Counting Affine Calculator and Applications

Sven Verdoolaege
INRIA Saclay — Île-de-France
Parc Club Orsay Universite, ZAC des vignes
4 rue Jacques Monod, 91893 Orsay, France
sven.verdoolaege@inria.fr

ABSTRACT

We present an interactive tool, called iscc, for manipulating sets and relations of integer tuples bounded by affine constraints over the set variables, parameters and existentially quantified variables. A distinguishing feature of iscc is that it provides a cardinality operation on sets and relations that computes a symbolic expression in terms of the parameters and domain variables for the number of elements in the set or the image of the relation. In particular, these expressions are piecewise quasipolynomials, which can be further manipulated in iscc. Besides basic operations on sets and piecewise quasipolynomials, iscc also provides an interface to code generation, lexicographic optimization, dependence analysis, transitive closures and the symbolic computation of upper bounds and sums of piecewise quasipolynomials over their domains.

1. INTRODUCTION

The polyhedral model [8] is a powerful formalism for analyzing and transforming program fragments that meet certain requirements. In particular, all loop bounds and all conditions should be (approximated by) affine expressions in the loop iterators and symbolic constants called parameters. Depending on the type of analysis performed, the array index expressions may also have to be affine. Several libraries have been developed for operating on representations in this polyhedral model, including `PolyLib` [14, 18], `PipLib` [9], `Omega` [6,12], `PPL` [2], `CLooG` [3], `barvinok` [16,17] and `isl` [15], and these libraries have been used in numerous applications. During experimentation and prototyping of new polyhedral techniques, however, there is a need for an interactive tool such as `iscc`, providing a uniform, high-level interface to some of these polyhedral libraries. The main contributions of `iscc` and this paper are

- a uniform interface to `isl` (for representation and manipulation of sets and relations), `CLooG` (for code generation) and `barvinok` (for counting).
- support for sets and relations containing elements from more than one space
- support for structured and named spaces
- compositions of relations on the one hand and counting or bounding functions on the other hand

The tool is distributed as part of the `barvinok` distribution, available from `http://freshmeat.net/projects/barvinok/`. The tool itself is a very thin layer on top of the underlying libraries. Support for the above contributions has therefore also been made available in these libraries themselves. Although `iscc` is primarily designed for use in the context of the polyhedral model, most of the supported operations take abstract integer sets or related objects as input and should therefore also be useful outside this context.

```
for (i = 0; i < N; ++i)
S1: t[i] = f(a[i]);
for (i = 0; i < N; ++i)
S2: b[i] = g(t[N-i-1]);
```

Figure 1: Simple program with temporary array `t`

2. SYNTAX

The language used by `iscc` is extremely simple. It supports operations on constants and dynamically typed variables and assignments to those variables. If the result of an expression is not used inside another expression and is not assigned to a variable, then it is printed on the screen. The operators are overloaded based on the types of the arguments, which may be sets, relations, piecewise quasipolynomials, piecewise quasipolynomial folds, lists, integers, strings or booleans.

A set contains integer tuples that satisfy some Presburger formula built from affine constraints, conjunctions (`and`), disjunctions (`or`), projections (`exists`) and negations (`not`). Note that the formula is immediately converted into disjunctive normal form, so it may sometimes be more efficient to construct a set from smaller sets by applying basic operations such as intersection (`*`), union (`+`) and difference (`-`). The description may involve (free) parameters and existentially quantified variables. We will call set descriptions involving existentially quantified variables *quasi-affine*. A set may also contain elements from different spaces, i.e., spaces with different dimensionalities, names or internal structures. For example, the set

```
[N] -> { S1[i] : 0 <= i < N; S2[i] : 0 <= i < N }
```

represents the iteration domains of the statements in the program in Figure 1. The optional parameters should be declared by placing them in a comma delimited list inside `[]` (followed by an `->`) in front of the main set description. The parameters are global and are identified by their names, so the order inside the list is arbitrary. The main set description consists of a `{}`-pair containing a semicolon delimited

list of subsets from a given space. A space is identified by its optional name (`S1` and `S2` in the example) and its dimension (the number of coordinates in the `[]` enclosed tuple following the optional name). The names of the coordinates are purely local within the set description. The space specification is separated by a colon from the actual constraints describing the elements from that space. The disjuncts are delimited by `or`, while the conjuncts are delimited by `and`. Each disjunct may be preceded by `exists` followed by a list of existentially quantified variables and a colon.

