A Polyhedral Compilation Library with Explicit Disequality Constraints

Sven Verdoolaege
Cerebras Systems
Belgium
sven@cerebras.net

Abstract

Polyhedral libraries usually have no explicit support for disequality constraints, requiring them to be encoded as a disjunction of inequality constraints instead. Every additional disequality constraint then typically leads to a doubling of the size of the representation. This paper shows how the isl library can be extended to support disequality constraints natively, allowing for a significantly reduced computation time due to the more efficient representation.

1 Introduction

Several support libraries have been used and have often even been specifically designed for polyhedral compilation. This includes libraries focused on actual polyhedra, i.e., rational sets bounded by affine constraints, such as `PolyLib` (Wilde 1993) and `PPL` (Bagnara et al. 2008), as well as libraries focused on sets of integer tuples described by Presburger formulas, such as `Omega` (Kelly, Maslov, et al. 1996), `isl` (Verdoolaege 2010), `Omega+` (Chen 2012) and `FPL` (Pitchanathan et al. 2021). All these libraries have an explicit representation for equality constraints, even though this is strictly speaking not required since an equality constraint $f(\mathbf{x}) = 0$ can be represented as a pair of inequality constraints $f(\mathbf{x}) \geq 0 \wedge f(\mathbf{x}) \leq 0$. They do this because equality constraints can be manipulated more efficiently, while, moreover, each (non-redundant) equality constraint reduces the effective dimension by one, further improving efficiency.

A *disequality* constraint $f(\mathbf{x}) \neq 0$ can also be represented as a pair of inequality constraints

$$f(\mathbf{x}) \geq 1 \vee f(\mathbf{x}) \leq -1. \tag{1}$$

This means that again an explicit representation is not required, but it can also lead to improved efficiency. In particular, since the libraries mentioned above move disjunctions out, each additional disequality constraint doubles the number of disjuncts in the outer disjunction, which can quickly become unmanageable.

While there has been some attention for explicit disequality constraints in the related field of abstract interpretation, see, e.g., Péron and Halbwachs (2007), they appear to have been mostly ignored within the field of polyhedral compilation. In particular, none of the libraries above support explicit disequality constraints. Seater and Wonnacott (2005) describe an (unimplemented) algorithm for detecting *inert* disequality constraints, i.e., those that can safely be ignored, but the "ert" disequality constraints would still cause exponential behavior. They do suggest that an alternative approach would be to allow explicit disequality constraints, but do not provide any details on the implications. The present paper provides such details. A `PBDD` (Kulkarni and Kruse 2022) is an alternative representation for polyhedra using decision diagrams that allows for an efficient representation of negations, including specifically negations of equality constraints, i.e., disequality constraints. Intersection, union, subtraction and complement operations can be performed directly on this representation. For other operations, the implementation relies on `isl`.

IMPACT 2024, January 17, 2024, Munich, Germany

```
for (int i = 0; i < n; ++i) {
  if (i == p0 || i == p1 || i == p2)
    continue;
  A[i] = i;
}
for (int i = 0; i < n; ++i) {
  if (i == p0 || i == p1 || i == p2)
    continue;
  B[i] = A[i];
}
```

Listing 1. Motivating Example of Kulkarni and Kruse (2022)

Disequality constraints typically appear when a set satisfying an equality constraint is subtracted from some other set. This can happen during dependence analysis (Seater and Wonnacott 2005) or even in the representation of statement instance sets such as in Listing 1, adapted from Kulkarni and Kruse (2022), which they report commonly happens inside `Polly` (Grosser, Größlinger, et al. 2012). Klebanov (2015) provides another example based on Meng and Smith (2011), which is, essentially, to count the number of elements in the set

$$\{ [r1] \} \cup \{ [r2] \} \cup \{ [r3] \} \cup \{ [r4] \} \cup \{ [r5] \} \cup \{ [r10] \},$$

with r1, r2, r3, r4, r5, r10 symbolic constants. This is a trivial problem in terms of counting, but not in terms of representing the result, since the count depends on the number of the symbolic constants that are equal to each other. Even with

explicit disequality constraints, it takes nearly 2s to compute, but without, it takes nearly 2min because there are many more cases to consider.

This paper describes how `isl` can be adjusted to support explicit disequality constraints. In order to be able to explain the modifications, some details about the base implementation are provided, i.e., the one without such support. Only the $\boxed{\neq}$-paragraphs should be considered as contributions of this paper. No claim is made that the base implementation is as efficient as possible, but it should be a reasonable starting point. Most of the algorithms of this base implementation have later also been reused in FPL.

2 Internal Representation

In `isl`, a set is represented as a union of *basic sets* of the form

$$\{\, \mathbf{x} : \exists \boldsymbol{\alpha}, \boldsymbol{\beta} : A_1\mathbf{n} + A_2\mathbf{x} + A_3\boldsymbol{\alpha} + A_4\boldsymbol{\beta} + \mathbf{a} \geq \mathbf{0} \wedge \boldsymbol{\beta} = \lfloor (B_1\mathbf{n} + B_2\mathbf{x} + B_3\boldsymbol{\alpha} + B_4\boldsymbol{\beta} + \mathbf{b})/\mathbf{d} \rfloor \,\}, \tag{2}$$

where all matrices have integer elements, the division by $\mathbf{d}$ is performed row-wise and B_4 is strictly lower triangular so that no β_i is defined in terms of itself or a later β_j. The $\boldsymbol{\alpha}$ are called (proper) *existentially quantified variables*, while the $\boldsymbol{\beta}$ are called *integer divisions*. Together, they form the *local variables*. As explained in Section 1, the description (2) may also involve explicit equality constraints. The basic sets in a union may overlap, but some operations (e.g., the counting of Section 4.5) require the basic sets to be disjoint, which can be achieved using set subtraction (Section 4.2). The integer tuple $\mathbf{x}$ may be missing from the description to represent constraints on only the symbolic constants, or it may be split up into a pair of tuples $\mathbf{x} \to \mathbf{y}$ to represent a binary relation on tuples. The tuples can also be named and nested, while tuples with different names and/or sizes can appear in the same (union) set (Verdoolaege 2011), but this is not important for the present paper.

If the coefficients of a constraint have a non-trivial common divisor g, then it is factored out. For inequality constraints, the constant term a is tightened to $\lfloor a/g \rfloor$. For equality constraints, the constant terms needs to be a multiple of g as well. Otherwise, the set is marked empty. Note that this normalization implies that if a constraint only involves a single symbolic constant or variable, then it has coefficient 1 or -1.

Besides sets and binary relations, `isl` also has object types representing functions. The basic type is that of a quasi-affine function, mapping the symbolic constants by themselves or in combination with some set variables to some rational value. It is called quasi-affine because it may involve integer divisions similar to those of (2) (but no proper existentially quantified variables). A piecewise quasi-affine function consists of a subdivision of the domain into disjoint cells, each represented by an `isl` set, with a quasi-affine function attached to each cell. The function value at a given point is the function value of the quasi-affine function attached to the cell containing the point. It is undefined if there is no such cell.

There are various constructors for sets, including one called `non_zero_set`, which takes a quasi-affine function f and returns the set where this function is non-zero. In the base `isl` implementation, this constructor returns a set consisting of two disjuncts, one with $f \geq 1$ and one with $f \leq -1$.

$\boxed{\neq}$**.** The description of a basic set (2) can now also have explicit disequality constraints $C\mathbf{z} + \mathbf{c} \neq \mathbf{0}$, with $\mathbf{z} = (\mathbf{n}, \mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\beta})$ grouping the symbolic constants and all variables, i.e.,

$$\{\, \mathbf{z}_2 : \exists \mathbf{z}_3, \mathbf{z}_4 : A\mathbf{z} + \mathbf{a} \geq 0 \wedge C\mathbf{z} + \mathbf{c} \neq 0 \wedge \mathbf{z}_4 = \lfloor (B\mathbf{z} + \mathbf{b})/\mathbf{d} \rfloor \,\}.$$

If the constant term of a disequality constraint is not a multiple of the common divisor of the coefficients, then the constraint is simply dropped. The `non_zero_set` constructor now returns a single-disjunct set with constraint $f \neq 0$.

