Integer Set Coalescing

Sven Verdoolaege
INRIA and KU Leuven
sven.verdoolaege@inria.fr

ABSTRACT

In polyhedral compilation, various core concepts such as the set of statement instances, the access relations, the dependences and the schedule are represented or approximated using sets and binary relations of sequences of integers bounded by (quasi-)affine constraints. When these sets and relations are represented in disjunctive normal form, it is important to keep the number of disjuncts small, both for efficiency and to improve the computation of transitive closure over-approximations and AST generation. This paper describes the set coalescing operation of `isl` that looks for opportunities to combine several disjuncts into a single disjunct without affecting the elements in the set. The main purpose of the paper is to explain the various heuristics and to prove their correctness.

1. INTRODUCTION AND MOTIVATION

Polyhedral compilation is a framework for analyzing and transforming program fragments that are "sufficiently regular" through a mathematical abstraction that models the individual statement instances and array elements using a compact representation. This compact representation may be the integer points in parametric polyhedra [11], or, more generally, a Presburger relation [13], i.e., a set of tuples of integers satisfying a Presburger formula. The constraints are usually kept in disjunctive normal form (or, more specifically, as a union of polyhedra) and to keep the representation as compact as possible, it is therefore important to keep the number of disjuncts as small as possible.

There are several operations that may result in representations of sets with more disjuncts than strictly necessary. The most obvious such operation is taking the union of two sets since the two sets may have disjuncts that can be combined into one. Two disjuncts in the same set representation that cannot originally be combined may also become combinable after a projection onto a lower-dimensional space. Subtracting one set from another is usually performed by negating each constraint of the subtrahend in turn, which may break up the result into smaller pieces than strictly necessary. Finally, parametric integer programming [6] recursively splits the parameter domain into two parts along a hyperplane as long as there are any rows with undetermined sign in the tableau. These hyperplanes may then split convex parts of the parameter domain with a single solution into multiple disjuncts.

For most operations that can be performed on sets of integer tuples, the outcome does not depend on the representation of the set. In these cases, reducing the number of disjuncts in the representation only helps to improve efficiency. However, there are also some operations, most notably those where heuristics are used to compute an approximation or where the result is not a Presburger set, that do depend on the choice of the representation. Prime examples are the computation of approximations of transitive closures [10, 18] and the generation of an AST from a polyhedral schedule [2,4]. As an illustration of the effect on transitive closures, repeating the `CLooG` equivalence experiments of [20, Table 1], both with and without the coalescing of this paper, shows that turning off coalescing not only makes 4 out 113 cases time out or run out of memory where the result was inconclusive, but that it also does the same on 2 cases where equivalence could be proved. Turning off coalescing in AST generation generally results in more verbose output, but the effect on the performance of the generated code is typically only noticeable on highly optimized code. For example, when applying the basic hybrid tiling strategy of [8] (without some of the extra improvements) on a 2D heat kernel, turning off coalescing makes the performance drop from 11.8 GFLOP/s to 10.6 GFLOP/s on a GK107GLM.

One way of reducing the number of disjuncts is to remove those disjuncts that are covered by other disjuncts, as is done automatically by the `DomainUnion` operation of `PolyLib` [21] or the internal `rm_redundant_conjs` function of `Omega` [9].[1] However, this does not help in case of disjuncts that only partially overlap or of disjuncts that have been split up into two or more disjoint disjuncts. For *rational* sets, i.e., polyhedra, partially overlapping disjuncts have been considered, where the problem is known as convexity recognition [3] or exact join detection [1].

A pragmatic approach is taken by, e.g., `CLooG` [2] in its original `PolyLib` backend. First the convex hull H of the input set S is computed. This convex hull may contain extra integer points $H \setminus S$. These are removed again to result in $H \setminus (H \setminus S)$. There are several issues with this approach.

IMPACT 2015
Fifth International Workshop on Polyhedral Compilation Techniques
Jan 19, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

`http://impact.gforge.inria.fr/impact2015`

[1] Note that the Omega project consistently interchanges the words "conjunct" and "disjunct".

First, the convex hull is an operation that is defined on rational sets and it is not obvious how to define the operation on integer sets, especially if existentially quantified variables are involved, without resorting to the even more expensive integer hull. In fact, the exact meaning of the convex hull on sets described by constraints involving existentially quantified variables is left undefined in `isl`, because there is no obvious definition in this case and the user should not rely on the specific result that the current implementation may happen to produce. The only guarantee provided is that the result of a convex hull operation will always be a single disjunct superset of the input.

Another issue with this approach is that if the convex hull contains additional integer points, then the result of the subtraction may in principle be composed of *more* disjuncts than the input. The number of constraints of the convex hull may also be exponential in the number of constraints in the input. There is no guarantee that this number can be reduced through simplification and even if it can, the convex hull would still need to be computed first. The constraints of the convex hull are also prone to having very large coefficients, which is especially problematic in case of AST generation. In practice, this approach also turns out to be significantly slower than the coalescing approach. For example, using coalescing, `isl` takes 16s on its AST generation test cases, while using the convex hull based approach takes 24min (Fourier-Motzkin based implementation) or 6min (wrapping based implementation). It should be noted that the convex hull implementation in `isl` may not be the most efficient. On the other hand, double description based implementations, where a polyhedron is represented both in terms of constraints and in terms of vertices and rays [14], can be very costly too.

In contrast to the convex hull based approach, the coalescing operation in `isl` never increases the total number of constraints. The approach is based on a pairwise combination of disjuncts that fit one of a number of recognized patterns. The analysis and any modifications of the constraints are based on solving LP problems of the same dimension as the input set. Modified constraints are always linear combinations of two original constraints, reducing the risk for the introduction of extremely large coefficients.

The remainder of the paper is organized as follows. Section 2 introduces some background information along with some building blocks that will be needed to perform the coalescing. Section 3 classifies the constraints of one disjunct in terms of their effect on the other disjunct. This classification is then used in Section 4 to describe the different coalescing heuristics on pairs of disjuncts. The handling of existentially quantified variables and multiple disjuncts is described in Section 5 and Section 6. After a discussion of related work in Section 7, the paper concludes in Section 8.

2. BACKGROUND AND COMPONENTS

This section describes some background information and introduces some operations that are available in the literature and that will be used to detect and exploit coalescing opportunities.

