 4$ contain syntactically identical code. These two loops on x could be combined straight away into one loop for $1 \leq x < 4$. Combing this loop with the loop for $4 \leq x \leq 4$ is not possible at first, since at $x = 4$ there is an additional case distinction for $y = 4$ (the apex of the parabolic piece of the border). But, of course, if we combine the loops on y inside the loops on x first, the first three loops an x only contain a loop on y for $1 \leq y \leq 9$ in their respective bodies, and $1 \leq x \leq 4$ can be scanned by a single loop on x (with a single loop on y inside).

The situation is more complex with the two loops on x for $4 < x \leq 7$. Combining the loops on y inside the x loops yields the following two loop nests:

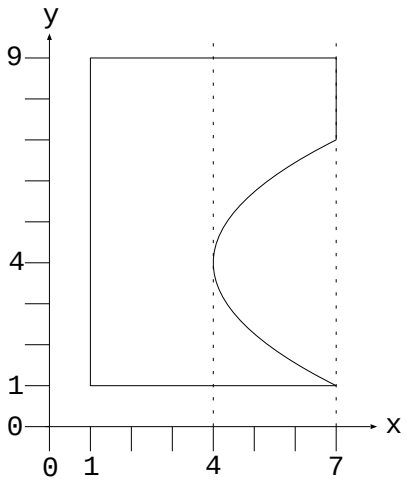
```
for (x=4+1; x<=7-1; x++) {
  for (y=1; y<=⌊4−√3x−12⌋; y++)
    T1(x,y);
  for (y=⌈4+√3x−12⌉; y<=9; y++)
    T1(x,y);
}
```

```
for (x=7; x<=7; x++) {
  for (y=1; y<=1; y++)
    T1(x,y);
  for (y=⌈4+√3x−12⌉; y<=9; y++)
    T1(x,y);
}
```

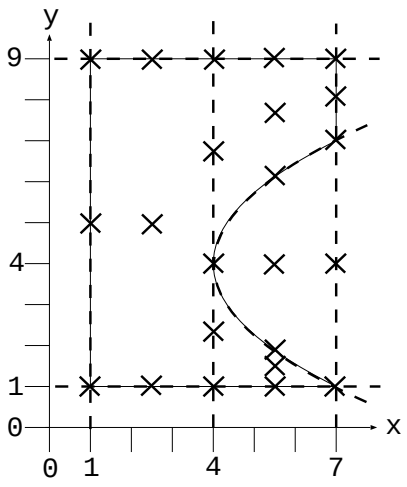
The obvious problem which prevents us from combining these two loops is that the upper bounds of the respective first loops on y are different, namely $\lfloor 4-\sqrt{3x-12} \rfloor$ and 1. But the values of the bounds are the same for $x = 7$, namely 1. This happens since both expressions are roots of polynomials which define the iteration domain and an implementation of a CAD algorithm naturally selects the root with the lower degree if two roots coincide in a stack (here: the stack over $x = 7$). This problem occurs every time when roots of different polynomials cross. We have to note that "crossing polynomials" excludes polynomials which vanish on an entire cylinder (i.e., which are called *identically zero* in a term used in the CAD literature). For example, the polynomial $x - 1$, whose root $x = 1$

```
// Generate loops from a cylindrical decomposition
//    n: dimensionality of the index sets
//    k: number of parameters
//    T_1,...,T_m: statements
//    D̂_1,...,D̂_m: extended index sets of the statements
// Parameters of code_gen:
//    S: generate loops for the cylinder over this section or sector
//    t: a test point from S
//    d: level of the loops to be generated
code_gen(S,t,d):
  code="";
  if d = n + k + 1 then
    for i:=1 to m
      if t ∈ D̂_i then
        code += "T_i(x_1, ..., x_n);";
      end if
    end for
  else
    let f_1,...,f_r be the sections defining the stack over S
    for i:=1 to r − 1
      code += section_code(S,f_i,t,d);
      code += sector_code(S,f_i,f_{i+1},t,d);
    end for
    code += section_code(S,f_r,t,d);
  end if
  return code;
section_code(S,f,t,d):
  inner=code_gen(section(S,f),(t,f(t)),d + 1);
  if inner≠"" then
    if d ≤ k then
      return "if p_d = f {" + inner + "}";
    else
      return "for x_{d−k}=ceil(f) to floor(f) {" + inner + "}";
    end if
  end if
  return "";
sector_code(S,f,g,t,d):
  t'=(t,rational_between(f(t),g(t)));
  inner=code_gen(sector(S,f,g),t',d + 1);
  if inner≠"" then
    if d ≤ k then
      return "if p_d > f and p_d < g {" + inner + "}";
    else
      return "for x_{d−k}=floor(f+1) to ceil(g-1) {" + inner + "}";
    end if
  end if
  return "";
```