2.1 Parallel Constraints

Various simplifications are performed on the constraint representation, including Gaussian elimination using the equality constraints as well as the removal of duplicate constraints. If, after Gaussian elimination, an inequality constraint is simplified to $a \geq 0$, with a a constant, then the constraint is dropped if a is non-negative and the basic set is marked empty if a is negative. If there are two inequality constraints $f(\mathbf{z}) + a_1 \geq 0$ and $f(\mathbf{z}) + a_2 \geq 0$, then only the one with the smallest constant term is kept. Similarly, if there are two inequality constraints $f(\mathbf{z}) + a_1 \geq 0$ and $-f(\mathbf{z}) + a_2 \geq 0$, then if $a_1 + a_2 < 0$, the basic set is marked as empty. If $a_1 + a_2 = 0$, the pair of inequality constraints is replaced by an equality constraint. Otherwise, if there is some existentially quantified variable α_i with a coefficient f_i in f that is greater than $a_1 + a_2$, then α_i can only attain the single value $\alpha_i = \lfloor (-f^\circ(\mathbf{z}) + a_2)/f_i \rfloor$, with $f^\circ(\mathbf{z}) = f(\mathbf{z}) - f_i\alpha_i$, and α_i is turned into an integer division, provided the constraint on B_4 is not violated.

$\boxed{\neq}$**.** If a disequality constraint is simplified to $a \neq 0$, with a a constant, then the constraint is dropped if a is not zero and the basic set is marked empty if a is zero. A disequality constraint can only be removed based on some other disequality constraint if the two are completely identical. However, if there is a disequality constraint $f(\mathbf{z}) + a_1 \neq 0$ and an inequality constraint $f(\mathbf{z}) + a_2 \geq 0$, then the disequality constraint can be dropped if $a_2 < a_1$. If $a_2 = a_1$, then the inequality constraint can first be replaced by $f(\mathbf{z}) + a_2 - 1 \geq 0$. Similarly for the same disequality interpreted as $-f(\mathbf{z}) - a_1 \neq 0$

3 Incremental LP Solver

Many operations in `isl` rely on an incremental LP solver based on the description of Detlefs et al. (2005).

3.1 Tableau

The incremental LP solver operates on a representation of a basic set called a *tableau*. In its basic form, no distinction is made between symbolic constants, variables and local variables, so that description (2) simplifies to $\{ \mathbf{z} : A\mathbf{z}+a \geq \mathbf{0} \}$, where each integer division $\beta = \lfloor (B\mathbf{z} + b)/d \rfloor$ is encoded as $B\mathbf{z} + b - (d-1) \leq d\beta \leq B\mathbf{z} + b$. Let

$$\mathbf{f} = A\mathbf{z} + \mathbf{a} \tag{3}$$

and let $\mathbf{z}$ and $\mathbf{f}$ be of sizes m and n respectively. Both $\mathbf{z}$ and $\mathbf{f}$ are considered tableau variables. The only difference is that the $\mathbf{f}$ variables are marked as being non-negative. The tableau is essentially an $n \times (m+1)$ matrix that defines n of the tableau variables in terms of the other m, with an extra column for the constant term. In principle, the entries in the tableau are rational numbers, but `isl` maintains a single shared denominator for all the entries in a row so that the numerators are all integers. The variables that appear in columns are called the *column variables*. Initially, these are the variables $\mathbf{z}$, but this can change as described below. Similarly for the variables in the rows. When the column variables are assigned the value zero, the value of the the row variables is given by the constant column. This assignment is called the *sample value* of the tableau and can easily be read off. A tableau is in a valid state if the sample value is non-negative for all non-negative variables. If a tableau is not in a valid state, then *pivots* can be performed until a valid state is reached, where each pivot interchanges a row variable and a column variable. In essence, the row that defines the row variable in terms of the column variables is used to define the selected column variable in terms of the row variable and the remaining column variables and this definition is plugged into the remaining rows. The details of which row and column variable to interchange are beyond the scope of this paper. If no valid state can be reached, then the tableau (as well as the set it represents) is empty. (Note that this particular tableau representation has no specific notion of an objective function. If one is needed, it is just added as a row with a variable that is not marked non-negative.)

Any variable corresponding to an equality constraint is moved into a column and then this column is *killed*, meaning that the variable is assigned the fixed value zero and that the column is (effectively) removed.

A tableau is a representation of a rational set. If a tableau can be put into a valid state, then the set has rational elements, but not necessarily integer elements. However, if the sample value of a valid tableau happens to consist of integer values for all of the original set variables $\mathbf{z}$, then these entries form an element of the set and the set is non-empty. Such a sample value is considered *valid*. Note that due to (3), the variables $\mathbf{f}$ will also have integer values.

Additional row variables can be added to the tableau after its initial creation. An undo stack keeps track of this addition, along with any conclusion that was drawn after the addition of the constraint so that they can all be undone when the tableau is reverted to the state before the addition. For example, some non-zero variables may have been determined to be redundant, some variables may have been determined to be zero or the tableau may have been determined to be empty. If no undo stack has been set up, then the rows corresponding to redundant variables, as well as dead columns are simply removed. Otherwise, they are preserved for a potential later revival. However, these rows and columns are ignored for pivot selection.

$\boxed{\neq}$**.** If there are $n^{\neq}$ disequality constraints in total, then the tableau is extended with $n^{\neq}$ rows and $n^{\neq}$ variables $\mathbf{g} = C\mathbf{z} + \mathbf{c}$ that are marked non-zero. A non-zero variable is allowed to attain a zero sample value and is never selected for pivoting. This means non-zero variables will always be row variables. They are otherwise treated in the same way as other (row) variables. Whenever a column is killed, the non-zero variables with a zero sample value are inspected. If any of them has only zero coefficients for the non-dead columns, then the tableau is marked empty. A sample value is only considered valid if, besides having integer values for the original set variables, it also has non-zero values for all non-zero variables.

3.2 Detecting Implicit Equality Constraints

Some of the inequality constraints may actually be equality constraints when combined with other inequality constraints. They can be detected by iterating through all non-negative variables that have a sample value smaller than 1 and checking if they can attain a value greater than or equal to 1 through pivoting. If not, a new row variable is added that is opposite to the given non-negative variable. By definition, this new variable cannot attain positive values. If it cannot attain the value zero either, the tableau is marked empty. Otherwise, it is moved into a column and the column is killed.

3.3 Detecting Redundant Constraints

If, when a non-negative variable is first added or during pivoting, a non-negative row variable is seen to be the sum of a non-negative sample value and a non-negative combination of only non-negative variables, then this variable is automatically marked redundant.

Non-negative variables can also be explicitly tested for being redundant as follows. Temporarily ignoring the non-negative character of a given variable, the tableau checks if it is possible to attain a sample value smaller than or equal to -1 through pivoting, without violating the other (non-redundant) non-negative variables. If not, the corresponding constraint is redundant since it can only attain non-negative values at integer points.

$\boxed{\neq}$. The detection of redundant non-negative variables is not affected by the addition of disequality constraints, but the corresponding non-zero variables could also be redundant. If the sample value of a non-zero variable is zero, then the variable is not redundant. If it is negative, then an attempt is made to attain a non-negative sample value. If this fails, the constraint is redundant. Similarly, if the sample value is positive, then an attempt is made to attain a non-positive sample value. If this fails, the constraint is again redundant.

3.4 Detecting Redundant Local Variables

Projecting out a variable itself is a trivial operation as it simply involves existentially quantifying the variable, i.e., moving it from $\mathbf{x}$ to $\boldsymbol{\alpha}$ in (2). However, `isl` performs various simplifications on a set description, including the detection of redundant local variables.