2.1 Set representation in `isl`

In `isl`, the constraints of a set may be specified by the user as an arbitrary Presburger formula, i.e., a first order logic formula that essentially only contains additions and comparisons. That is, the interpreted function symbols are the integer constants, plus ($+$), minus ($-$), integer division by an integer constant ($\lfloor \cdot/d \rfloor$), one for each positive integer d, and a set of symbolic constants. The single interpreted relation symbol is less-than-or-equal ($\leq$). Internally, this description is converted into disjunctive normal form

$$\left\{ \mathbf{x} : \bigvee_i \left(\exists_j \alpha_{i,j} : \bigwedge_k (t_{i,k}(\mathbf{x}, \boldsymbol{\alpha}_i) \geq 0) \right) \right\},$$

with $t_{i,k}(\mathbf{x}, \boldsymbol{\alpha}_i)$ a quasi-affine expression, i.e., an affine expression in the variables, symbolic constants and integer divisions of other quasi-affine expressions. That is, a quasi-affine expression is a term constructed from variables, symbolic constants, integer constants, addition, subtraction and integer division by an integer constant. Any common factor among the coefficients of an inequality is removed. That is, $g\,t(\mathbf{x}) \geq 0$ with g an integer greater than one is replaced by $t(\mathbf{x}) \geq 0$. Pairs of inequalities $t(\mathbf{x}) \geq 0$ and $-t(\mathbf{x}) \geq 0$ are replaced by an equality constraint $t(\mathbf{x}) = 0$ and used to simplify the other constraints. Some operations on integer sets require the existentially quantified variables to have been eliminated first. This elimination is performed by computing unique representatives of these variables using parametric integer programming [6], which may introduce additional integer division expressions.

2.2 Incremental LP solver

An LP solver can be used to find the extremal value of an affine expression over a polyhedron, i.e., a *rational* set bounded by affine constraints. The LP solver that is used internally in `isl` is incremental, meaning that constraints can be added and removed again, and is modeled after the one in `Simplify` [5]. The input polyhedron usually represents the rational superset of one of the disjuncts in an integer set description, bounded by the same constraints. Each integer division $\lfloor t(\mathbf{x})/d \rfloor$ in such a description is replaced by an additional variable α and constraints $t(\mathbf{x}) - (d+1) \leq d\,\alpha \leq t(\mathbf{x})$ to ensure that the integer values attained by α correspond exactly to the integer division.

In the internal representation of the LP solver (the *tableau*), each constraints $t_j(\mathbf{x}) \geq 0$ is represented by a constraint variable y_j. If the polyhedron is described by m constraints in a space of dimension n, then the tableau expresses m row variables as affine expressions of n column variables, where the $m + n$ row and column variables form a permutation of the n coordinate variables and the m constraint variables. A *pivot* step interchanges a column and a row variable. The *sample value* of a tableau assigns zero to all column variables and the constant term of the affine expression of the row variables in terms of the column variables to those row variables. When an extra constraint is added, its initial sample value may be negative, in which case pivots are performed until all constraint variables have a non-negative sample value. When an equality $t_j(\mathbf{x}) = 0$ is added to the tableau, the corresponding variable is moved to a column and marked *dead*, meaning that it will never be considered for pivoting and will therefore always have a zero value. As a preparation for the coalescing, an attempt is made for each inequality constraint to see if it can be pivoted into having a strictly positive sample value (or a value greater than or equal to one in case of integer sets). If not, then the corresponding variable is similarly moved to a dead column.

2.3 Wrapping

Wrapping is a technique for taking a constraint that is invalid for a given polyhedron and "rotating" it around a valid constraint for that polyhedron until it becomes valid. That is, wrapping adds the smallest multiple of the valid constraint to the invalid constraint such that the linear combination becomes valid. The technique was originally designed to compute a facet of the convex hull of a union of polytopes given a previously computed facet and their shared ridge [7]. Although the technique applies to a union of polyhedra, we will only need it for wrapping constraints around a single polyhedron. Wrapping is therefore described here in this more restricted context.

After applying an affine transformation to both the polyhedron and the two constraints, we may assume that the valid constraint is $x_1 \geq 0$, while the invalid constraint is $x_2 \geq 0$. The aim is to find the greatest a such that $x_2 \geq a\, x_1$ is a valid constraint. Note that this may fail if the polyhedron is unbounded in the negative x_2 direction. The trick of finding this a is to determine the corresponding constraint of the cone generated by the polyhedron, without explicitly computing this cone. Note that the cone generated by a set S is the polyhedron $\{\, \lambda\, \mathbf{x} : \lambda \geq 0 \wedge \mathbf{x} \in S \,\}$ or $\{\, \mathbf{x} : \exists \lambda \geq 0 : \mathbf{x} \in \lambda\, S \,\}$. The desired value of a is the smallest value of the ratio x_2/x_1, with $x_1 \neq 0$, in this cone, or simply the smallest value of x_2 when x_1 is fixed to 1. Note that there is no need to project out λ in order to obtain this value. Given a polyhedron bounded by the constraints

$$A \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix} \geq \mathbf{0}$$

the value of a may then be obtained by solving the LP problem

$$\min x_2 \quad \text{s.t. } A \begin{bmatrix} \lambda \\ \mathbf{x} \end{bmatrix} \geq \mathbf{0} \wedge \lambda \geq 0 \wedge x_1 = 1.$$

Note that this problem has the same dimension as the input polyhedron. The extra variable λ is compensated by the equality $x_1 = 1$. If $a = n/d$, with n an integer and d a positive integer, then the new constraint is $-n\, x_1 + d\, x_2 \geq 0$, or, in terms of the original, untransformed, constraints

$$-n\, t_1(\mathbf{x}) + d\, t_2(\mathbf{x}) \geq 0.$$

If the minimum is unbounded, i.e., $a = -\infty$ ("$= -1/0$"), then this means S is unbounded in the negative x_2 direction. In this case, wrapping fails. The "wrapped" constraint would simply be $t_1(\mathbf{x}) \geq 0$ which would not be suitable for coalescing since it would not have any contribution from the $t_2(\mathbf{x}) \geq 0$ constraint.

The wrapping technique will be used by some of the coalescing heuristics in Section 4 to turn an initially invalid constraint into a valid constraint. As in the case of computing a convex hull, there is a risk that the result may have large coefficients. There is however a major difference. When using wrapping to compute the convex hull, the wrapping is applied recursively to the results of other wrappings. The final constraints may therefore contain contributions from many constraints in the input. During coalescing, wrapping is only ever applied to a pair of constraints that appear in the input. The risk for large coefficients is therefore much lower. Moreover, if a large coefficient would be introduced, it can be detected immediately, without having to wait for the ultimate effect at the end of the recursion. The coalescing procedure of `isl` can be instructed to refuse coalescings with coefficients that are larger than those in the input. This option is currently turned on by default.

The wrapping LP problems are currently created from scratch in `isl`. In principle, it would be possible to start from a tableau of S, add variables corresponding to the constraints $x_1 \geq 0$ and $x_2 \geq 0$ (in the original coordinate system), move these variables into column positions and then use the resulting state of the tableau as the coordinate transformation that moves these constraints into their $x_1 \geq 0$ and $x_2 \geq 0$ form. Finally, the constant terms can be considered as the coefficient of a newly introduced λ variable, after which the constraints $\lambda \geq 0$ and $x_1 = 0$ can be added.