Relations are binary relations on pairs of spaces and are defined in a similar way, except that the single space is replaced by a pair of spaces separated by `->`. Relations need not be (single-valued) functions. As an example of a relation, the read access relations of the program in Figure 1 can be represented as

```
[N] -> { S1[i] -> a[i]; S2[i] -> t[N-i-1] }
```

while the write access relations can be represented as

```
{ S1[i] -> t[i]; S2[i] -> b[i] }
```

Note that for reasons of brevity we did not include the constraints on the iterators. Assuming we assigned the set of iteration domains to the variable `D`, we do not need to specify them explicitly in the description of the access relations, but we can instead intersect the domain of the relation with `D`, as in

```
W := { S1[i] -> t[i]; S2[i] -> b[i] } * D;
```

Finally, relations can be embedded in a set, by placing the pair of spaces of the relation inside an extra `[]`-pair. The resulting structured space can then in turn be used as a domain or codomain of a relation and this process can be continued arbitrarily. As a typical example, consider the following relation, which maps an iteration space to the write access relation corresponding to that space.

```
{ S1[i] -> [S1[i]->t[i]]; S2[i] -> [S2[i]->b[i]] }
```

When such a relation is applied to a set of one or more statement iterations, the result will be an embedded access relation for those iterations.

3. BASIC OPERATIONS

Any given integer set can be represented in many different ways. Most of the operations supported by `iscc` are independent of the representation. That is, they will produce the same result whatever representation is chosen, although, of course the *representation* of the result may depend on the representation of the input. The main exceptions are operations that involve some form of approximation such as the polyhedral hull (`poly`), the transitive closure (`^+`) and the computation of an upper bound (`ub`) on a piecewise quasipolynomial over all elements in its domain. Some of these operations will be explained below.

Basic operations on sets and relations include the obvious intersection (`*`), union (`+`) and set difference (`-`). Note that, as illustrated above, if a relation is intersected with a set (in that order), then it is actually the domain of the relation that is intersected with the set. Pairs of sets or relations can be checked for equality or inclusion using the `=` and `<=` operators. Composition of two relations, `M1` and `M2` is written `M2(M1)` or `M2 after M1`, while the join (the same operation with the arguments reversed) is written `M1 . M2` or `M1 before M2`. The postfix `^-1` operator is used to obtain the inverse of a relation. Domain and range can be obtained using the `domain` and `range` operators. Sometimes, it is convenient to have a relation that, when applied to a relation, performs one of these two operations and this is exactly what the operators `domain_map` and `range_map` provide. The resulting relations map an embedded copy of (part of) the original relation to its domain or range. The `wrap` operation can be used to obtain such an embedding, while the inverse operation is called `unwrap`. That is, if `R` is a relation then `domain R` is the same as `(domain_map R)(wrap R)`.

A sample element can be obtained using the `sample` operation. This is also the main way of checking whether a set or relation is empty. If so, then the operation will return an (explicitly) empty set. Note that the underlying `isl` library will often implicitly check whether a set or relation is empty. The affine hull (`aff`) operation returns a description of all the (explicit and implicit) equalities defining the set. If the input contains elements from different spaces, then the affine hull is computed for each space individually.