The procedure for detecting redundant local variables is similar to part of the Omega test (Pugh 1991). Consider all pairs of lower and upper bounds on a given local variable α in a given disjunct (2). If each such pair is such that there is at least one integer value between the bounds, then α can safely be eliminated (using Fourier-Motzkin) without introducing any other (integer) values for the other variables. This includes the case where there are only lower bounds or only upper bounds since then there are no pairs of lower and upper bounds. Consider a pair $n^{\mathrm{l}}\alpha + f^{\mathrm{l}} \geq 0$ and $-n^{\mathrm{u}}\alpha + f^{\mathrm{u}} \geq 0$ with $n^{\mathrm{l}}, n^{\mathrm{u}} \geq 0$ and $f^{\mathrm{l}}, f^{\mathrm{u}}$ not involving α. Let $g = \gcd(n^{\mathrm{l}}, n^{\mathrm{u}})$, $m^{\mathrm{l}} = n^{\mathrm{l}}/g$ and $m^{\mathrm{u}} = n^{\mathrm{u}}/g$. Then $m^{\mathrm{u}} f^{\mathrm{l}} \leq m^{\mathrm{u}} m^{\mathrm{l}} g\alpha \leq m^{\mathrm{l}} f^{\mathrm{u}}$, or, equivalently (since $mx \leq mf$ is equivalent to $mx \leq mf + (m-1)$),

$$-m^{\mathrm{u}} f^{\mathrm{l}} - (m^{\mathrm{u}} - 1) \leq m^{\mathrm{u}} m^{\mathrm{l}} g\alpha \leq m^{\mathrm{l}} f^{\mathrm{u}} + (m^{\mathrm{l}} - 1). \quad (4)$$

If

$$\Big(m^{\mathrm{l}} f^{\mathrm{u}} + (m^{\mathrm{l}} - 1)\Big) - \Big(-m^{\mathrm{u}} f^{\mathrm{l}} - (m^{\mathrm{u}} - 1)\Big) + 1 \geq m^{\mathrm{u}} m^{\mathrm{l}} g \quad (5)$$

holds, then there is at least one integer value of α satisfying (4). If $n^{\mathrm{l}} = 1$ or $n^{\mathrm{u}} = 1$, then (5) is automatically satisfied since it is then equivalent to the rational elimination of α from the two constraints. If $m^{\mathrm{l}} f^{\mathrm{u}} - m^{\mathrm{u}} f^{\mathrm{l}} = 0$, then (5) involves only numerical constants and can be directly evaluated. Otherwise, condition (5) is evaluated in the tableau of the disjunct. For example, for y in $S = \{\, [x,z] : \exists y : x \leq y \leq z \,\}$, condition (5) has the form $z - x + 1 \geq 1$, which holds in S and the description can be simplified to $S = \{\, [x,z] : x \leq z \,\}$.

$\boxed{\neq}$. In the presence of disequality constraints, it is not sufficient to have a gap that holds at least one integer value since that integer value may not satisfy one of the disequality constraints. However, for any value of the other variables, a given disequality constraint can only invalidate a single value of the local variable. Let $n^{\neq}$ be the number of disequality constraints involving α. Condition (5) can be replaced by

$$\Big(m^{\mathrm{l}} f^{\mathrm{u}} + (m^{\mathrm{l}} - 1)\Big) - \Big(-m^{\mathrm{u}} f^{\mathrm{l}} - (m^{\mathrm{u}} - 1)\Big) + 1 \geq (1 + n^{\neq}) m^{\mathrm{u}} m^{\mathrm{l}} g$$

to ensure that elimination is still performed correctly, with the elimination of a local variable causing all disequality constraints involving that variable to be removed as well. For $S' = \{\, [x,z] : \exists y : x \leq y \leq z \wedge y \neq 0 \,\}$, the modified condition is $z - x + 1 \geq 2$, which is not valid on the entire S'. That is $S' \neq \{\, [x,z] : x \leq z \,\}$. In particular, $[0,0] \notin S'$. On the other hand, $\{\, [x] : \exists y : 0 \leq y \leq 9 \wedge y \neq x \,\} = \{\, [x] \,\}$ since $0 - (-9) + 1 \geq 2$. Still, in general, the presence of disequality constraints reduces the opportunity for detecting redundant local variables.

3.5 Emptiness and Sampling

Emptiness of a basic set is determined by looking for an element satisfying the constraints, i.e., a sample. If no sample can be found, the set is empty. First, an attempt is made to write the basic set as a Cartesian product of lower-dimensional sets (Verdoolaege and Bruynooghe 2008, Section 7). Each of these lower-dimensional sets is then handled separately. Zero- and one-dimensional sets have a trivial solution. A zero-dimensional set has an integer point if and only if it is rationally non-empty. A one-dimensional set has either no constraints, in which case any value will do, a single equality constraint, which fixes the single variable to a single (integer) value, or one or two inequality constraints, in which case a value saturating an inequality constraint can be chosen. Note that in a one-dimensional set all constraints are parallel, so that redundant and conflicting constraints are automatically removed (see Section 2.1).

Next, the bounded dimensions are separated from the unbounded dimensions. This is achieved by computing the recession cone of the set S, i.e., the set bounded by all equality and inequality constraints with the constant terms replaced by zero. The (implicit) equality constraints of the recession cone (as computed in Section 3.2) determine the bounded directions. Let $n' \leq n$ be the number of linearly independent equality constraints and let $n'' = n - n'$. A unimodular transformation moves these n' directions into the first dimensions $\mathbf{z}'$. The problem is then decomposed into sampling a bounded n'-dimensional set S', defined by the constraints not involving the unbounded dimensions $\mathbf{z}''$, and sampling an n''-dimensional cone obtained by plugging in the sample value (if any) for the bounded dimensions.

To find a sample in a bounded set, a backtracking search is performed. In each level, the minimal and maximal rational values are determined in a given direction and then all integer values in between are considered, fixing the value for lower levels by killing a column in the tableau. Note that there may be no such integer values, in which case the procedure backtracks immediately. This happens in particular if fixing a value causes the tableau to become empty. The directions in which to search are determined by generalized basis reduction (Cook et al. 1993). As soon as an integer sample is found, either because some intermediate state of the tableau

has a valid sample or because the lowest level of the search has been reached, the search is terminated.

As soon as a sample value $\mathbf{s}'$ of S' has been found, the original set is known to be non-empty. In fact, if $n'' > 0$, it will have an infinite number of integer points. It is just a matter of selecting one of them. Plugging in $\mathbf{s}'$ in the unimodularly transformed original set yields a polyhedron S'' in the $\mathbf{z}''$ variables. Pick a *rational* point $\mathbf{r}''$ in S'', e.g., by reading it off from the corresponding tableau. If this happens to be an integer point, a sample of S'' has been found. Otherwise, consider the (unimodularly transformed) recession cone, projected onto the final n'' dimensions and shifted to $\mathbf{r}''$, which forms a subset of S''. If the recession cone is defined by $A\mathbf{z}'' \geq 0$, then the shifted cone is defined by $A\mathbf{z}'' \geq A\mathbf{r}''$. In order not to have to scale the coefficients to handle the denominator of $\mathbf{r}''$, this affine cone can be further restricted to $C'' = \{ \mathbf{z}'' : A\mathbf{z}'' \geq \lceil A\mathbf{r}'' \rceil \}$. Let T be the cone of elements $\mathbf{t}$ such that the entire unit cube at $\mathbf{t}$ lies inside C'', i.e.,

$$T = \{ \mathbf{t} : \forall I \subseteq \mathbb{N}_{\leq n''} : \mathbf{t} + \sum_{i \in I} \mathbf{e}_i \in C'' \}, \tag{6}$$

with $\mathbb{N}_{\leq n''}$ the natural numbers smaller than or equal to n'' and $\mathbf{e}_i$ the unit vectors in the space. The constraints of T are all shifted copies of those of C''. In particular, for a constraint $A_j\mathbf{z} \geq \lceil A_j\mathbf{r}'' \rceil$ of C'', the shifted copies are $A_j\mathbf{z} \geq \lceil A_j\mathbf{r}'' \rceil - \sum_{i \in I} a_{ji}$ of C''. The most restrictive of these shifted copies is

$$A_j\mathbf{z} \geq \lceil A_j\mathbf{r}'' \rceil - \sum_{i : a_{ji} < 0} a_{ji} \tag{7}$$

and the constraints of the form (7) are therefore sufficient to define T. Now take any *rational* point $\mathbf{t}$ in T. By (6), $\mathbf{s}'' = \lceil \mathbf{t} \rceil$ is an element of C'' and therefore of S''. Concatenating $\mathbf{s}'$ and $\mathbf{s}''$ and unimodularly transforming the result back to the original space yields a sample of S.