2.4 Variable Compression

Variable compression [12] is a technique for mapping an integer lattice defined by a set of equalities to the standard integer lattice (in a lower-dimensional space). Given the equalities $M\mathbf{x} + \mathbf{c} = 0$, the first step is to compute the (left) Hermite normal form

$$M = \begin{bmatrix} H_1 & 0 \end{bmatrix} \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} \qquad \text{or} \qquad M \begin{bmatrix} U_1 & U_2 \end{bmatrix} = \begin{bmatrix} H_1 & 0 \end{bmatrix},$$

with U and $Q = U^{-1}$ unimodular matrices and H_1 lower triangular. Setting

$$\begin{bmatrix} \mathbf{x}_1' \\ \mathbf{x}_2' \end{bmatrix} = \begin{bmatrix} Q_1 \\ Q_2 \end{bmatrix} \mathbf{x} \qquad \text{i.e.,} \qquad \mathbf{x} = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1' \\ \mathbf{x}_2' \end{bmatrix},$$

the equalities become $H_1\, \mathbf{x}_1' + \mathbf{c} = 0$, i.e., $\mathbf{x}_1' = -H_1^{-1}\mathbf{c} =: \mathbf{c}'$. We may assume that the elements of $\mathbf{c}'$ are integers as otherwise the set satisfying the given equalities is empty and can be discarded. The transformation is then defined by

$$\mathbf{x} = -H_1^{-1}\mathbf{c} + U_2 \mathbf{x}_2' \qquad \text{and} \qquad \mathbf{x}_2' = Q_2 \mathbf{x}.$$

Since the equalities reduce to $-\mathbf{c} + \mathbf{c} = 0$ in terms of the transformed $\mathbf{x}_2'$, we see that we have indeed mapped the lattice for $\mathbf{x}$ described by the original equalities to the standard lattice for $\mathbf{x}_2'$.

3. CONSTRAINT TYPING

A crucial step in determining if and how two disjuncts A and B can be coalesced is to check how the constraints of A relate to B. Let us first consider the case where the sets are considered as rational sets. This case serves as an introduction to the integer case and will also be used directly in Section 4.1. Let $t(\mathbf{x}) \geq 0$ be a constraint of A. Its effect on B may of one of three types

- $\min_{\mathbf{x} \in B} t(\mathbf{x}) \geq 0$. The constraint is *valid* for B.
- $\max_{\mathbf{x} \in B} t(\mathbf{x}) < 0$. The constraint *separates* A from B.
- $\min_{\mathbf{x} \in B} t(\mathbf{x}) < 0$ and $\max_{\mathbf{x} \in B} t(\mathbf{x}) \geq 0$. The constraint *cuts* B into a part where it is valid and a part where it is not.

Note that determining the type of a constraint only requires the solution of one LP problem. When the constraint is added to the tableau, the initial sample value will be either negative or non-negative. In the first case, the constraint can only be a separating or a cut constraint, while in the second case, it can only be a valid or a cut constraint. In both cases, a single LP problem is sufficient to determine which of the two applies. In fact, in case of a cut constraint, it is

not even necessary to find the exact minimum or maximum. The search can be aborted as soon as the sample value of the constraint variable changes sign. If A has any equality constraints $t(\mathbf{x}) = 0$, then the types of both $t(\mathbf{x}) \geq 0$ and $-t(\mathbf{x}) \geq 0$ are determined.

Let us now consider the case of integer sets and let us assume for now that the two disjuncts have no existentially quantified variables or integer divisions. A discussion on how these are handled is postponed until Section 5. We may assume that the constraint $t(\mathbf{x}) \geq 0$ has only integer coefficients. This means that the constraint can only attain integer values on integer points. The class of valid constraints can therefore be slightly extended to the case where $\min_{\mathbf{x} \in B} t(\mathbf{x}) > -1$. This allows for some constraints that only cut off rational points from the rational approximation of B to be considered valid for B.

We may also consider a couple of special cases of separating constraints.

- $t = -1$ in the tableau (ignoring dead columns). This means that $t(\mathbf{x}) + 1 = 0$ is valid for the entire set B. The constraint $t(\mathbf{x}) \geq 0$ is said to be *adjacent to an equality*.
- $t = -1 - u$, with u a variable corresponding to a constraint of B. This means that every integer point in the universe satisfies either $t(\mathbf{x}) \geq 0$ or $u(\mathbf{x}) \geq 0$. The constraint $t(\mathbf{x}) \geq 0$ is said to be *adjacent to an inequality*. Note that this case can also be discovered more efficiently by maintaining a hash table of the linear parts of the constraints of B.

Constraints that are adjacent to an equality or an inequality are crucial for the cases considered in Section 4.2 and Section 4.3. If any equalities are involved that impose stride constraints, then these special cases may fail to be recognized depending on how the inequality constraints are expressed with respect to these equalities. In such cases (in particular, if there is an equality involving more than two variables or an equality with coefficients different from one, negative one or zero), the inequality constraints are first simplified in a standard lattice. That is, a variable compression (Section 2.4) is computed and applied to each constraint. If the linear coefficients of the resulting constraint have a common factor g, then it is divided out and the constant term c is replaced by $\lfloor c/g \rfloor$. Afterwards, the constraint is mapped back to the original space. Since the stride g has been removed from the constraint, it can more easily be recognized as being adjacent to an equality of inequality.

EXAMPLE 3.1. *Consider the set* $\{ (x, y) : 2x = 3y \wedge 0 \leq x \leq 6 \} \cup \{ (-3, -2) \}$. *The constraint* $x \geq 0$ *reduces to* $-2 \geq 0$ *in the tableau of disjunct* $\{ (-3, -2) \}$ *and is therefore not recognized as being adjacent to an equality of this disjunct. Variable compression yields* $x = -3x'$, $y = -2x'$ *and* $x' = -x+y$. *In the compressed space, the constraint is of the form* $-3x' \geq 0$, *which is simplified to* $-x' \geq 0$ *and transformed back to* $x - y \geq 0$. *This simplified constraint reduces to* $-3 - (-2) \geq 0$ *and therefore is recognized as adjacent to an equality.*

4. COALESCING HEURISTICS

Coalescing in `isl` works by considering a number of templates. Given two disjuncts, constraint typing is performed to determine the effect of the constraints of one disjunct on the other. Based on these constraint types, one or more cases may apply and, if needed, further tests and/or computations are performed. If these are successful, then the two disjuncts are replaced by a single disjunct. The cases described in Section 4.1 do not introduce any additional points and therefore also apply to rational sets. The other cases introduce extra rational points, but only if they are known not to attain an integer value on a given constraint, ensuring that no integer points are introduced. These cases can be grouped into three classes, those where the designated constraint is adjacent to an inequality (Section 4.2), those where it is adjacent to an equality (Section 4.3) and those where one disjunct sticks out of the other by at most one (Section 4.4). Figure 1 groups the different cases. Table 1 lists the number of times each case is applied while running the test suites of `isl` [16], `pet` [19] and `PPCG` [17]. Note that the subset case does not include those instances where two disjuncts are described by exactly the same constraints. These instances are removed through other means. Although Table 1 is mainly meant to show the relative occurrence of the different cases, for completeness it also shows the total number of pairs of disjuncts considered during coalescing.