Fig. 4. Code generation algorithm



(a) depiction



(b) CAD sections and test points

```
for (x=1; x<=1; x++) {
  for (y=1; y<=1; y++) T1(x,y);
  for (y=1+1; y<=9-1; y++) T1(x,y);
  for (y=9; y<=9; y++) T1(x,y);
}
for (x=1+1; x<=4-1; x++) {
  for (y=1; y<=1; y++) T1(x,y);
  for (y=1+1; y<=9-1; y++) T1(x,y);
  for (y=9; y<=9; y++) T1(x,y);
}
for (x=4; x<=4; x++) {
  for (y=1; y<=1; y++) T1(x,y);
  for (y=1+1; y<=4-1; y++) T1(x,y);
  for (y=4; y<=4; y++) T1(x,y);
  for (y=4+1; y<=9-1; y++) T1(x,y);
  for (y=9; y<=9; y++) T1(x,y);
}
for (x=4+1; x<=7-1; x++) {
  for (y=1; y<=1; y++)
    T1(x,y);
  for (y=1+1; y<=⌈4−√(3x−12)⌉-1; y++)
    T1(x,y);
  for (y=⌈4−√(3x−12)⌉; y<=⌊4−√(3x−12)⌋; y++)
    T1(x,y);
  for (y=⌈4+√(3x−12)⌉; y<=⌊4+√(3x−12)⌋; y++)
    T1(x,y);
  for (y=⌊4+√(3x−12)⌋+1; y<=9-1; y++)
    T1(x,y);
  for (y=9; y<=9; y++)
    T1(x,y);
}
for (x=7; x<=7; x++) {
  for (y=1; y<=1; y++)
    T1(x,y);
  for (y=⌈4+√(3x−12)⌉; y<=⌊4+√(3x−12)⌋; y++)
    T1(x,y);
  for (y=⌊4+√(3x−12)⌋+1; y<=9-1; y++)
    T1(x,y);
  for (y=9; y<=9; y++)
    T1(x,y);
}
```

(c) generated code to scan the domain

Fig. 5. Code for domain $\{(x, y) \mid 1 \leq x \leq 7, 1 \leq y \leq 9, (y-4)^2+12-3x \geq 0\}$ according to the algorithm in Figure 4

crosses the roots $y = 1$ and $y = 7$ (and which vanishes on $\{1\} \times \mathbb{R}$), does not inhibit the combining of the loops for $x = 1$ and $1 < x < 4$. To be able to combine loops in the general case that several polynomials have the same root in a stack, the CAD procedure must retain all the roots which can be used as bounds (i.e., which define the sections), and the code generation procedure selects those which achieve a maximum of combining possibilities. So the intermediate code (after combining the loops on y) is

```
for (x=4+1; x<=7-1; x++) {
  for (y=1; y<=⌊4−√(3x−12)⌋; y++)
    T1(x,y);
  for (y=⌈4+√(3x−12)⌉; y<=9; y++)
    T1(x,y);
}
for (x=7; x<=7; x++) {
  for (y=OneOf{1,⌈4−√(3x−12)⌉}; y<=OneOf{1,⌊4−√(3x−12)⌋}; y++)
    T1(x,y);
  for (y=⌈4+√(3x−12)⌉; y<=9; y++)
    T1(x,y);
}
```

where `OneOf` means that the code output procedure is free to choose either of the given roots. Combining loops as much as possible yields the desired simple code which has already been shown in Figure 2(b).

5 Examples

In this section we give several examples of the code generated by our procedure outlined in Section 4. We start by comparing the code generated to that generated by polyhedral code generators for polyhedral input in Section 5.1 before we show some non-polyhedral cases in Section 5.2.