$\boxed{\neq}$. Disequality constraints are ignored during the construction of the recession cone. Disequality constraints involving unbounded dimensions are left out of the description of the bounded set, just like the other constraints involving unbounded dimensions. These are the "inert" disequality constraints of Seater and Wonnacott (2005). They do not affect emptiness, but they do affect the sample value when the full-dimensional cone is sampled. During the search in the bounded set, the remaining (ert) disequality constraints are ignored, except for their effect on the interpretation of a valid sample and on the killing of columns, potentially resulting in an empty tableau during the search.

The original procedure for sampling S'' may happen to pick a point that lies on one of the $n^{\neq}$ inert disequality constraints. The cone T is therefore further restricted to a cone that contains the base points of hypercubes of size $1 + n^{\neq}$ rather than just 1. That is, the constraints (7) are replaced by

$$A_j\mathbf{z} \geq \lceil A_j\mathbf{r}'' \rceil - (1 + n^{\neq}) \sum_{i : a_{ji} < 0} a_{ji}. \tag{8}$$

The disequality constraints are sorted based on the position of their last non-zero coefficient. If $\mathbf{s}'' = \lceil \mathbf{t} \rceil$ happens to lie on one or more of the disequality constraints, then one with the smallest position of the last non-zero coefficient is picked and $\mathbf{s}''$ is incremented at that position. A disequality that has been handled in this way never needs to be reconsidered because $\mathbf{s}''$ will only be incremented at the same or a later position. Incrementing the same position only moves the point further away from the disequality constraint. Incrementing a later position does not affect the disequality constraint at all. This process needs to be repeated at most $n^{\neq}$ times and because of (8), the entire sequence of points lies in C''.

An alternative approach would be to break up the disequality constraints on-the-fly. In the worst case, this results in $2^{n^{\neq}}$ subsets, but only a single subset would be active at any time and the search can be terminated as soon as a sample has been found in any of the subsets. Furthermore, if all subsets are empty, in which case all $2^{n^{\neq}}$ would have to be considered, the emptiness is likely to be discovered early on so that large pieces of the search tree could be cut. However, the modifications to the original implementation described above seemed easier to implement.

3.6 Scanning

Scanning iterates over all integer points in a (bounded) set. It applies the same procedure as sampling a bounded set from Section 3.5, except that the search is not terminated when a sample has been found.

$\boxed{\neq}$. Scanning is unaffected by disequality constraints as long as no hidden assumptions are made that only apply to convex sets. In particular, in the absence of disequality constraints, at the innermost level, all integer values between the minimal and maximal rational value are in the set, but if there are any disequality constraints, then the tableau may be empty for some of those values.

3.7 Hull Operations

While `isl` provides a convex hull operation, it is rarely used in practice since it is very much an operation on rational sets and tends to result in constraints with large coefficients. The "simple" hull provides an alternative and consists of only constraints that already appear in the original set description, possibly shifted by a constant. This simple hull may then of course be much larger than the convex hull. The (integer) affine hull is computed by starting from and an initial (under)approximation, typically a single sample point, looking for a sample that violates any of the equality constraints defining the current approximation and then updating the approximation with this sample using the procedure of Karr (1976) until no more outside samples can be found.

$\boxed{\neq}$. The computation of the convex hull is unaffected and ignores disequality constraints. The computation of the

affine hull is also unaffected, except for the changes to the sampling procedure of Section 3.5. For example, consider the set $\{ [x, y] : -1 \le x + y \le 1 \wedge -1 \le x - y \le 1 \wedge x \ne 0 \}$. An initial sample point could be $\{ [1, 0] \}$, i.e., $\{ [x, y] : x = 1 \wedge y = 0 \}$. No sample points in the original set can be found satisfying $y < 0$, $y > 0$ or $x > 1$. Only $x < 1$ yields a new sample point, $\{ [-1, 0] \}$, and the approximation is updated to $\{ [x, 0] \}$. This is the final affine hull since no further sample points can be found satisfying $y < 0$ or $y > 0$. As a side-effect of the affine hull computation, the description of the original set is simplified to $\{ [x, 0] : -1 \le x \le 1 \wedge x \ne 0 \}$.

The simple hull computation requires a bit more thought since this operation is defined in terms of the representation of the set and it is not immediately obvious how to extend that to disequality constraints. Two sensible options are to either ignore disequality constraints or to only preserve disequality constraints that are valid in all disjuncts. The first option seems the safest since some users may be assuming that the result is convex (in its local variables). However, the operation is sometimes "abused" to turn a set with a single disjunct into an object of type basic set. In this case, the disequality constraints of that single disjunct should be preserved, so a separate operation has been added to perform the type conversion.

3.8 Coalescing

Coalescing consists of replacing a pair of disjuncts by a single, equivalent disjunct (Verdoolaege 2015). As a trivial example, $\{ [x] : 0 \le x \le 10 \} \cup \{ [x] : 5 \le x \le 15 \}$ can be replaced by the single disjunct $\{ [x] : 0 \le x \le 15 \}$. Coalescing is implemented in `isl` by first determining the relation between a disjunct and the constraints of the other disjunct and then handling different patterns, possibly combined with some further validity checks. In particular, a constraint of one disjunct may be *valid* for the other disjunct (i.e., redundant as in Section 3.3), it may *separate* the second disjunct from the first (i.e., it may be invalid for the entire disjunct), or it may *cut* the second disjunct (i.e., be valid for only part of this disjunct). For certain patterns, some special cases of separation are also considered, but these are not relevant here. One of the more trivial patterns is where all constraints of one disjunct are valid for the other disjunct. In this case, the second disjunct is a subset of the first and the pair can be replaced by just the first disjunct.

A disjunct that involves (proper) existentially quantified variables is not considered for coalescing. Disjuncts involving integer divisions are handled, but only if the two disjuncts have the same integer divisions or if they can easily be made to be the same.

$\boxed{\neq}$. The type of a disequality $f \ne 0$ is determined as follows. If the corresponding equality $f = 0$ conflicts with the other disjunct, i.e., adding the equality makes the tableau empty, then the disequality $f \ne 0$ is considered valid for this other disjunct. If, on the other hand, the equality $f = 0$ is a linear combination of the equality constraints of the other disjunct, then the disequality $f \ne 0$ is considered to separate the other disjunct. Otherwise, the disequality is considered to cut the other disjunct.