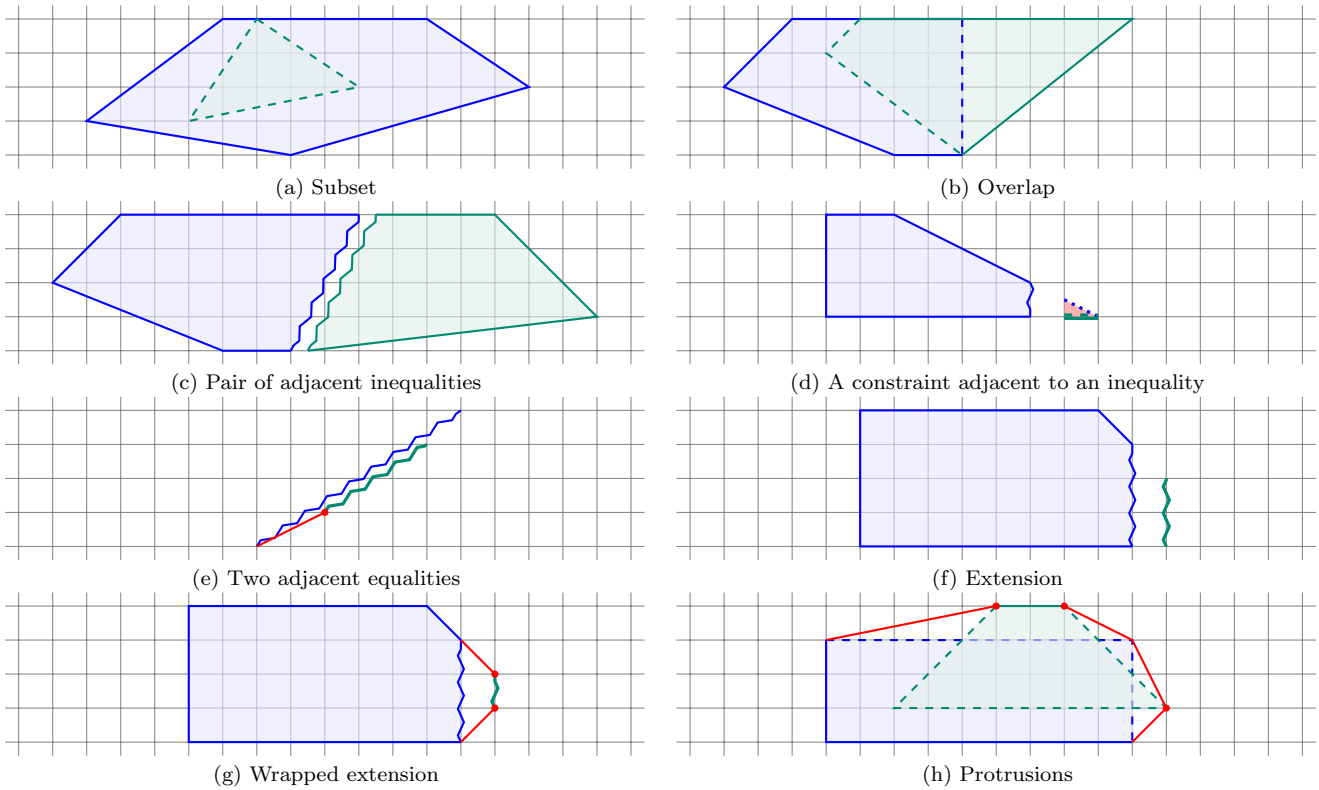
	`isl`	`pet`	`PPCG`
Subset	151	448	645
Overlap	18	33	22
Pair of adjacent inequalities	27	4155	190
Adjacent to an inequality	4	17	0
Two adjacent equalities	28	90	39
Extension	57	386	287
Wrapped extension	12	24	36
Protrusions	16	25	101
All pairs	1448	21250	3420

Table 1: Application incidence in test suite

4.1 Rational Cases

The simplest case is one where all constraints of disjunct A are valid for disjunct B. An example is shown in Figure 1a, where valid constraints are drawn as solid lines and cut constraints as dashed lines. In this case, disjunct B can be dropped since it is a subset of A. This case applies to both rational and integer sets, although in the integer case, the rational approximation of B is allowed to slightly stick out of A due to the relaxed definition of valid constraints.

A second case that also applies to rational sets is one where both disjuncts have cut constraints, but the corresponding facets lie entirely inside the other disjunct. This means that there is no way to move out of the union of the two disjuncts through a cut constraint, which in turns means that the union can be replaced by the valid constraints of the two disjuncts. Note that this also implies that there are no separating constraints. An example is shown in Figure 1b. This case has been described before [3], where it is detected by setting up LP problems for each pair of cut constraints of the two disjuncts. Each LP problem is bounded by the valid constraints of both disjuncts and a constraint $t_i(\mathbf{x}) + \epsilon_i \leq 0$ for each of the two cut constraints $t_i(\mathbf{x}) \geq 0$. If the maximal value of the two ϵ_i is strictly greater than zero, then it is possible to move outside of the pair of cut constraints inside the valid constraints of both disjuncts and the case does not



(a) Subset (b) Overlap

(c) Pair of adjacent inequalities (d) A constraint adjacent to an inequality

(e) Two adjacent equalities (f) Extension

(g) Wrapped extension (h) Protrusions

Figure 1: Coalescing Patterns

apply.

In `isl`, a different approach is taken to detect this case. Instead of checking if it is possible to move outside a pair of cut constraints, the approach checks if all the cut facets (i.e., the intersections of the cut constraint with the disjunct) lie inside the other disjunct. In particular, for each cut constraint $t(\mathbf{x}) \geq 0$ of A, the constraint $-t(\mathbf{x}) \geq 0$ is (temporarily) added to the tableau of A and marked as an equality. Then the procedure checks whether all cut constraints of B are valid in the result, using the (more restrictive) rational criterion of validity. Note that the valid constraints of B do not need to be checked since they are definitely valid for a subset of A. If this is the case for all cut facets of A, then the union of A and B is replaced by a set C bounded by the valid constraints of A and B. Note that the number of LP problems that need to be solved is the same as in the procedure of [3], but that the tableau of A can be reused and that the dimension of the LP problem is reduced by one due to the extra equality. On the other hand, if coalescing can be applied, then all LP problems in the procedure of [3] are empty, such that the only cost is in setting up the tableau.