The lexicographically maximal [minimal] element of a set can be obtained using the `lexmax` [`lexmin`] operator. When applied to a relation, these operators return the maximal [minimal] image element for each domain element. The `lexmax` operation can therefore be used to perform dependence analysis, but handling the general case is fairly complicated. In particular, if `R` contains the read accesses, `W` the write accesses and `S` the schedule, i.e., a relation between the iteration domains and a common iteration space ordered in the lexicographic order, then the flow dependences could be computed as

```
((lexmax((((range_map R).W^-1) * (domain_map(R) .
    (S >> S))).S)).S^-1)^-1.domain_map(R)
```

We will discuss the `>>` operator used in this expression below, but we will not explain the expression itself in detail because there is an easier way to obtain dependences. Since dependence analysis is so important, it is provided as an operator on the write and read accesses and the schedule. The operator is used as follows

```
(last W before R under S)[0]
```

The first argument specifies the potential sources of the dependences, the second argument the sinks and the third argument the schedule. In case of flow dependences, the potential sources are the write access relations, while the sinks are the read access relations. Note that the dependence analysis operator returns two results, the actual dependences and the subset of sinks for which no corresponding source could be found. The `[0]` postfix operator selects the first of these results. When applied to a schedule corresponding to the program in Figure 1, e.g.,

```
S := { S1[i] -> [0,i]; S2[i] -> [1,i] };
```

and with `W` and `R` set to the write and read access relations, the above call to the dependence analysis operator returns

```
[N] -> { S1[i] -> S2[-1 + N - i] : i <= -1 + N and
     i >= 0 }
```

Given the above dependence relation, dependence distance vectors can be computed by applying the `deltas` operator,

which computes differences between image and domain elements. Of course, we need to apply this operation in a space where the difference makes sense, e.g, the scheduling domain. Assuming `Dep` contains the above dependence relation, we therefore compute

```
deltas (S^-1.Dep.S);
```

which results in

```
[N] -> { [1, i1] : exists (e0 = [(-1 - N + i1)/2]:
     2e0 = -1 - N + i1 and i1 >= 1 - N and i1 <=
    -1 + N) }
```

Notice that the second coordinate in this set is odd. The `[]` in the definition of the existentially quantified variable `e0` denote the greatest integer part.

Code that visits each element of a set in lexicographical order can be obtained by applying the `codegen` operator. Note that the given set may only contain elements from a single space since the lexicographical order is only defined within a given space. When applied to a relation, code is generated for each element in the domain of the relation according to the lexicographic order on the image. We can then check whether the above schedule does indeed correspond to Figure 1 by calling

```
codegen (S * D);
```

which results in

```
if (N >= 1) {
  for (c2=0;c2<=N-1;c2++) {
    S1(c2);
  }
  for (c2=0;c2<=N-1;c2++) {
    S2(c2);
  }
}
```

This code can be transformed by applying a different schedule. For example,

```
S2 := [N] -> { S1[i] -> [i,0]; S2[i] -> [N-i-1,1] };
codegen (S2 * D);
```

which results in

```
if (N >= 1) {
  for (c1=0;c1<=N-1;c1++) {
    S1(c1);
    S2(-c1+N-1);
  }
}
```

To ensure that the above code is semantically equivalent to the original, we need to check that no dependences are violated, i.e., that no statement instance is scheduled before any statement instance on which it depends. In other words, if we construct a relation from statement instances to earlier statement instances, then the intersection of this relation with the dependence relation should be empty. The `>>` operator can be used to construct such a relation. When applied to a pair of sets, it constructs a relation between elements of the first set and elements of the second set such that the first are lexicographically larger than the second. When applied to a pair of relations, the resulting relation is constructed from elements in the domains of the input relations, but the lexicographical order is applied to the corresponding image elements. Since

```
double x[2][10];
int old = 0, new = 1, i, t;
for (t = 0; t < 1000; t++) {
    for (i = 0; i < 10; i++)
        x[new][i] = g(x[old][i]);
    new = (new+1) % 2; old = (old+1) % 2;
}
```

Figure 2: Flip-flop example from [1, Fig. 3]

```
i = 0; j = 0;
while (i <= 100) {
    if (A[i] <= A[j]) {
        i = i + 2; j = j + 1;
    } else
        i = i + 4;
}
```

Figure 3: Example from [10, page 35]

```
(S2 >> S2) * Dep;
```

results in the empty set, the schedule does not violate any dependences.