If any disequality constraint of either disjunct cuts the other disjunct, then the two disjuncts are not coalesced. If all disequality constraints of both disjuncts are valid for the other disjunct, the coalescing is performed as before. The only minor adjustment is that the disequality constraints are also taken into account if the two disjuncts are replaced by a single disjunct with constraints from both disjuncts. The only new pattern is one where a single disequality constraint $f \ne 0$ of disjunct A separates disjunct B, while all other constraints of A are valid for B. The tableau of A is then modified to represent a set A' where $f \ne 0$ is replaced by $f = 0$. Note that $B \subseteq A'$ because $f \ne 0$ separates B and all other constraints of A are valid for B. If all constraints of B are valid in this tableau, i.e., also $A' \subseteq B$, then the pair is replaced by a single disjunct defined by all valid constraints of A and B, i.e., those of A, except the disequality $f \ne 0$. As an example, consider the set

$$\{ [x, y] : y \ge -2x \wedge y \ge 2x \wedge y \ne 3 \} \cup \{ [-1{:}1, 3] \}. \quad (9)$$

The constraints $y \ge -2x$ and $y \ge 2x$ are valid for $B = \{ [-1{:}1, 3] \}$, while $y \ne 3$ separates B. The modified tableau represents $A' = \{ [x, 3] : 3 \ge -2x \wedge 3 \ge 2x \}$. Note that no tightening is performed in the tableau representation. Still, the constraints $x + 1 \ge 0$ and $-x + 1 \ge 0$ are valid in this tableau because they do not attain a value smaller than or equal to -1.

4 Quantifier Elimination

The operations in this section require the removal of (proper) existentially quantified variables, either because constraints need to be transplanted from one set to another (e.g., set subtraction of Section 4.2) or because local variables are treated as regular variables and can only be allowed to attain a single value to avoid overcounting in Section 4.5. In theory the gist operation of Section 4.3 could simply ignore constraints involving existentially quantified variables, but the `isl` implementation removes them instead.

The problem with existentially quantified variables is that they can attain more than one value for fixed values of the other variables and that these possible values depend on all the constraints. In `isl`, they are removed by selecting one particular value for them that can moreover be described as a quasi-affine expression of the other variables. In particular, parametric integer programming (Feautrier 1988) is used to compute the (lexicographically) minimal value that can be attained by these variables, as a function of the other variables. This may result in a split of a disjunct into multiple disjuncts and in the introduction of additional integer divisions, but it will remove all existentially quantified variables.

4.1 Parametric Integer Programming

The `isl` implementation of parametric integer programming uses essentially the same tableau of Section 3.1, but the variables are split into two types: the parameters and the optimization variables. The parameters always remain column variables, while the optimization variables can become row variables through pivoting. For computing the (lexicographic) minimum of a set, the symbolic constants are the parameters, while the set variables (along with any local variables) are optimization variables. For computing the minimum range element of a binary relation, both the symbolic constants and the domain variables of the binary relation are considered parameters. For quantifier elimination, the symbolic constants and all tuple variables are parameters, while the local variables are optimization variables. Integer divisions that are defined in terms of only parameters can also be considered parameters.

Similarly to Section 3.1, pivoting is based on the signs of the row variables. However, in this case, the signs cannot simply be read off from the sample value of the tableau. Instead, the sign is now determined by a parametric constant term, i.e., an affine expression in the parameter, and the question is then whether there are any values of the parameters where this expression is negative or non-negative. A separate "context" tableau is maintained in terms of only the parameters of the main tableau. An affine expression $f(\mathbf{n})$ can be non-negative if adding the constraint $f(\mathbf{n}) \geq 0$ to the context tableau does not make it empty. In the original `piplib` (Feautrier et al. 2007) implementation, emptiness of the context tableau is performed through a non-parametric lexicographic optimization. In `isl`, an incremental variant of the procedure of Section 3.5 is used.

If the parametric constant term $f(\mathbf{n})$ can attain both negative and non-negative values, then the context is split into $f(\mathbf{n}) \geq 0$ and $f(\mathbf{n}) \leq -1$ and a (possibly) different solution is computed. This naturally leads to a decision tree with such splits in the nodes and solutions in the leaves. In the `isl` implementation, this is immediately converted to either a set/binary relation representation or a piecewise quasi-affine function.

When using parametric integer programming for quantifier elimination, these splits, if there are any, cause the single disjunct of which the existentially quantified variables are eliminated to be split into multiple disjuncts. Extra integer divisions can get introduced when the candidate solution is non-integral and a parametric cut is introduced to cut away some rational solutions (without cutting out any integral solutions).

When the minimum of a set is computed that consists of multiple disjuncts, then the minima of all disjuncts are first computed individually and then combined. Feautrier (1991) describes a procedure for combining decision trees. In `isl`, a piecewise quasi-affine function representation is computed for the minimum of each disjunct and pairs of such functions are then successively combined into a single function. This function is then optionally converted to a set or binary relation. Given two piecewise quasi-affine functions $p^1(\mathbf{x})$ and $p^2(\mathbf{x})$, with n^i cells S^i_j with associated function f^i_j, the representation of the minimum has at most $2n_1n_2$ cells, half of which of the form $S^1_j \cap S^2_k \cap \{ \mathbf{x} : f^1_j(\mathbf{x}) \preccurlyeq f^2_k(\mathbf{x}) \}$ with associated function f^1_j and the other half of the form $S^1_j \cap S^2_k \cap \{ \mathbf{x} : f^1_j(\mathbf{x}) \succ f^2_k(\mathbf{x}) \}$ with associated function f^2_k.

$\boxed{\neq}$**.** If a disjunct involves any disequality constraint, then the solution is computed in the form of a piecewise quasi-affine function. Just as in Section 3.1, the tableau keeps track of the disequality constraints in extra rows, updating them when pivoting, but otherwise ignoring them until an integral solution has been found in some subset of the original context. Each disequality is then represented as $f(\mathbf{n}) + g(\mathbf{c}) \neq 0$, with $\mathbf{c}$ the current column variables. These column variables are set to zero in the solution, so the disequality is violated if $f(\mathbf{n}) = 0$. This constraint is evaluated in the context tableau. If it can never hold, the original disequality is respected by the current solution. Otherwise, the context is again split, but now into $f(\mathbf{n}) \neq 0$ and $f(\mathbf{n}) = 0$. In the first part the disequality is satisfied and the other disequality constraints are evaluated, potentially resulting in further splits. In the second part, two separate solutions are computed, one where $f(\mathbf{n})+g(\mathbf{c}) \neq 0$ is replaced by $f(\mathbf{n})+g(\mathbf{c}) \geq 1$ and one where it is replaced by $f(\mathbf{n}) + g(\mathbf{c}) \leq -1$. These two solutions are then locally combined using the procedure above into a single solution for the part $f(\mathbf{n}) = 0$. The two cases are the same as those of (1), but they are only considered if and when they are needed.

4.2 Set Subtraction

Subtracting a set $A = \bigcup_i A_i$ from a set $B = \bigcup_j B_j$ is performed by successively subtracting the disjuncts of A from those of B, where the disjuncts A_i and B_j are basic sets. Quantifier elimination is first applied to A so that the description of A can be assumed to be free of (proper) existentially quantified variables. This means the constraints of A_i can be evaluated in the context of B_j. Each redundant constraint is ignored since its negation conflicts with B_j. The disjuncts of the difference $B_j \setminus A_i$ correspond to the non-redundant constraints $f_k(\mathbf{z}) \geq 0$. In particular, each disjunct is equal to B_j with an extra constraint $f_k(\mathbf{z}) \leq -1$. Since set subtraction is also used to make the disjuncts of a set disjoint, the disjuncts of the difference are made disjoint by also adding the constraints $f_\ell(\mathbf{z}) \geq 0$ for all $\ell < k$. An equality constraint is treated as a pair of inequality constraints, with each leading to an extra disjunct if it is individually non-redundant.

The same procedure is also used for checking if one set is a subset of another, but the difference is not computed

completely. Instead, the computation is terminated as soon as one of the pieces is determined to be non-empty.

$\boxed{\neq}$. A disequality $f \neq 0$ in the description of A_i is considered redundant for B_j if adding the constraint $f = 0$ makes B_j empty. A non-redundant disequality results in a disjunct in the difference with $f = 0$ and is added to each subsequent disjunct in the difference. The handling of equality constraints $g = 0$ is slightly modified when computing a set difference. If both corresponding inequality constraints are non-redundant for B_j then they now result in a single disjunct in the difference with extra constraint $g \neq 0$, rather than two disjuncts with extra constraints $g \geq 1$ and $g \leq -1$ respectively. When evaluating a subset relation, an equality constraint continues to be treated as a pair of inequality constraints. This can lead to additional cases being considered, but the extra cases are necessarily empty and are usually obviously so.