It is clear that $A \cup B$ is a subset of C because all constraints of C are valid for both disjuncts. To see that C is also a subset of $A \cup B$, take any point $\mathbf{x}_1 \in C \setminus (A \cup B)$. A line connecting $\mathbf{x}_1$ with an element of $A \cup B$ meets a facet F of either A or B. Assume it is a facet of B and let $t_1(\mathbf{x}) \geq 0$ be the corresponding facet constraint. We have $t_1(\mathbf{x}_1) < 0$ and so $t_1(\mathbf{x}) \geq 0$ is a cut constraint. This means that there is some (rational) point $\mathbf{x}_2$ satisfying the constraints of A as well as the constraint $t_1(\mathbf{x}) < 0$ as otherwise the constraint $t_1(\mathbf{x}) \geq 0$ would have been marked valid for A. The line connecting $\mathbf{x}_1$ and $\mathbf{x}_2$ meets a facet of A in a (rational) point also violating $t_1(\mathbf{x}) \geq 0$, but this is impossible since all cut constraints of B are valid for all cut facets of A. To motivate the assumption that the facet F is a facet of B, assume now that it is a facet of A instead. The above reasoning can then be followed to find a facet of B separating $\mathbf{x}_1$ from $A \cup B$. This completes the proof.

4.2 Constraints adjacent to inequalities

This section describes cases where the input disjuncts can be seen as the result of cutting the single output disjunct along a hyperplane. Since we are dealing with integer sets, this means that one disjunct has an inequality $t(\mathbf{x}) \geq 0$, while the other has an inequality $-t(\mathbf{x}) - 1 \geq 0$. That is, they have adjacent inequalities.

In the purest case, both disjuncts have only valid constraints except for one constraint that is adjacent to an inequality of the other disjunct. An example is shown in Figure 1c, where the separating constraints are drawn as zigzag lines. Let $t(\mathbf{x}) \geq 0$ be the constraint of A that is adjacent to an inequality of B and let C be the set bounded by the valid constraints of A and B. We have $C \cap \{\, \mathbf{x} : t(\mathbf{x}) \geq 0 \,\} = A$ since $t(\mathbf{x}) \geq 0$ is the only constraint of A that does not appear among the constraints of C and since the other constraints of C are valid for A. Similarly, $C \cap \{\, \mathbf{x} : -t(\mathbf{x}) - 1 \geq 0 \,\} = B$. There can be no integer points where $t(\mathbf{x})$ attains a value strictly between 0 and -1 because it has integer coefficients. As a result, $C = A \cup B$ and the two disjuncts can be replaced by C.

There are also cases where cutting a set along a hyperplane results in one part satisfying an additional equality in

another direction. An example is shown in Figure 1d. The dotted line is a continuation of a constraint from the left (blue) disjunct that has been simplified away in the right (green) disjunct because the integer points in this right part satisfy an additional equality. The test for this case starts by checking that disjunct A has only valid constraints, except for one constraint $t(\mathbf{x}) \geq 0$ that is adjacent to an inequality of B. No such restrictions are imposed on the constraints of B. Instead, the constraint $t(\mathbf{x}) \geq 0$ is (temporarily) removed from the tableau for A and replaced by the constraint $-t(\mathbf{x}) - 1 \geq 0$. Furthermore, all valid constraints of B are also added to the tableau. This results in a tableau for a set A'. If all constraints of B that are not valid for A check out to be valid for A', then the two disjuncts are again replaced by a set C bounded by the valid constraints of A and B. Clearly $C \cap \{\, \mathbf{x} : t(\mathbf{x}) \geq 0 \,\} = A$ since $t(\mathbf{x}) \geq 0$ is again the only constraint of A that does not appear among the constraints of C and since the other constraints of C are valid for A. On the other hand $C \cap \{\, \mathbf{x} : -t(\mathbf{x}) - 1 \geq 0 \,\} = A'$ by construction and we already know that $A' \subseteq B$ since the procedure explicitly checks for this condition. We also have $A' \supseteq B$ since all constraints of C are valid for B by construction while the constraint $-t(\mathbf{x}) - 1 \geq 0$ is valid for B because the constraint $t(\mathbf{x}) \geq 0$ separates A from B, i.e., $B \cap \{\, \mathbf{x} : t(\mathbf{x}) \geq 0 \,\} = \emptyset$.

4.3 Constraints adjacent to equalities

The cases in this section are similar to those of the previous section, in the sense that again the inputs can be seen as the result of cutting the output along a hyperplane, except that now one or both of the disjuncts satisfy the hyperplane or its opposite as an equality.

Let us first consider the case where both disjuncts satisfy such an equality. This case can be recognized as a constraint $t(\mathbf{x}) \geq 0$ that forms half of an equality constraint of A having been typed as being adjacent to B. The other constraints of A and B may be represented in different ways since adding $t(\mathbf{x})$ to the constraints of A or adding $t(\mathbf{x}) + 1$ to the constraints of B does not have any effect. Some of these constraints may therefore happen to be valid for the other disjunct whereas others may not. The invalid constraints of A (except $t(\mathbf{x}) \geq 0$) are then wrapped around the constraint $-t(\mathbf{x}) \geq 0$ to include B, while the invalid constraints of B (except $-t(\mathbf{x}) - 1 \geq 0$) are wrapped around the constraint $t(\mathbf{x}) + 1 \geq 0$ to include A. Figure 1e shows two such disjuncts (in zigzag lines) as well as a (red) constraint of the right (green) disjunct that has been wrapped to include the left (blue) disjunct. If this case is discovered and all wrappings are successful, i.e., there are no unboundedness issues, then the union is replaced by a set C bounded by the valid constraints of A and B along with all wrapped constraints and the constraint $t(\mathbf{x}) + 1 \geq 0$. The total number of constraints does not increase since each wrapped constraint that is added is obtained from an invalid constraint (which is removed), while the removal of $-t(\mathbf{x}) \geq 0$ (for which not wrapped constraint is added) compensates the addition of $t(\mathbf{x}) + 1 \geq 0$.

It should be clear that C contains both A and B since all its constraints are valid for both A and B. Furthermore, $-t(\mathbf{x}) \geq 0$ (a constraint of A valid for B) and $t(\mathbf{x}) + 1 \geq 0$ (an explicitly added constraint) are valid for C, meaning that $t(\mathbf{x})$ can only attain the values 0 and -1 over C. In particular $C \cap \{\, \mathbf{x} : t(\mathbf{x}) = 0 \,\} = A$ since each constraint $q(\mathbf{x}) \geq 0$ of A is either

- $t(\mathbf{x})$ or $-t(\mathbf{x})$ (which are covered by $t(\mathbf{x}) = 0$),
- valid for B, or,
- replaced by a wrapping constraint of the form $-n\,t(\mathbf{x}) + d\,q(\mathbf{x}) \geq 0$ with $d > 0$. Plugging in $t = 0$ in these wrapping constraints yields the original $q(\mathbf{x}) \geq 0$ constraints.

Similarly, $C \cap \{\, \mathbf{x} : t(\mathbf{x}) = -1 \,\} = B$ proving that $C = A \cup B$.