4. TRANSITIVE CLOSURES

The transitive closure of a relation can be computed using the `^+` postfix operator. Actually, the transitive closure of a quasi-affine relation may not be quasi-affine [13], or, even if it is, it may be too expensive to compute. The operation therefore returns an overapproximation of the transitive closure, along with a boolean that is false when the result may not be exact. That is, if the boolean is true, then the result is known to be exact. The details of the transitive closure algorithm implemented in `isl` will appear in a forthcoming publication. The purpose of this section is then not to show the accuracy or the speed of the algorithm, but rather the *ease* with which it can be used in an application due to the high-level interface of `iscc`.

In particular, we will consider the application of invariant analysis, where invariants on the program variables are computed at different control points. Let us first consider an example from [1], reproduced in Figure 2. The authors consider several variations of essentially interchanging the values of `new` and `old` and the main objective is to show that `new` and `old` always have different values. The variation shown in Figure 2 is one for which the authors are unable to prove this invariant, mainly because they do not support existentially quantified variables. The effect of the loop on the two variables can be represented as

```
T := { [new, old] -> [(new+1) % 2, (old+1) % 2] };
```

Computing `T^+` results in

```
({ [new, old] -> [o0, o1] : exists (e0 = [(-new -
    old - o0 + o1)/2]: 2e0 = -new - old - o0 + o1
    and o0 >= 0 and o0 <= 1 and o1 >= 0 and o1 <=
    1) }, True)
```

showing that in this case the computed transitive closure is exact. Applying this transitive closure to the initial state, i.e., computing `(T^+)({[0,1]})`, we obtain

```
{ [i0, i1] : exists (e0 = [(-1 - i0 + i1)/2]: 2e0
    = -1 - i0 + i1 and i0 >= 0 and i0 <= 1 and i1
    >= 0 and i1 <= 1) }
```

This set describes all reachable states after at least one iteration of the loop. In general, the result would be an overapproximation of the set of reachable states, but in this case we know that the result is exact. Note that we did not have to explicitly take the first element in the result of the transitive closure. When a list is given consisting of a relation and a boolean where simply a relation is expected, the first element of the list is used implicitly. Also note that the set of reachable states is not represented in the simplest way. Although the desired invariant is available, it may not be apparent from this representation. The invariant can be made more explicit by computing the affine hull of the above set, resulting in

```
{ [i0, 1 - i0] }
```

As a second example, take that of [10], reproduced in Figure 3. In this case, we consider three program points: at the start, before the loop and after the loop. The transitions between these points can be described as

```
T := {
    zero[i,j] -> one[0,0];
    one[i,j] -> one[i+4,j] : i <= 100;
    one[i,j] -> one[i+2,j+1] : i <= 100;
    one[i,j] -> two[i,j] : i > 100
};
```

The transitive closure of this transition relation can again be computed exactly. Applying this transitive closure to an arbitrary initial state and taking the union with this initial state, we obtain the set of all reachable states exactly. In particular, the result of

```
Init := { zero[i,j] };
(T^+)(Init) + Init;
```

is

```
{ one[i0, i1] : (2i1 = i0 and i0 >= 2 and i0 <=
    102) or (exists (e0 = [(i0)/2], e1 = [(-i0 + 2
    i1)/4]: 2e0 = i0 and 4e1 = -i0 + 2i1 and i1 >=
     1 and i1 <= 50 and 2i1 <= -4 + i0 and i0 <=
    104)); one[0, 0]; one[i0, 0] : exists (e0 = [(
    i0)/4]: 4e0 = i0 and i0 >= 4 and i0 <= 104);
    two[i0, i1] : (2i1 = i0 and i0 >= 101 and i0
    <= 102) or (exists (e0 = [(i0)/2], e1 = [(-i0
    + 2i1)/4]: 2e0 = i0 and 4e1 = -i0 + 2i1 and i1
     >= 1 and i1 <= 50 and 2i1 <= -4 + i0 and i0
    <= 104 and i0 >= 101)); two[i0, 0] : exists (
    e0 = [(i0)/4]: 4e0 = i0 and i0 >= 101 and i0
    <= 104); zero[i, j] }
```