5.1 Comparison with Polyhedral Code Generation

A Triangular Index Set As a first example, consider the triangular domain $D = \{(x, y) \mid y \geq 1, y \leq x, y \leq -x + p\}$ depicted in Figure 6. Polyhedral code generators like CLooG have no difficulty combining parts of index sets bounded by different upper (or lower) bounds, like the bounds $y \leq x$ and $y \leq -x + p$ in the example. CLooG implicitly computed a cylindrical decomposition of $\mathbb{R}^2$ with the two sections $x_1 = 1$ and $x_2 = p$ to decompose $\mathbb{R}$ and the two sections

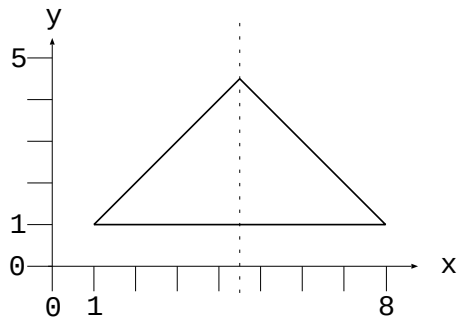
$$y_1(x) = 1$$

and

$$y_2(x) = \begin{cases} x & \text{if } 1 \leq x \leq \frac{p}{2} \\ -x + p & \text{if } \frac{p}{2} < x \leq p \end{cases}$$

to decompose $[1, p] \times \mathbb{R}$. Note that this decomposition is *not* algebraic (cf. Definition 11), because y_2 is not a polynomial (due to its non-smoothness at $x = \frac{p}{2}$).

Therefore, a cylindrical algebraic decomposition must have an additional section at $x_3 = \frac{p}{2}$ and the scanning of the x-dimension in the code generated by our method is divided into two loops for both halves of the triangle. This is a slight optimisation in terms of loop bound evaluation costs, because the polyhedral code has to evaluates two upper bounds for y and take their minimum for every value of x. But, since loop bounds are usually only evaluated rarely compared to the number of executions of the loop body, the polyhedral code is to be considered superior, because it does not duplicate the loop body.



(a) depiction for $p = 9$

```
if (p == 2) {
  for (x=1; x<=1; x++) {
    for (y=1; y<=1; y++)
      T1(x,y);
  }
}
if (2+1 <= p) {
  for (x=1; x<=⌊p/2⌋; x++) {
    for (y=1; y<=x; y++)
      T1(x,y);
  }
  for (x=⌊p/2⌋+1; x<=p-1; x++) {
    for (y=1; y<=-x+p; y++)
      T1(x,y);
  }
}
```

(b) our code

```
for (x=1; x<=p-1; x++) {
  for (y=1; y<=min(-x+p,x); y++)
    T1(x,y);
}
```

(c) code generated by CLooG

Fig. 6. Example: domain $D = \{(x, y) \,|\, y \geq 1, y \leq x, y \leq -x + p\}$

Quilleré's example As a second example, we consider the problem given (and solved) by Quilleré et al. [QRW00] with

$$D_1(m, n) = \{(x, y) \,|\, 1 \leq x \leq n, 1 \leq y \leq m\}$$
$$D_2(m, n) = \{(x, y) \,|\, x = y, 3 \leq x \leq n\}$$

shown in Figure 7. The codes were generated assuming that $m, n \geq 4$. Again, the polyhedral code is shorter. The reason here is that our algorithm only generates loops which are non-empty in the reals (a proper $\mathbb{R}$-scan does not guarantee non-emptiness in the integers, though), but CLooG's code can contain empty

loops for certain parameter constellations. For example, the last loop on x in CLooG's code is empty for $m \geq n$.

5.2 Non-polyhedral Examples

A Non-linear Schedule In automatic loop parallelisation in the polyhedron model, each operation of a given program is assigned a time to execute (the so-called schedule) and a processor to execute on (the so-called allocation). Both schedules and allocations are usually chosen as (stepwise) affine functions, because polyhedral code generation requires them to be affine functions. It has been argued [AZ00] that non-linear schedules found by quadratic programming can provide substantially shorter overall execution times. An example used by Achtziger et al. [AZ00, Example 2.2] is a recurrence equation with the index set $D(n)$ defined by