Let us now consider the case where only one of the disjuncts satisfies an equality along the hyperplane that separates the two disjuncts. In particular, let A have an inequality $t(\mathbf{x}) \geq 0$ that is adjacent to an equality of B and no other separating constraints. Two subcases can be considered. In the first subcase, A is further assumed not to have any cut constraints. Let A' be the result of relaxing constraint $t(\mathbf{x}) + 1 \geq 0$ and A'' the result of enforcing the constraint $t(\mathbf{x}) + 1 = 0$ on A'. If A'' is a subset of B, i.e., if all constraints of B are valid for A'', then $A \cup B$ can be replaced by A'. In particular, A'' contains B since A has no cut constraints and B satisfies $t(\mathbf{x}) + 1 = 0$. In other words $A'' = B$ and so $A' = A \cup A'' = A \cup B$. An example of this subcase is shown in Figure 1f.

Example 4.1. *Continued from Example 3.1. After simplification, the constraints of the first disjunct are of the form* $\{\, (x, y) : 2x = 3y \wedge 0 \leq x - y \leq 2 \,\}$. *We have already seen that the constraint* $x - y \geq 0$ *is adjacent to an equality of the other disjunct. Relaxing this constraint yields* $A' = \{\, (x, y) : 2x = 3y \wedge -1 \leq x - y \leq 2 \,\}$. *Enforcing* $x - y = -1$ *yields* $A'' = \{\, (x, y) : 2x = 3y \wedge -1 = x - y \,\} = \{\, (-3, -2) \,\}$. *This is a subset of the second disjunct and so the two disjuncts can be replaced by* A'.

In the second subcase, B is not a pure extension of A, but can still be included through wrapping. In particular, the constraints of B that are not valid for A (except $-t(\mathbf{x}) - 1 \geq 0$) are wrapped around the constraint $t(\mathbf{x}) + 1 \geq 0$ to include A. Figure 1g shows an example where wrapping is successful, including the two (red) wrapped constraints. In this second subcase, A is allowed to have cut constraints, which are then wrapped around $-t(\mathbf{x}) \geq 0$ to include B. These wrapped constraints may however not be valid for A, so this property needs to be verified explicitly. In fact, these wrapped necessarily cut off some rational points, but they may leave all the integer points unharmed. Figure 2 illustrates this phenomenon. The left (blue) disjunct has a constraint adjacent to an equality of the right (green) disjunct as well as a constraint that cuts the right disjunct. Wrapping this cut constraint to include the right disjunct results in the dotted (red) line. This wrapped constraint cuts off rational points from the left disjunct, but no integer points. In particular, the minimum value of the constraint expression is strictly greater than -1, so the constraint is still recognized as a valid constraint for the left disjunct.

If all the wrappings are successful and the wrapped cut constraints of A are all valid for A, then the two disjuncts are replaced by a set C bounded by the valid constraints of A and B, the constraint $t(\mathbf{x}) + 1 \geq 0$ and all wrapped constraints. We have $C \cap \{\, \mathbf{x} : t(\mathbf{x}) \geq 0 \,\} = A$ since all constraints that are not valid for B (except $t(\mathbf{x}) \geq 0$) are replaced by stricter constraints that still include the integer

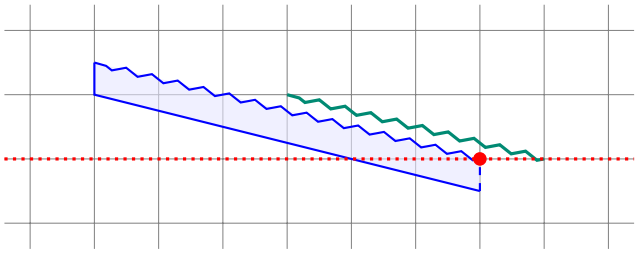


Figure 2: Wrapped extension with cuts

points of A while all constraints that are valid for B are included in the constraints of C. We also have $C \cap \{ \mathbf{x} : -t(\mathbf{x}) - 1 \geq 0 \} = C \cap \{ \mathbf{x} : -t(\mathbf{x}) - 1 = 0 \} = B$ for reasons similar to the case of two adjacent equalities. The fact that the number of constraints does not increase also follows from a similar reasoning.

4.4 Protrusions

The final case is one where one of the disjuncts sticks out of the other disjunct by less than two. An example is shown in Figure 1h, along with the wrapping constraints that are used to construct a single disjunct representation of the union. If the other disjunct were subtracted from the disjunct that is sticking out, then the resulting pieces would lie on equalities adjacent to the other disjunct and the corresponding heuristics could be used to try and coalesce in the pieces. Of course, it is better to try and coalesce the input disjuncts directly, which is what this section describes.

Let A be a disjunct with only valid constraints and cut constraints for B such that the minimum of each constraint expression over B is strictly greater than -2. For each cut constraint $t(\mathbf{x}) \geq 0$ of A, construct $B' = B \cap \{ \mathbf{x} : t(\mathbf{x}) + 1 = 0 \}$. If B' is empty, then reclassify $t(\mathbf{x}) \geq 0$ as a valid constraint. Otherwise, consider the constraints of B that are not valid for A and that are not redundant for B'. Wrap each of these constraints around $t(\mathbf{x}) + 1 \geq 0$ to include A. If all the wrappings succeed, then the two disjuncts can be replaced by a single disjunct set C bounded by all valid constraints, the wrapped constraints and the constraints $t(\mathbf{x}) + 1 \geq 0$ for each remaining cut constraint $t(\mathbf{x}) \geq 0$ of A. Note that wrapping can only fail if A extends beyond the constraint without bound. This means in particular that if both disjuncts stick out of the other by less than two, then coalescing will succeed whichever disjunct we take as disjunct A. The resulting wrapped constraints may however depend on this choice if the amount by which one disjunct sticks out of the other is not exactly 1.

To see that $A \cup B \subseteq C$, note that the constraints $t(\mathbf{x}) \geq 0$ that were reclassified as valid constraints are indeed valid for B since $t(\mathbf{x}) > -2$ and $t(\mathbf{x}) \neq -1$ over B. The wrapped constraints are valid for A by construction. They are also valid for B since they are derived from constraints that were valid for B but not for A and are therefore more relaxed (on the $t(\mathbf{x}) + 1 \geq 0$ side) than the original constraints. To see that also $A \cup B \supseteq C$, take any $\mathbf{x}^* \in C \setminus A$. There must be at least one cut constraint of A, say $t(\mathbf{x}) \geq 0$, violated by $\mathbf{x}^*$. Since the constraints of C include the constraint $t(\mathbf{x}) + 1 \geq 0$, we necessarily have $t(\mathbf{x}^*) = -1$. This means that the corresponding B' was not empty as otherwise the constraint $t(\mathbf{x}) \geq 0$ would have been reclassified as a valid constraint and included in the constraints of C. Using the same reasoning of Section 4.3 applied to A and B' we find that $C \cap \{ \mathbf{x} : t(\mathbf{x}) + 1 \} \subseteq B'$. In other words $\mathbf{x}^* \in B' \subseteq B$, establishing the result.