If we are only interested in affine invariants (rather than quasi-affine), we can apply the `poly` operator to this set and obtain

```
{ one[i0, i1] : i0 <= 104 and 2i1 <= i0 and 2i1 <=
     204 - i0 and i1 >= 0; two[i0, i1] : i0 <= 104
     and i0 >= 102 and 2i1 >= 104 - i0 and 2i1 <=
    204 - i0; zero[i, j] }
```

5. BASIC COUNTING

The `card` operator can be used to compute the cardinality of a set, i.e., the number of elements in the set. If the input set is parametric, then the result of this operation will be a piecewise quasipolynomial in the parameters, where a quasipolynomial is a polynomial expression that may involve greatest integer parts of affine expressions. When the `card` operator is applied to a relation, then what is counted is the number of image elements associated to each domain element.

As a simple application of counting, let us compute (an upper bound on) the minimal number of memory elements required to store the `t`-array in the program of Figure 1. We first need to determine for each array element, when it is live, i.e., when it has been written and still needs to be read. We have already computed the dependences for this program before and the resulting dependence relation is already single-valued, meaning that the dependence relation itself already contains the live ranges for each of the array elements. We assume here that each statement only writes a single value so that this value can be identified by the statement instance that writes the value. In general, any given value may be used several times and to determine the live ranges, we need to compute for each write operation, the last read (according to the schedule) of the value written by that write operation. That is, assuming `Dep` contains the dependence relation, we need to compute

```
LR := (lexmax (Dep . S)) . S^-1;
```

In this example, the result is equal to `Dep`. Now, we need to compute how many values are live at each iteration of the program, i.e., how many elements in `LR` are such that the write occurs strictly before the given iteration and the read occurs at or after the given iteration. To do so, we first compute for each value, the set of iterations in its live range. Then, we take the inverse of the resulting relation and we count the number of image elements (i.e., live ranges) for each domain element (i.e., iteration). That is, we compute

```
LLT := (S << S) * (D -> D);
LGE := (S >>= S) * (D -> D);
After_Write := domain_map(LR) . LLT;
Before_Read := range_map(LR) . LGE;
N_Live := card ((After_Write * Before_Read)^-1);
```

The result is

```
[N] -> { S2[i] -> (N - i) : i <= -1 + N and i >= 0;
    S1[i] -> i : i <= -1 + N and i >= 1 }
```

This means that, as expected, the number of memory elements required increases for each iteration of `S1` and then decreases again for each iteration of `S2`. The `->` operator used in the computations of `LLT` and `LGE` constructs a universe relation between the given domain and codomain.

The final step is to compute an upper bound on this number over all iterations:

```
ub N_Live;
```

The result is

```
([N] -> { max(N) : N >= 2; max(N) : N = 1 }, True)
```

As in the case of the transitive closure, the second element in the result shows whether the first element is exact, i.e.,

whether the result is not just an upper bound but the actual maximum. The result of an upper bound computation is called a *fold* and may in general consist of a piecewise list of quasipolynomials. It may happen that, as above, some pieces have the same associated list. The `coalesce` operation can then sometimes help to simplify the representation. Applying this operation to the fold above results in

```
[N] -> { max(N) : N >= 1 }
```

It should be noted that the above sequence of operations used to compute the number of live elements is independent of the input program. All that is needed is a description of the iteration domains, the read and write access relations and the schedule. We may, for example, use the `S2` schedule instead and then the result is

```
([N] -> { max(1) : N >= 1 }, True)
```

That is, with this schedule, a single memory element is sufficient to store all elements of the `t`-array.