4.3 Gist

The gist operation is based on that of `Omega` (Pugh and Wonnacott 1995), where the gist of A given a context B is intended to describe the extra information in A not already provided by B. In essence, it is a set A' such that $A' \cap B = A \cap B$ and such that A' has a description that is as "simple" as possible. There are many possible such A' and the `isl` implementation mainly tries to remove redundant constraints. Quantifier elimination is applied to both A and B in order to be able to compare their respective constraints. This means that the final result of the gist may consist of more disjuncts than the input. If, after quantifier elimination, B consists of multiple disjuncts, then it is replaced by its simple hull so that redundancy can be evaluated in the tableau of this single disjunct. Some special cases have also been implemented such as one where $B \subseteq A$, in which case the universe set is used for A'.

$\boxed{\neq}$. Care needs to be taken not to apply the simple hull indiscriminately. That is, if the context consists of a single disjunct already, then the disequality constraints should be preserved. It could also be useful to preserve disequality constraints that are valid for all disjuncts. Another change is that disequality constraints of A that are redundant with respect to B are also removed using the mechanism of Section 3.3.

4.4 Transitive Closure

In general, the transitive closure of a binary relation cannot be represented exactly using Presburger formulas (Kelly, Pugh, et al. 1996). The `isl` implementation described by Verdoolaege, Cohen, et al. (2011) always produces a result that contains the exact transitive closure as a subset, along with an indication of whether this is an approximation. Most of the implementation applies operations to sets and binary relations as a whole and only one part needs to consider the constraint representation. This happens in particular in the computation of (an approximation of) kS, with S some single-disjunct set and $k \geq 1$. Quantifier elimination is first applied so that this representation does not involve proper existentially quantified variables. If the constraints $A\mathbf{x}+\mathbf{a} \geq 0$ of disjunct S do not involve any symbolic constants, then the constraints of kS are $A\mathbf{x} + k\mathbf{a} \geq 0$, $k \geq 1$. Otherwise, a case distinction is made. Constraints that do not involve symbolic constants also have their constant terms multiplied by k. Constraints that only involve symbolic constants are simply copied without modification. For any other constraint, if the symbolic constant term is non-positive over S, then the constraint is copied since that means that the linear expression in the set variables is non-negative and therefore k times the set variables will satisfy the same constraint. Otherwise, the constraint is dropped from the approximation of kS.

$\boxed{\neq}$. Since $f \neq 0$ is just $f \geq 1 \vee f \leq -1$, the same reasoning can be applied to the two inequality constraints separately. The conclusion (multiplying constant term by k or not) is usually the same for both of them and can therefore be applied directly to the disequality constraint. Only the case of a non-positive symbolic constant term cannot be applied directly. The symbolic constant term of the disequality would need to only attain values between -1 and 1 for the corresponding terms of the inequality constraints to both be non-positive. This case is therefore not currently considered for disequality constraints. Note also that, in general, if the transitive closure cannot be computed exactly, then breaking up the input into subsets typically leads to more accurate results. A single disjunct with a disequality will therefore also typically lead to less accurate results than two disjuncts with inequality constraints.

4.5 Counting

While `isl` does not currently support any counting itself, the counting library `barvinok` (Verdoolaege, Seghir, et al. 2007) does use `isl` for some set manipulations. This includes quantifier elimination and then on each disjunct parameter and variable compression (Meister 2004; Meister and Verdoolaege 2008) and decomposing the disjunct into a Cartesian product of lower-dimensional sets (Verdoolaege and Bruynooghe 2008, Section 7). In particular, `isl` provides a function that takes a basic set and a callback returning a piecewise quasi-polynomial. The `isl` function calls the callback zero or more times and combines the result, i.e., it multiplies the results obtained for the factors in the Cartesian product.

$\boxed{\neq}$. The core counting algorithm in `barvinok`, i.e., the callback mentioned above, does not know or even need to know about disequality constraints. Instead, the `isl` function handles the disequality constraints using a variant of the inclusion-exclusion principle (Andreescu and Feng 2004). For example, if $S = S' \cap \{ \mathbf{x} : f_1(\mathbf{x}) \neq 0 \wedge f_2(\mathbf{x}) \neq 0 \}$, then

$$\operatorname{card} S = \operatorname{card} S' - \operatorname{card} S'_1 - \operatorname{card} S'_2 + \operatorname{card} S'_{1,2}$$

with $S'_i = S' \cap \{ \mathbf{x} : f_i(\mathbf{x}) = 0 \}$ and $S'_{1,2} = S' \cap \{ \mathbf{x} : f_1(\mathbf{x}) = 0 \wedge f_2(\mathbf{x}) = 0 \}$. If there are $n^{\neq}$ disequality constraints, then there are $2^{n^{\neq}}$ cases to consider. Splitting up each disequality constraint into a pair of inequality constraints would result in the same number of cases. However, all but one of the cases considered here are of a lower dimensionality since they have one or more additional equality constraints. This means computing the cardinality of these other cases is (much) cheaper. Furthermore, some combinations of corresponding equality constraints may conflict with each other and then any case with additional equality constraints no longer needs to be considered, cutting off a branch from the tree of cases.

5 Polyhedral Compilation

Most `isl` operations are generic operations on integer sets, but some are more specific to polyhedral compilation. They are briefly described in this section and are not heavily affected by the presence of disequality constraints.

5.1 Scheduling

The support for scheduling in `isl` is described in detail by Verdoolaege and Janssens (2017). The search for schedule coefficients is based on the collection of inequality constraints that are valid for all points in a set.

$\boxed{\neq}$. Internal disequality constraints have no effect on the collection of valid (affine) inequality constraints. Redundant disequality constraints are removed in Section 3.3 and disequality constraints that cut off an entire facet cause a tightening of the corresponding inequality constraint in Section 2.1. It would also be possible to detect disequality constraints $f \neq 0$ that are equivalent to either $f \geq 1$ or $f \leq -1$ given the other constraints. This could be done in a way that is similar to the detection of redundant disequality constraints.

5.2 AST generation

AST generation takes a schedule and generates an AST that visits the scheduled elements in the order specified by the schedule. The `isl` implementation is described by Grosser, Verdoolaege, et al. (2015). While generating loops at a certain level, the inner schedule dimensions are projected out. The strategy for breaking up the sets of values of the current schedule dimension is selected through an option. The "atomic" setting ensures that a single loop is generated for each scheduled domain, "separate" means that each loop executes a fixed set of scheduled domains (so that the body is free of `if`-conditions) and "unroll" that each loop consists of a single iteration. If the option is not specified explicitly, then the sets of values are simply broken up into disjoint subsets, with an effect that is somewhat similar to "separate", but without any guarantees. When using the atomic option, a simple hull is computed and any constraints not covered by this simple hull will get inserted as `if`-conditions in the generated AST. At several points during the construction, a set not involving the inner schedule dimensions is split up into a list of (disjoint) basic sets that then undergo further processing, including sorting them in the scheduling order.

```
for (int c0 = 0; c0 < n; c0 += 1)
  if (c0 != p0 && c0 != p1 && c0 != p2) {
    A[c0] = (c0);
    B[c0] = A[c0];
  }
```

Listing 2. Optimized Example

$\boxed{\neq}$. By default, disequality constraints involving the current schedule dimension are removed when constructing loops. The disequality constraints may then appear as `if`-conditions in the generated AST, meaning that a new type of operation representing `!=` needs to be added to the AST representation. If the separate option is specified, disequality constraints involving the current schedule dimension are first split up into a disjunction of inequality constraints, ensuring the body remains free of `if`-conditions. The same split of disequality constraints is performed whenever a set is split up into a list of basic sets. This removes disequality constraints that get introduced through set subtractions and ensures that the resulting basic sets continue to satisfy the assumptions of the operations performed on them. In particular, it ensures that the basic sets can continue to be sorted along the current schedule dimension.