Unlike all the other coalescing cases, the number of constraints may grow here since the same invalid constraint of B may stick out of several cut constraints of A, as in Figure 1h. In practice, though, the number of constraints effectively increases only in very rare cases since several wrapped constraints may coincide and since the $t(\mathbf{x}) + 1 \geq 0$ constraints are redundant in practically all cases. Still, the implementation checks that the final number of constraints does not increase and opts out of the coalescing if it does. So far, this has never happened in practice.

EXAMPLE 4.2. *Consider the set* $\{ (x, y) : (x \geq 0 \wedge y \geq 2 \wedge y \geq x + 1) \vee (x \geq 1 \wedge y \geq x) \}$. *Taking the first disjunct as A, we see that the other disjunct (B) sticks out of the constraints $y \geq 2$ and $y \geq x + 1$ by 1, while $x \geq 1$ is the only constraint of B that is invalid for A. Wrapping $x \geq 1$ around $y \geq 1$ yields $x + y \geq 2$, while wrapping it around $y \geq x$ also yields $x + y \geq 2$. Furthermore, the added constraints $y \geq 1$ and $y \geq x$ are redundant. The final result is* $\{ (x, y) : x \geq 0 \wedge y \geq x \wedge x + y \geq 2 \}$. *Note that in this particular example, we could also take the second disjunct as the A disjunct. In this case, the corresponding B only sticks out a single constraint of A ($x \geq 1$). The final result is the same.*

5. QUANTIFIED VARIABLES

So far, we have assumed that the two disjuncts that need to be checked for coalescing have the same number of variables. This makes sense since we would only want to coalesce sets that live in the same space, while the symbolic constants that the constraints may refer to can be aligned over all disjuncts of the sets and then treated as variables for the purpose of coalescing.

If there are any existentially quantified variables or integer divisions, however, then their total number may be different over the two disjuncts. Even if their numbers are the same, coalescing may only be possible for some matchings of those variables across the two disjuncts. The `isl` implementation takes a pragmatic approach. First of all, the integer divisions in both disjuncts are sorted. Moreover, they are "harmonized" across the two disjuncts in the sense that if one disjunct has an integer division $\lfloor f(\mathbf{x})/d \rfloor$, while the other has an integer division $\lfloor (f(\mathbf{x}) + n\,d)/d \rfloor$, then the second is replaced by $\lfloor f(\mathbf{x})/d \rfloor + n$.

EXAMPLE 5.1. *As a simple example, take the set* $\{ (x) : (\exists \alpha : x - 1 \leq 3\alpha \leq x) \vee (\exists \alpha : x - 2 = 3\alpha) \}$. *Internally, this representation is automatically converted into* $\{ (x) : (x - 1 \leq 3 \lfloor (x + 1)/3 \rfloor \leq x) \vee (x - 2 = 3 \lfloor (x - 2)/3 \rfloor) \}$. *Since $\lfloor (x + 1)/3 \rfloor$ is equal to $\lfloor ((x - 2) + 3)/3 \rfloor$, it is replaced by $\lfloor (x - 2)/3 \rfloor + 1$, resulting in* $\{ (x) : (x - 4 \leq 3 \lfloor (x - 2)/3 \rfloor \leq x - 3) \vee (x - 2 = 3 \lfloor (x - 2)/3 \rfloor) \}$. *This input matches the extension pattern, yielding* $\{ (x) : x - 4 \leq 3 \lfloor (x - 2)/3 \rfloor \leq x - 2 \}$. *The standard simplification mechanism in `isl` then notices that all constraints that involve the integer division $\lfloor (x - 2)/3 \rfloor$ are implied by the meaning of this integer division, resulting in these constraints and the integer division being removed. The final result is* $\{ (x) \}$.

If both disjuncts only have integer divisions (and no uneliminated existentially quantified variables) and if these integer divisions are the same for both disjuncts, then all cases

of Section 4 are considered. Otherwise, the following four options are considered until one of them succeeds.

- If the total number of integer divisions and existentially quantified variables is the same for both disjuncts, then all cases are considered on the given order of these expressions and variables. Note that this is unlikely to succeed unless the expressions and variables in the same positions are the same or sufficiently similar. However, failure should be discovered fairly quickly. Sufficiently similar integer divisions are for example $\lfloor x/2 \rfloor$ and $\lfloor (x+1)/2 \rfloor$. Sufficiently similar existentially quantified variables are those that roughly attain the same values.

  EXAMPLE 5.2. *Consider the set* $\{ (x) : (\exists j : i = 4j \wedge 0 \leq i \leq 100) \vee (\exists j : 4j+1 \leq i \leq 4j+2 \wedge 0 \leq i \leq 100) \}$. *Identifying the two existentially quantified variables in the two disjuncts, we see that the first disjunct is an extension of the second disjunct, yielding the coalesced result* $\{ (\exists j : 4j \leq i \leq 4j+2 \wedge 0 \leq i \leq 100) \}$. *Note that before this input can even reach the coalescing stage,* `isl` *would have automatically eliminated the existentially quantified variables, but it would have replaced them by two distinct integer divisions (*$\lfloor i/4 \rfloor$ *and* $\lfloor (i+1)/4 \rfloor$*), so the same case applies.*

- If one of the disjuncts has only integer divisions, which moreover form a subset of the integer divisions of the other disjunct, then this other disjunct is checked for being a subset of the first disjunct. This test only requires typing of constraints of the first disjunct, which have zeros inserted at the positions of the other integer divisions and existentially quantified variables of the other disjunct.

  EXAMPLE 5.3. *As a simple example, consider the set* $\{ (x) : (0 \leq x \leq 100) \vee (\exists \alpha : 5 \leq x \leq 40 \wedge x = 5\alpha) \}$. *The total number of integer divisions and existentially quantified variables is not the same in the two disjuncts. However, the first has no uneliminated existentially quantified variables. Moreover, inserting an extra variable in the first disjunct corresponding to the existentially quantified variable* α *in the second yields two constraints that are valid for the second disjunct. This second disjunct is therefore a subset of the first and can be removed.*

- If one of the disjuncts has only integer divisions, which do not initially form a subset of the integer divisions of the other disjunct, but where the additional integer divisions can be simplified away using the equalities of the other disjunct, then this other disjunct is checked for being a subset of the first disjunct restricted to the subset that satisfies these equalities. If this is the case, then the second disjunct is necessarily also a subset of the first disjunct. As in the previous case, this test only requires typing of constraints of the first disjunct, which have zeros inserted at the positions of the other integer divisions and existentially quantified variables of the other disjunct.

  EXAMPLE 5.4. *Consider the following set:* $\{ (x, y) : (x = 2 \lfloor x/2 \rfloor \wedge y = 2 \lfloor y/2 \rfloor) \vee (x = 2 \lfloor x/2 \rfloor \wedge y = x) \}$. *The first disjunct involves two integer divisions, while the second has only one. However, the second disjunct satisfies an additional equality* $(y = x)$*, which can be used to simplify the integer division* $\lfloor y/2 \rfloor$ *to* $\lfloor x/2 \rfloor$*, which is equal to the other integer division. After adding the equality, the first disjunct therefore has the same integer divisions as the second. Moreover, the constraints of the first disjunct are satisfied by the second disjunct, which can then be removed.*

- If both disjuncts have only integer divisions that do not themselves refer to any nested integer divisions, and if, moreover, those of disjunct B form a subset of those of disjunct A while the additional integer divisions of A can be simplified away using the equalities of B, then those additional integer divisions are added to B and all cases are considered. Note that the absence of nested integer divisions requirement is only imposed to ease the comparison of the integer divisions in the two disjuncts.