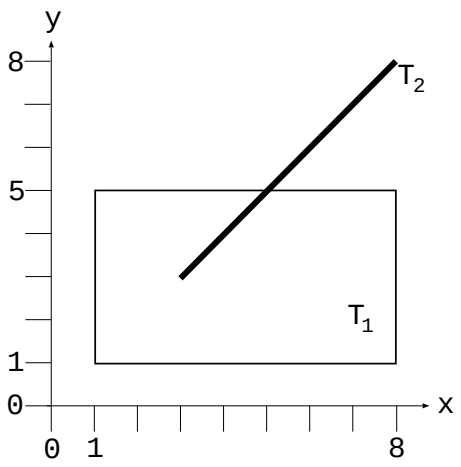
$$\begin{aligned} 2 &\leq x \leq n \\ 4 &\leq y \leq n \\ n - x &\leq y \end{aligned}$$

for $n \geq 7$ and dependences

$$\begin{aligned} (x, y-1) &\to (x, y) \\ (x-1, x) &\to (x, y). \end{aligned}$$

Achtziger et al. compute $t = (n-3) \cdot x + y$ as a non-linear, nearly optimal schedule. To generate code for a parallel execution, we use the equation defining the schedule and the original index set to define the domain of the statement and generate code for the variable ordering (t, x, y). We can compare the code generated by our algorithm with code generated by applying (a generalised version of) Fourier-Motzkin elimination [GGL04]. Fourier-Motzkin elimination can be used to generate code here, since there is only one index set which is, in addition, a conjunction of formulas linear in the variables. Both codes are shown in Figures 8(a) and 8(b). As Fourier-Motzkin elimination (or an equivalent algorithm) is at the heart of polyhedral code generation, the code generated by Fourier-Motzkin elimination is shorter, because it generates several lower and upper bounds for each loop, if required. Our CAD based code is longer, but incurs less overhead in the loop bounds.

Sieve of Eratosthenes The sieve of Eratosthenes is a well-known algorithm for computing the prime numbers in $2, \ldots, n$. A related, but slightly different, problem is to compute the number of factorisations of the numbers $2, \ldots, n$ into two factors (excluding 1) not considering the ordering of the factors. For example, 96 can be factored as $2 \cdot 48$, $4 \cdot 24$, $6 \cdot 16$, and $8 \cdot 12$. The sequential code for computing the number of factorisations into two factors is given in Figure 9(a). At the end of the program, $A[j]$ will contain the number of factorisations



(a) depiction for $m = 5$, $n = 8$

```
for (x=1; x<=2; x++) {
  for (y=1; y<=m; y++) T1(x,y);
}
for (x=3; x<=min(m-1,n); x++) {
  for (y=1; y<=x-1; y++) T1(x,y);
  T1(x,x); T2(x,x);
  for (y=x+1; y<=m; y++) T1(x,y);
}
if (m <= n) {
  for (y=1; y<=m-1; y++) T1(m,y);
  T1(m,m); T2(m,m);
}
for (x=m+1; x<=n; x++) {
  for (y=1; y<=m; y++) T1(x,y);
  T2(x,x);
}
```

(b) CLooG code

```
if (n <= m-1) {
  for (x=1; x<=3-1; x++) { for (y=1; y<=m; y++) T1(x,y); }
  for (x=3; x<=n; x++) {
    for (y=1; y<=x-1; y++) T1(x,y);
    for (y=x; y<=x; y++) { T1(x,y); T2(x,y); }
    for (y=x+1; y<=m; y++) T1(x,y);   }
} else if (n == m) {
  for (x=1; x<=3-1; x++) { for (y=1; y<=OneOf(m,n); y++) T1(x,y); }
  for (x=3; x<=OneOf(n,m)-1; x++) {
    for (y=1; y<=x-1; y++) T1(x,y);
    for (y=x; y<=x; y++) { T1(x,y); T2(x,y); }
    for (y=x+1; y<=OneOf(m,n); y++) T1(x,y);   }
  for (x=OneOf(n,m); x<=OneOf(n,m); x++) {
    for (y=1; y<=OneOf(m,x,n)-1; y++) T1(x,y);
    for (y=OneOf(m,x,n); y<=OneOf(m,x,n); y++) { T1(x,y); T2(x,y); }   }
} else if (m+1 <= n) {
  for (x=1; x<=3-1; x++) { for (y=1; y<=m; y++) T1(x,y); }
  for (x=3; x<=m-1; x++) {
    for (y=1; y<=x-1; y++) T1(x,y);
    for (y=x; y<=x; y++) { T1(x,y); T2(x,y); }
    for (y=x+1; y<=m; y++) T1(x,y);   }
  for (x=m; x<=m; x++) {
    for (y=1; y<=OneOf(m,x)-1; y++) T1(x,y);
    for (y=OneOf(m,x); y<=OneOf(m,x); y++) { T1(x,y); T2(x,y); }   }
  for (x=m+1; x<=n; x++) {
    for (y=1; y<=m; y++) T1(x,y);
    for (y=x; y<=x; y++) T2(x,y);   }
}
```