6. WEIGHTED COUNTING

The `card` operator of the previous section simply counts the number of elements in a set. In some applications, however, we would like to assign a weight to each point and compute the sum of all these weights. This can be accomplished using the `sum` operator. As an example, let us assign to each point i in the interval $[0, N]$ the weight i^2, i.e.,

```
F := [N] -> { [i] -> i^2 : 0 <= i <= N };
sum F;
```

The result is

```
[N] -> { (1/6 * N + 1/2 * N^2 + 1/3 * N^3) : N >= 0 }
```

The same result can be obtained by applying the function `F` to its domain, i.e., `F(domain F)`. Applying a piecewise quasipolynomial to a set evaluates the function in each element of the set and returns the sum. Similarly, applying a fold to a set evaluates the fold in each element and then returns a bound on the result. Although it should be clear from its textual representation whether a function is a piecewise quasipolynomial or a fold, it may sometimes be inconvenient to print out the function and then the `typeof` operator can be used instead. As usual, applying a function to a relation means that it is applied to the image elements. If the relation is single-valued, then this composition amounts to a substitution because the sum or the bound is computed over a single element. The application `F(R)` may also be written as `R.F`.

Let us apply these compositions to a technique for estimating the dynamic memory requirements of the Java program in Figure 4. The example is borrowed from [7], while the technique is explained in detail in [5]. Essentially, it is assumed that at the end of each method execution, the memory allocations performed during that execution that are no longer needed are released. In a simplified analysis, we then need to keep track of the amount of allocated memory that is returned by a method and the amount that is captured (i.e., not returned). The memory required for running a method is then the sum of the amount of memory captured and the maximal memory requirement for any nested method invocation.

We start by writing down the iteration domains.

```
void m0(int m) {
    for (c = 0; c < m; c++) {
       m1(c);                          /* S1 */
       B[] m2Arr = m2(2 * m - c); /* S2 */
    }
}
void m1(int k) {
    for (i = 1; i <= k; i++) {
       A a = new A();                  /* S3 */
       B[] dummyArr = m2(i);           /* S4 */
   }
}
B[] m2(int n) {
    B[] arrB = new B[n];               /* S5 */
    for (j = 1; j <= n; j++)
       B b = new B();                  /* S6 */
    return arrB;
}
```

Figure 4: A Java program

```
D := { m0[m] -> S1[c] : 0 <= c < m;
       m0[m] -> S2[c] : 0 <= c < m;
       m1[k] -> S3[i] : 1 <= i <= k;
       m1[k] -> S4[i] : 1 <= i <= k;
       m2[n] -> S5[];
       m2[n] -> S6[j] : 1 <= j <= n };
DM := (domain_map D)^-1;
```

Notice how we separate the method arguments from the local iterators so that we can easily construct a mapping from the method arguments to the combination of method arguments and local iterators in `DM`. This relation can then be composed with functions to compute the sum or a bound over all the iterations of the statements in a method.

The amount of memory returned, captured and required by method `m2` can then be computed as follows.

```
ret_m2 := DM . { [m2[n] -> S5[]] -> n : n >= 0 };
cap_m2 := DM . { [m2[n] -> S6[j]] -> 1 };
req_m2 := cap_m2 + { m2[n] -> max(0) };
```

The single iteration of `S5` allocates n memory elements, which are returned, while each iteration of `S6` allocates one memory element, which is captured. For consistency with the other methods, we add a zero fold to `req_m2` to make sure `req_m2` itself is a fold.

For method `m1`, we first define the parameter bindings for all calls returning memory that is captured by `m1` in `CB_m1`.

```
CB_m1 := { [m1[k] -> S4[i]] -> m2[i] };
ret_m1 := { m1[k] -> 0 };
cap_m1 := DM . ({ [m1[k] -> S3[i]] -> 1 }
                    + (CB_m1 . ret_m2));
req_m1 := cap_m1 + (DM . CB_m1 . req_m2);
```

No memory is returned by the method, while the captured memory consists of the memory allocated in `S3` and the memory returned by the call to `m2`. The composition of `ret_m2` and `req_m2` with `CB_m1` only performs a substitution since `CB_m1` is single-valued. The actual summation and bounding happens during the composition with `DM`.