6 Related Work

In terms of rational sets, Imbert (1993) considers negations of *systems* of equality constraints to be able to support variable elimination in the presence of negations of equality constraints. Péron and Halbwachs (2007) focus on disequality constraints of the form $x \neq y$ or $x \neq 0$ in the context of difference-bound matrices. Ghorbal et al. (2012) propose an abstract representation that is the difference of two convex sets, i.e., a polyhedron with a hole. Kulkarni and Kruse (2022) propose an alternative representation for (the integer points in) a union of polyhedra consisting of a decision diagram with constraints in the nodes. They support a more general form of negation, i.e., not just the negation of an equality constraint. However, they only support a limited number of operations and do not support existentially quantified variables. Seater and Wonnacott (2005) do consider general Presburger sets, but focus on the detection of disequality constraints that can be safely ignored.

7 Experiments

The changes described above have been implemented on top of a development version of `isl`. Users of `isl` need little to no modifications. The only change required for the polyhedral model extractor `pet` (Verdoolaege and Grosser

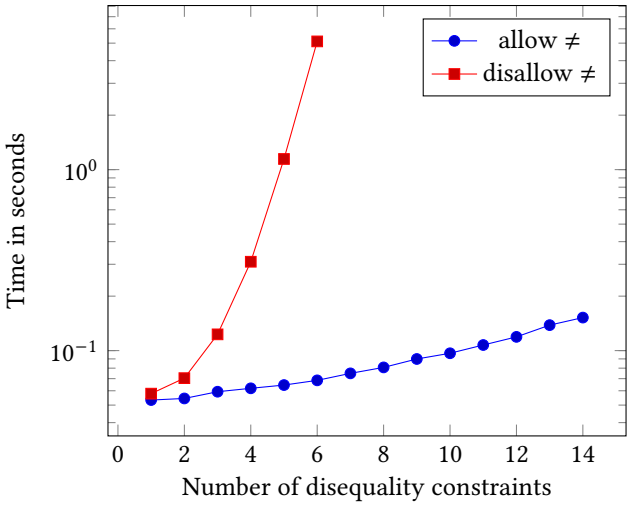

Figure 1. Execution time of PPCG on variants of Listing 1

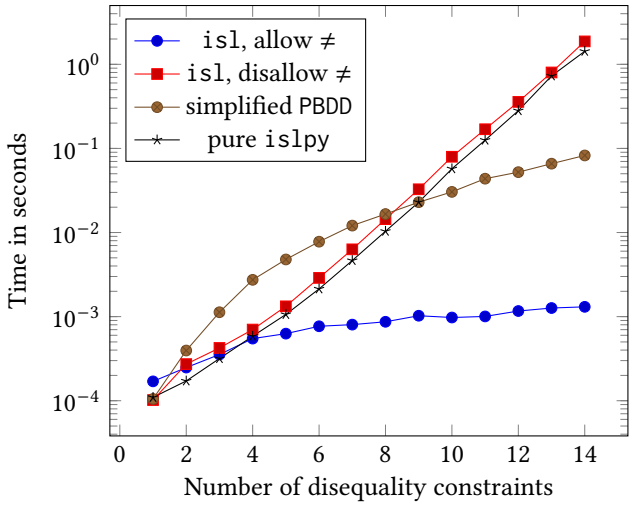

Figure 2. Iterative intersection of complement

2012) is that one of the test cases is further simplified due to a gist with a context involving a disequality. The polyhedral code generator PPCG (Verdoolaege, Juega, et al. 2013) does not require any adjustments to take the code in Listing 1 and produce the code in Listing 2. Note that PPCG does not specify the "separate" AST generation option and therefore allows the AST generator to generate `if`-conditions inside loop bodies. The code in Listing 1 has 3 disequality constraints. Figure 1 shows the PPCG execution time when varying the number of disequality constraints, both with support for disequality constraints and without. Without explicit disequality constraints, the representation of the statement instance sets and derived information quickly grows out of control.

The use of explicit disequality constraints has not yet been evaluated in the context of DTG (Verdoolaege, Kudlur, et al. 2020) because DTG has its own local `isl` changes, making it more challenging to have it use the version of `isl` with support for disequality constraints.

Figure 2 reproduces an experiment of Kulkarni and Kruse (2022) building the set $\{ [i] : \bigwedge_{j \le n^{\neq}} i \neq \mathrm{p}_j \}$ with PBDD using `islpy` (Klöckner 2014) version 2023.2.5, which apparently uses `isl` version 0.26. The same figure also shows the time needed for the same construction using the development version of `isl`, both with and without support for explicit disequality constraints. Note that execution times between the two pairs cannot be compared directly because PBDD has been implemented in Python and because the underlying version of `isl` is different. Note also that intersection in `isl` implies an implicit emptiness check.

Figure 3 performs a similar experiment using lexicographic minimization (Section 4.1) and cardinality (Section 4.5), operations that are not (directly) supported by PBDD. Note that the input set needs to be bounded for these operations to make sense, so the test set is first intersected with

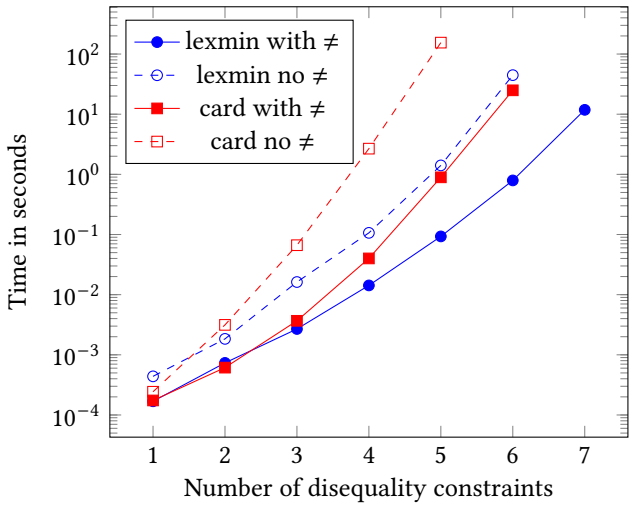

Figure 3. Lexicographic minimization and cardinality

$\{ [i] : 0 \le i < \mathrm{n} \}$. In both cases, support for explicit disequality constraints allows an extra disequality constraint to be handled within the same time budget. The main limiting factor here is the size of the result of the operation, which would require more than disequality constraints to represent efficiently. Note, however, that such inputs represent extreme cases that should be rare in practice.

8 Conclusion

This paper has shown that it is feasible to extend the internal representation of `isl` with explicit disequality constraints and to adjust all the operations supported by `isl` accordingly, leading to significantly reduced computation times in a realistic scenario. In future, it may be useful to consider explicit representations of other constraint types that would otherwise lead to disjunctions, e.g., lexicographic constraints.

A Prototype

The attached x86-64 ELF executable `iscc` can be used to experiment with the support for explicit disequality constraints, by specifying or leaving out the `--no-allow-disequality` option. Note that the executable bit may need to be turned on after saving the attachment.

References

Andreescu, Titu and Zuming Feng (2004). "Inclusion-Exclusion Principle." In: *A Path to Combinatorics for Undergraduates: Counting Strategies*. Boston, MA: Birkhäuser Boston, pp. 117–141. DOI: 10.1007/978-0-8176-8154-8_6.

Bagnara, Roberto, Patricia M. Hill, and Enea Zaffanella (2008). "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems." In: *Science of Computer Programming* 72.1–2, pp. 3–21. DOI: 10.1016/j.scico.2007.08.001.

Chen, Chun (June 2012). "Polyhedra scanning revisited." In: *SIGPLAN Not.* 47.6, pp. 499–508. DOI: 10.1145/2345156.2254123.