(c) our code

Fig. 7. Example from [QRW00] (under the assumption $m, n \geq 4$)

```
for (t=3*n-8+1; t<=3*n-6; t++)
  parfor (x=2; x<=⌊(t-n)/(n-4)⌋; x++)
    for (y=(-n+3)*x+t; y<=(-n+3)*x+t; y++)
      T1(x,y);
for (t=3*n-6+1; t<=n*n-7*n+16; t++)
  parfor (x=⌈(t-n)/(n-3)⌉; x<=⌊(t-n)/(n-4)⌋; x++)
    for (y=(-n+3)*x+t; y<=(-n+3)*x+t; y++)
      T1(x,y);
for (t=n*n-7*n+16+1; t<=n*n-3*n+4; t++)
  parfor (x=⌈(t-n)/(n-3)⌉; x<=⌊(t-4)/(n-3)⌋; x++)
    for (y=(-n+3)*x+t; y<=(-n+3)*x+t; y++)
      T1(x,y);
for (t=n*n-3*n+4+1; t<=n*n-2*n; t++)
  parfor (x=⌈(t-n)/(n-3)⌉; x<=n; x++)
    for (y=(-n+3)*x+t; y<=(-n+3)*x+t; y++)
      T1(x,y);
```

(a) code generated by CAD for $n \geq 7$

```
for (t=3*n-8; t<=n*n-2*n; t++)
  parfor (x=max(2,⌈(t-n)/(n-3)⌉); x<=min(min(n,⌊(t-4)/(n-3)⌋),⌊(t-n)/(n-4)⌋); x++)
    for (y=max(max(-4,-x+n),t+(-n+3)*x); y<=min(n,t+(-n+3)*x); y++)
      T1(x,y);
```

(b) code generated by Fourier-Motzkin elimination for $n \geq 7$

Fig. 8. Example from [AZ00] with schedule $t = (n-3) \cdot x + y$

```
for (i=2; i*i<=n; i++) {
  for (j=i*i; j<=n; j+=i)
    A[j]++;
}
```

(a) sequential code

```
for (i=2; i*i<=n; i++) {
  for (k=i; k*i<=n; k++) {
    j=k*i;
    A[j]++;
  }
}
```

(b) normalised sequential code

```
parfor (j=4; j<=n; j++) {
  for (i=2; i<=⌊√j⌋; i++) {
    for (k=⌈j/i⌉; k<=⌊j/i⌋; k++)
      A[j]++;
  }
}
```

(c) parallel code

Fig. 9. Example: computing the number of 2-factorisations

of j (assuming that A is initialised with all zeros). Obviously, the loop on j can be executed in parallel for a given i, since different j iterations access different elements of A. But such a parallel execution requires to setup a parallel execution and synchronise after the j loop for every iteration of i. It is desirable to exchange the loops on i and j in order to make the outer loop the parallel loop. To do so, we first have to normalise the program such that all loops have unit stride. This is achieved by applying the substitution $j := k \cdot i$ on the loop on j. The resulting normalised program is shown in Figure 9(b). Now, code generation for the domain defined by

$$\begin{aligned} 2 \leq i \wedge i^2 \leq n \\ i \leq k \wedge k \cdot i \leq n \\ j = k \cdot i \end{aligned}$$

with the variable ordering (j, i, k) can be applied. This yields the code shown in Figure 9(c). The loop on j, which is now the outermost loop, is marked as parallel. We have to point out that the transformed code is less efficient than the original code because, by exchanging the loops on i and j, it is not guaranteed (by the construction of the loops) that j is a multiple of i. That is why the loop on k, whose only function is to check whether i divides j, has to be present in the code, and it has many iterations which are empty in the integers. So a substantial number of processors is required to achieve a speedup.