The computation of the memory requirement of `m0` is very similar. The only new operation is the composition of two

folds in `req_m1 . req_m2`.

```
CB_m0 := { [m0[m] -> S1[c]] -> m1[c];
      [m0[m] -> S2[c]] -> m2[2 * m - c] };
ret_m0 := { m0[m] -> 0 };
cap_m0 := DM . CB_m0 . (ret_m1 + ret_m2);
req_m0 := cap_m0 + (DM . CB_m0 . (req_m1 . req_m2));
```

The final result (`req_m0`) is equal to

```
{ m0[m] -> max((-2 + 2 * m + 2 * m^2), (5/2 * m +
    3/2 * m^2)) : m >= 2; m0[m] -> max((5/2 * m +
    3/2 * m^2)) : m = 1 }
```

7. RELATED WORK AND CONCLUSION

Two related interactive polyhedral tools are the Omega calculator [11] and `SPPoC` [4]. The syntax of `iscc` was very much inspired by that of the Omega calculator. However, the Omega calculator only knows sets and relations. In particular, it does not perform any form of counting. An earlier version of `barvinok` came with a modified version of the Omega calculator that introduced an operation for counting the number of elements in a set, but it would simply print the result and not allow any further manipulations. `SPPoC` does support counting, but only the basic operation of counting the elements in a set. In particular, it does not support weighted counting, nor the computation of upper bounds. It also only supports (single-valued) functions and not generic relations like the Omega calculator and `iscc`. In fact, all variables are treated in the way the Omega calculator and `iscc` would treat parameters. Internally, `SPPoC` uses `PolyLib`, `PipLib` and `Omega` to perform its operations. The `isl` library contains an improved implementation of the operation (essentially `lexmin`) supported by `PipLib`. Barvinok's algorithm, implemented in the `barvinok` library, is usually much faster than the algorithm implemented in `PolyLib` [17].

Furthermore, the ability to work with named and nested spaces and the ability of sets and relations to contain (pairs of) elements from different spaces are not available in the Omega calculator and `SPPoC`. Instead, the user of the Omega calculator would have to "normalize" all spaces to the same dimensionality by adding arbitrarily valued coordinates and manually add an extra coordinate identifying the space. `iscc` not only makes these manipulations redundant, it also allows some operations such as the affine hull and the lexicographic optimum to operate on each of the spaces separately.

8. REFERENCES

[1] C. Ancourt, F. Coelho, and F. Irigoin. A modular static analysis approach to affine loop invariants detection. *Electron. Notes Theor. Comput. Sci.*, 267:3–16, October 2010.

[2] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

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

[4] P. Boulet and X. Redon. SPPoC: manipulation automatique de polyèdres pour la compilation. *Technique et Science Informatiques*, 20(8):1019–1048, 2001.

[5] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *International Symposium on Memory Management*, pages 141–150. ACM, ACM, jun 2008.

[6] C. Chen. Omega+ library, 2009. `http://www.chunchen.info/omega/`.

[7] P. Clauss, F. J. Fernández, D. Garbervetsky, and S. Verdoolaege. Symbolic polynomial maximization over convex sets and its application to memory requirement estimation. ICPS Research Report 06-04, Université Louis Pasteur, Oct. 2006.

[8] P. Feautrier. *The Data Parallel Programming Model*, volume 1132 of *LNCS*, chapter Automatic Parallelization in the Polytope Model, pages 79–100. Springer-Verlag, 1996.

[9] P. Feautrier, J. Collard, and C. Bastoul. Solving systems of affine (in)equalities. Technical report, PRiSM, Versailles University, 2002.

[10] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Université Scientifique et Médicale de Grenoble, Mar. 1979.

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

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

[13] 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.

[14] V. Loechner. PolyLib: A library for manipulating parameterized polyhedra. Technical report, ICPS, Université Louis Pasteur de Strasbourg, France, Mar. 1999.

[15] 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.

[16] S. Verdoolaege and M. Bruynooghe. Algorithms for weighted counting over parametric polytopes: A survey and a practical comparison. In M. Beck and T. Stoll, editors, *The 2008 International Conference on Information Theory and Statistical Learning*, July 2008.

[17] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.

[18] D. K. Wilde. A library for doing polyhedral operations. *International Journal of Parallel, Emergent and Distributed Systems*, 15(3):137–166, 2000.