

♥ Integer maps

Integer maps are binary relations between integer sets.

The first set in the relation is called domain.
The second set is the range or the output.

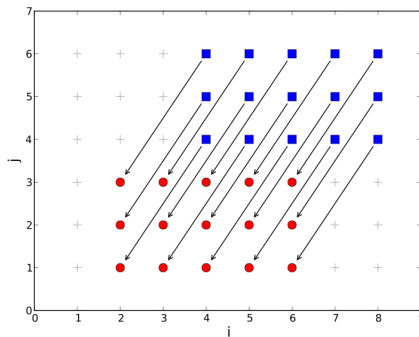


Figure 3: A two-dimensional integer map

Figure 3 illustrates the integer map $M = \{(i, j) \rightarrow (i-2, j-3)\}$ with the input values restricted to the elements contained in the blue rectangular of Figure 1. Each black arrow represents a relation between one input tuple and one output tuple. The input values (blue squares) shown are the very same values as illustrated in Figure 1. The output values (red circles) are the same values, but translated according to M .

The general form of an integer map is:

$$M = \{ \vec{i} \rightarrow \vec{o} \in \mathbb{Z}^d \times \mathbb{Z}^d \mid f(\vec{i}, \vec{o}, \vec{p}) \}$$

input tuples

output tuples

Presburger formula

Presburger formula that evaluates to true, if \vec{i} and \vec{o} are related in M for given parameters \vec{p} . Integer maps can represent arbitrary relations and can, contrary to what the use of the " \rightarrow " notation suggests, relate multiple output values to a single input value. The Presburger formulas that describe integer maps follow the rules presented for

Named unions sets / named union maps

✓ named integer sets are integer sets containing named tuples.
eg $\{S(i, j)\}$ an integer set with the name "S".

✓ named integer maps: integer maps between two named spaces.

✓ named union set, an integer set contains tuples from different spaces. eg $\{S_1(i, j); S_2(i)\}$

✓ named union maps: binary relations between integer sets.

2.2 MODEL AND TRANSFORM IMPERATIVE PROGRAMS

Polyhedra or, in our case, integer sets and maps can be used to model "sufficiently regular" compute programs with the goal to reason about and precisely control higher-level properties without distraction from imperative or lower-level constructs. To do so the individual statement instances in a program (i.e., each dynamic execution of a statement inside a loop nest), their execution order as well as the individual array elements accessed are modeled, analyzed and transformed, whereas control flow constructs, loop induction variables, loop bounds or array subscripts are hidden and only regenerated when converting a transformed loop nest back to imperative constructs.

side note: A set can be approximated by computing various hulls (convex, affine, simple, polyhedral.)

Polyhedral representation

what kind of higher-level property?

1. Goal: reason about and precisely control higher-level property without distraction from imperative or low-level construct.

2. What is modeled?

- ① individual statement instances
- ② their execution order
- ③ individual array element accessed

3. What is hidden?

- ① control-flow construct
- ② loop induction variables
- ③ loop bounds
- ④ array subscripts

Let's see the illustrative example

```
for (i = 1; i <= n; i+=1) → i-loop  
  for (j = 1; j <= i; j+=1) → j-loop  
S:   A[i][j] = A[i-1][j] + A[i][j-1];
```

↑
compute statement

To model this computation, 4 data structures are constructed

- ① iteration space I ← integer set, describes the set of statement
- ② schedule S ← assigns to each statement a integer map, possibly multi-dimensional time.
- ③ a relation of read-accesses A_{read} → defines memory locations
- ④ a relation of must-write-access A_{write} → defines execution order

```

for (i = 1; i <= n; i+=1)
  for (j = 1; j <= i; j+=1)
S:   A[i][j] = A[i-1][j] + A[i][j-1];

```

↓ we get the following model

statements

execution
order →

$I = \{S(i, j) \mid 1 \leq j \leq i \leq n\}$ integer set

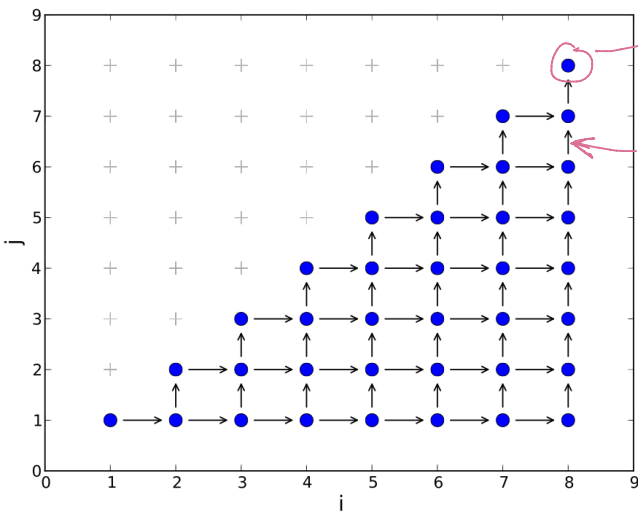
$S = \{S(i, j) \rightarrow (i, j)\}$ integer map assigned to statement instance.