6 Future Improvements

Our algorithm produces code which has the properties that it is a proper $\mathbb{R}$-scan and that every loop generated has exactly one lower and one upper bound. Both properties offer room for improvement. Since the theory of integers with addition and multiplication is undecidable, there cannot be a general improvement over proper $\mathbb{R}$-scans. But there are many special cases in practical code generation. Therefore, it is worthwhile to invest in integral non-emptiness tests for common cases, e.g., linear formulas. If a region (section or sector) is bounded by linear formulas only, its integral feasibility can be tested.

The other direction for improvement is to reduce the code size by combining inner loops with different upper and/or lower bounds by using minima and maxima in the bounds. For example, in the triangular index set example (cf. Figure 6), the two loops on x cannot be combined because of the different inner loops on y. But the upper bounds of the loops on y are compatible in the sense that each bound is stricter than the other bound in its respective x-region, i.e., because the implications

$$\begin{aligned} 1 \leq x \leq \frac{p}{2} &\Rightarrow x \leq -x + p \\ \frac{p}{2} < x \leq p &\Rightarrow -x + p \leq x \end{aligned}$$

hold, it could be detected that the loops on y can be combined into the loop nest generated by polyhedral code generation. This is a *semantic* test that goes beyond the syntactic test we perform on the roots defining the sections of two neighbouring stacks when deciding whether the codes for the two stacks can be merged into a single code.

7 Conclusions

Cylindrical algebraic decomposition enables the generation of target loop code for index sets with arbitrary polynomial bounds. We have presented a basic algorithm, which we have implemented as a prototype. The algorithm relies on the computation of a cylindrical algebraic decomposition (CAD) of $\mathbb{R}^n$ (for n-dimensional index sets). The requirement of the decomposition to be algebraic causes the generated code to be quite lengthy. But we have shown that loops can be combined by syntactic reasoning (provided that the CAD procedure emits all polynomials whose roots coincide over a region). Each loop generated has exactly one lower and one upper bound and, when a loop is reached, the lower bound is less than or equal to the upper bound, i.e., the loop is non-empty in the reals, but there is no guarantee of integral non-emptiness. Integral non-emptiness is undecidable in the general case but may be checked for common cases like linear bounds in a future version of the algorithm. Further reduction of the code size is another goal which may be achieved by combining suitable loops and thereby introducing minima and maxima of bounds in the new loop bounds.

At the moment, the practical applicability of the presented algorithm is limited. We are hoping for more research in this area, now that a code generation procedure available.

References

[ACM98] Dennis S. Arnon, George E. Collins, and Scott McCallum. Cylindrical Algebraic Decompositions I: The Basic Alogrithm. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 136–151. Springer-Verlag, 1998.

[AI91] Corinne Ancourt and François Irigoin. Scanning Polyhedra with DO Loops. *Third ACM SIGPLAN Symposium on Priciples & Practice of Parallel Programming*, 26(7):39–50, July 1991.

[AZ00] Wolfgang Achtziger and Karl-Heinz Zimmermann. Finding quadratic schedules for affine recurrence equations via nonsmooth optimization. *J. VLSI Signal Process. Syst.*, 25(3):235–260, 2000.

[Bas04] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[CF93] Jean-François Collard and Paul Feautrier. Automatic generation of data parallel code. In H.J. Sips, editor, *Proc. of the Fourth International Workshop on Compilers for Parallel Computers*, pages 321–332, December 1993.

[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, jul 2004.

[Iri88] François Irigoin. Code generation for the hyperplane method and for loop interchange. Technical Report ENSMP-CAI-88-E102/CAI/I, Ecole Normale Supérieure des Mines de Paris, October 1988.

[KPR95] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *FRONTIERS '95: Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 321–332, Washington, DC, USA, 1995. IEEE Computer Society.

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

[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.

[RKRS07] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. *SIGPLAN Not.*, 42(6):405–414, 2007.

[Wet95] Sabine Wetzel. Automatic code generation in the polyhedron model. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, November 1995. http://www.fmi.uni-passau.de/loopo/doc/wetzel-d.ps.gz.