Cook, William, Thomas Rutherford, Herbert E. Scarf, and David F. Shallcross (1993). "An Implementation of the Generalized Basis Reduction Algorithm for Integer Programming." In: *ORSA Journal on Computing* 5.2.

Detlefs, David, Greg Nelson, and James B. Saxe (2005). "Simplify: a theorem prover for program checking." In: *J. ACM* 52.3, pp. 365–473. DOI: 10.1145/1066100.1066102.

Feautrier, Paul (1988). "Parametric Integer Programming." In: *RAIRO Recherche Opérationnelle* 22.3, pp. 243–268.

Feautrier, Paul (1991). "Dataflow analysis of array and scalar references." In: *International Journal of Parallel Programming* 20.1, pp. 23–53. DOI: 10.1007/BF01407931.

Feautrier, Paul, Jean François Collard, and Cédric Bastoul (2007). *PIP/PipLib: A Solver for Parametric Integer Programming Problems*.

Ghorbal, Khalil, Franjo Ivančić, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta (2012). "Donut domains: efficient non-convex domains for abstract interpretation." In: *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'12. Philadelphia, PA: Springer-Verlag, pp. 235–250. DOI: 10.1007/978-3-642-27940-9_16.

Grosser, Tobias, Armin Größlinger, and Christian Lengauer (2012). "Polly - Performing polyhedral optimizations on a low-level intermediate representation." In: *Parallel Processing Letters* 22.04. DOI: 10.1142/S0129626412500107.

Grosser, Tobias, Sven Verdoolaege, and Albert Cohen (July 2015). "Polyhedral AST generation is more than scanning polyhedra." In: *ACM Transactions on Programming Languages and Systems* 37.4, 12:1–12:50. DOI: 10.1145/2743016.

Imbert, Jean-Louis (1993). "Variable elimination for disequations in generalized linear constraint systems." In: *The Computer Journal* 36.5, pp. 473–484. DOI: 10.1093/comjnl/36.5.473.

Karr, Michael (1976). "Affine Relationships Among Variables of a Program." In: *Acta Informatica* 6, pp. 133–151. DOI: 10.1007/BF00268497.

Kelly, Wayne, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott (Nov. 1996). *The Omega Library*. Tech. rep. University of Maryland.

Kelly, Wayne, William Pugh, Evan Rosser, and Tatiana Shpeisman (1996). "Transitive closure of infinite graphs and its applications." In: *International Journal of Parallel Programming* 24.6, pp. 579–598. DOI: 10.1007/BFb0014196.

Klebanov, Vladimir (2015). *Personal communication*.

Klöckner, Andreas (2014). "Loo.Py: Transformation-based Code Generation for GPUs and CPUs." In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: ACM, 82:82–82:87. DOI: 10.1145/2627373.2627387.

Kulkarni, Shubhang and Michael Kruse (June 2022). "Polyhedral Binary Decision Diagrams for Representing Non-Convex Polyhedra." In: *12th International Workshop on Polyhedral Compilation Techniques (IMPACT 2022)*. Budapest, Hungary.

Meister, Benoît (Dec. 2004). "Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization." PhD thesis. Université Louis Pasteur.

Meister, Benoît and Sven Verdoolaege (Apr. 2008). "Polynomial Approximations in the Polytope Model: Bringing the Power of Quasi-Polynomials to the Masses." In: *Digest of the 6th Workshop on Optimization for DSP and Embedded Systems, ODES-6*. Ed. by Jagadeesh Sankaran and Tom Vander Aa. DOI: 10.5281/zenodo.10003255.

Meng, Ziyuan and Geoffrey Smith (2011). "Calculating Bounds on Information Leakage Using Two-Bit Patterns." In: *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*. PLAS '11. San Jose, California: Association for Computing Machinery. DOI: 10.1145/2166956.2166957.

Péron, Mathias and Nicolas Halbwachs (2007). "An Abstract Domain Extending Difference-Bound Matrices with Disequality Constraints." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Byron Cook and Andreas Podelski. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 268–282. DOI: 10.1007/978-3-540-69738-1_20.

Pitchanathan, Arjun, Christian Ulmann, Michel Weber, Torsten Hoefler, and Tobias Grosser (Oct. 2021). "FPL: Fast Presburger Arithmetic through Transprecision." In: *Proc. ACM Program. Lang.* 5.OOPSLA. DOI: 10.1145/3485539.

Pugh, William (1991). "The Omega test: a fast and practical integer programming algorithm for dependence analysis." In: *Proceedings of the 1991 ACM/IEEE conference on*

Supercomputing. Albuquerque, New Mexico, United States: ACM Press, pp. 4–13. DOI: 10.1145/125826.125848.

Pugh, William and David Wonnacott (1995). "Going beyond integer programming with the Omega test to eliminate false data dependences." In: *Parallel and Distributed Systems, IEEE Transactions on* 6.2, pp. 204–211. DOI: 10.1109/71.342135.

Seater, Robert and David Wonnacott (2005). "Efficient Manipulation of Disequalities During Dependence Analysis." In: *Proceedings of the 15th International Conference on Languages and Compilers for Parallel Computing*. LCPC'02. College Park, MD: Springer-Verlag, pp. 295–308. DOI: 10.1007/11596110_20.

Verdoolaege, Sven (2010). "isl: An Integer Set Library for the Polyhedral Model." In: *Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

Verdoolaege, Sven (Apr. 2011). "Counting Affine Calculator and Applications." In: *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*. Chamonix, France. DOI: 10.13140/RG.2.1.2959.5601.

Verdoolaege, Sven (Jan. 2015). "Integer Set Coalescing." In: *Proceedings of the 5th International Workshop on Polyhedral Compilation Techniques*. Amsterdam, The Netherlands. DOI: 10.13140/2.1.1313.6968.

Verdoolaege, Sven and Maurice Bruynooghe (July 2008). "Algorithms for Weighted Counting over Parametric Polytopes: A Survey and a Practical Comparison." In: *The 2008 International Conference on Information Theory and Statistical Learning*. Ed. by Matthias Beck and Thomas Stoll. DOI: 10.5281/zenodo.10031041.

Verdoolaege, Sven, Albert Cohen, and Anna Beletska (2011). "Transitive Closures of Affine Integer Tuple Relations and Their Overapproximations." In: *Proceedings of the 18th International Conference on Static Analysis*. SAS'11. Venice, Italy: Springer-Verlag, pp. 216–232. DOI: 10.1007/978-3-642-23702-7_18.

Verdoolaege, Sven and Tobias Grosser (Jan. 2012). "Polyhedral Extraction Tool." In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France. DOI: 10.13140/RG.2.1.4213.4562.

Verdoolaege, Sven and Gerda Janssens (June 2017). *Scheduling for PPCG*. Report CW 706. Leuven, Belgium: Department of Computer Science, KU Leuven. DOI: 10.13140/RG.2.2.28998.68169.

Verdoolaege, Sven, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor (2013). "Polyhedral parallel code generation for CUDA." In: *ACM Trans. Archit. Code Optim.* 9.4, p. 54. DOI: 10.1145/2400682.2400713.

Verdoolaege, Sven, Manjunath Kudlur, Rob Schreiber, and Harinath Kamepalli (Jan. 2020). "Generating SIMD Instructions for Cerebras CS-1 using Polyhedral Compilation Techniques." In: *10th International Workshop on Polyhedral Compilation Techniques (IMPACT'20)*. Bologna, Italy. DOI: 10.5281/zenodo.4295955.

Verdoolaege, Sven, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe (June 2007). "Counting integer points in parametric polytopes using Barvinok's rational functions." In: *Algorithmica* 48.1, pp. 37–66. DOI: 10.1007/s00453-006-1231-0.

Wilde, Doran K. (1993). *A Library for doing polyhedral operations*. Tech. rep. 785. IRISA, Rennes, France, 45 p.