  EXAMPLE 5.5. *Consider the following set:* $\{ (x) : (x = 2 \lfloor x/2 \rfloor \wedge 0 \leq x \leq 10) \vee x = 12 \}$. *The integer division* $\lfloor x/2 \rfloor$ *can be introduced into the second disjunct with value* 6 *which is then coalesced into the first as an extension.*

6. MULTIPLE DISJUNCTS

If there are more than two disjuncts then each pair of disjuncts is compared in turn. In particular, each disjunct is considered in turn and compared to all the previously considered disjuncts. If any pair of disjuncts can be replaced by a single disjunct then this single disjunct is compared again against all previously considered disjuncts. There is one twist to this scheme and that is that disjuncts that live in the same affine space are first compared against each other before being compared to disjuncts living in other affine spaces. In particular, disjuncts that live in two adjacent hyperplanes are first combined within the hyperplanes before they are combined across the hyperplanes. The reason is that in principle any two disjuncts that live in adjacent hyperplanes can be coalesced together. First combining disjuncts inside a hyperplane therefore does not affect the potential for coalescing across the hyperplanes. On the other hand, first coalescing across hyperplanes results in a disjunct that no longer lives entirely within one of the hyperplanes. Although it may still be possible to combine this disjunct with other disjuncts inside the two hyperplanes, this is no longer guaranteed. Note that both the fact that only pairs of disjuncts are ever compared and the order in which the disjuncts are treated may result in missed coalescing opportunities.

7. HISTORY AND RELATED WORK

The coalesce operation in `isl` was initially introduced to allow `isl` to be used as a replacement for `PolyLib` as a `CLooG` backend, where it was used as an alternative for the convex hull based approach of the introduction. It was subsequently used in an equivalence checker [20] and by the time of the first release of `isl`, the subset, overlap, pair of adjacent inequalities and extension cases were supported. This version was presented informally (without proofs) at the AMS 2009 Spring Western Section Meeting [15]. Version 0.02 gained support for wrapped extensions and protrusions, i.e., cases

that require wrapping, version 0.05 for pairs of adjacent equalities and 0.12 for a constraint adjacent to an inequality. Version 0.15 will include the use of variable compression and the more advanced support for existentially quantified variables and integer divisions described in this paper.

Convexity recognition [3] for rational sets considers the same cases as those in Section 4.1, but uses a somewhat different algorithm. Exact join detection [1] appears to have been developed in parallel with `isl` coalescing. Like convexity recognition, it also only deals with rational sets and is based on the double description of polyhedra, which `isl` tries to avoid. As explained in the introduction, `Omega` [9] checks for each disjunct whether it is a subset of any other disjunct and it does so in a way that is more general, but possibly also more expensive, than the subset case of coalescing. The `CLooG` paper [2] mentions an "unisolate" operation, which may be a special case of coalescing, but no details are provided on how this operation works and no implementation was ever made publicly available.

8. CONCLUSIONS AND FUTURE WORK

When representing integer sets in disjunctive normal form, it is important to keep the number of disjuncts low, both for efficiency of the computations and for the quality of the final result. However, the reduction of disjuncts should not be performed at all cost. In particular, the reduction itself should not take too much time and should not introduce too many extra constraints or constraints with large coefficients. The integer set coalescing operation of `isl` uses LP techniques to detect and exploit a number of patterns, without increasing the number of constraints and (by default) without introducing large coefficients. It may not be able to discover all coalescing opportunities, but the discovery is performed in a reasonably efficient way and the results are proven to be correct.

It may be interesting to consider automatically performing coalescing more frequently. It is currently only used during the computation of transitive closures, during AST generation and to simplify the input of the scheduler. It could be useful to also apply it after every projection operation. It may also be interesting to try and coalesce more than two disjuncts at once, assuming such cases occur in practice.

Acknowledgements

This research was partly supported by FWO-Vlaanderen, project G.0232.06N; the European FP7 project CARP id. 287767 and the COPCAMS ARTEMIS project.

9. REFERENCES

[1] R. Bagnara, P. Hill, and E. Zaffanella. Exact join detection for convex polyhedra and other numerical abstractions. *Comput. Geom. Theory Appl.*, 43(5):453–473, 2010.

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

[3] A. Bemporad, K. Fukuda, and F. D. Torrisi. Convexity recognition of the union of polyhedra. *Comput. Geom.*, 18(3):141–154, 2001.

[4] C. Chen. Polyhedra scanning revisited. *SIGPLAN Not.*, 47(6):499–508, June 2012.

[5] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[6] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[7] K. Fukuda, T. M. Liebling, and C. Lütolf. Extended convex hull. In *Proc. of the 12th Canadian Conference on Computational Geometry*, pages 57–63, 2000.

[8] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for GPUs. In *Proc. of Annual IEEE/ACM Int. Symp. on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[9] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega library. Technical report, University of Maryland, Nov. 1996.

[10] W. Kelly, W. Pugh, E. Rosser, and T. Shpeisman. Transitive closure of infinite graphs and its applications. *Int. J. Parallel Program.*, 24(6):579–598, 1996.

[11] V. Loechner and D. K. Wilde. Parameterized polyhedra and their vertices. *Int. Journal of Parallel Programming*, 25(6):525–549, Dec. 1997.

[12] B. Meister. *Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization*. PhD thesis, Université Louis Pasteur, Dec. 2004.

[13] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Program. Lang. Syst.*, 16:1248–1278, July 1994.

[14] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1986.

[15] S. Verdoolaege. An integer set library for program analysis. Advances in the Theory of Integer Linear Optimization and its Extensions, AMS 2009 Spring Western Section Meeting, San Francisco, California, 25-26 April 2009, Apr. 2009.

[16] S. Verdoolaege. isl: An integer set library for the polyhedral model. In K. Fukuda, J. Hoeven, M. Joswig, and N. Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *Lecture Notes in Computer Science*, pages 299–302. Springer, 2010.

[17] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for CUDA. *ACM TACO*, 9(4):54, 2013.

[18] S. Verdoolaege, A. Cohen, and A. Beletska. Transitive closures of affine integer tuple relations and their overapproximations. In *Proc. of the 18th Int. Conference on Static Analysis*, SAS'11, pages 216–232, Berlin, Heidelberg, 2011. Springer-Verlag.

[19] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Second Int. Wksp on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

[20] S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification 21*, pages 599–613. Springer, June 2009.

[21] D. K. Wilde. A library for doing polyhedral operations Technical Report 785, IRISA, Rennes, France, 1993.