$A_{read} = \{S(i, j) \rightarrow A(i-1, j); S(i, j) \rightarrow A(i, j-1)\}$ } integer map assigned to statement

$A_{write} = \{S(i, j) \rightarrow A(i, j)\}$

the memory locations they accesses

memory location ↓



statement
instance

data dependency

Figure 4: Iteration Space – Unmodified

The illustration of I in Figure 4 represents each statement instance with a single dot. The arrows between such statement instances illustrate data flow dependencies modeled by an integer map $D = \{S(i, j) \rightarrow S(i, j+1); S(i, j) \rightarrow S(i+1, j)\}$. Those dependencies relate statement instances with the statement instances they depend on. Computing precise data-flow dependencies [54, 104, 92] is one analysis that is significantly facilitated by the use of an integer set based representation.

* ~~compute precise data-flow dependences~~ is one analysis that is significantly facilitated by the use of an integer set based representation.

improve data-locality based on polyhedral representation.

↓
ensure that ~~statement instances that operate on the same data elements are executed in close time proximity.~~

2. each statement instance is always mapped to a point in time that is ~~later than~~ the execution time of all statement instances it depends on.

↓
~~within these constraints we are free to modify the schedule.~~

tiling is in this case both legal and effective. To implement loop tiling we define a new schedule

$$S' = \{S(i)(j) \rightarrow ([i/3])([j/3])(i)(j)\}$$

which defines an execution order where the statement instances are always executed in blocks of size 3×3 . The new execution order is illustrated in Figure 5 and is shown in two levels.

1. the blue blocks are executed in lexicographic order. At the lower level,
2. within the individual blue blocks, the statement instances are again executed in lexicographic order. As statement instances that are close by are placed in the same block, they are also executed close by in time.

one advantage of integer set representation

compute precise data-flow dependencies.

static control part / program

A SCoP is a program (region) that consists of a set of statements possibly enclosed by (not necessarily perfectly) nested loops and conditional branches. Within this region read-only scalar values are called parameters. The statements in the SCoP are side effect free, besides explicit reads and writes to multi-dimensional arrays or scalar values. Loops are regular loop bounds with a single lower and a single upper bound and increments by fixed, positive integers ($i+=10$). Both loop bounds and array accesses are (piecewise-quasi) affine expressions in terms of parameters and induction variables of outer loops.

imperative program that can be translated into polyhedral representation

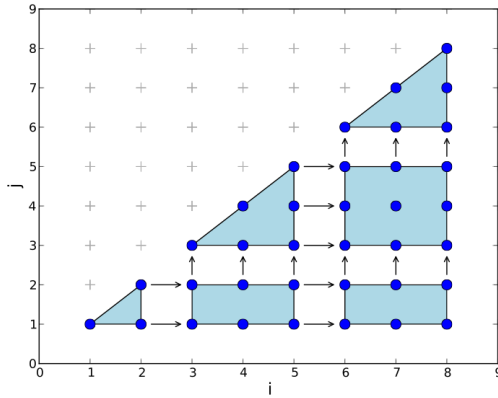
perfectly fit into RNN computation

I: iteration space is an integer set

S: schedule \rightarrow assign each statements a multi-dimensional time

How to understand locality? \rightarrow statement instances operating on the same data elements are executed in close time proximity.

Loop Tiling to increase locality



← scheduling leads to new execution order

Figure 5: Iteration Space - Tiled

tiling is in this case both legal and effective. To implement loop tiling we define a new schedule

$$S' = \{S(i)(j) \rightarrow ([i/3])([j/3])(i)(j)\}$$

materialize the transformations back to imperative codes

polyhedral AST generator

The polyhedral representation

The representation we use to model SCoPs consists of the following components:

1. **ITERATION SPACE/DOMAIN** The set of statements instances that are part of a SCoP. It is modeled as a **named union set**, where each named component of the union set describes a statement, with individual instances of a statement being described by the elements contained in the corresponding named set.
2. **ACCESS RELATIONS** A set of read, write and may-write access relations relate statement instances to the **data-locations** they access. These access relations are modeled as named union maps.
3. **DEPENDENCES** A relation between statement instances that defines **restrictions on the execution order**, due to **producer-consumer** relationships or **the shared usage of certain data locations**. Data dependences are modeled as named union maps.
4. **SCHEDULE** An **execution order** which assigns each statement instance a **multi-dimensional execution time**. One statement instance is executed before another statement instance, if its execution time is **lexicographically smaller**.

It is important to note that there is a **strong separation** between the statement instances themselves and the order in which they are executed. Program optimizations that **do not change the set of statements** that are executed, but **only change the order they are executed in**, consequently **only affect the schedule**.

Note for schedule:

1. program optimizations only change the order that statements are executed.
2. a schedule is only valid if it is of a form such that **all data-dependences go forward in time**.
the time at the source of the dependence must be **lexicographically strictly smaller** than the time at the target.
3. 1. given a set of dependence, we can compute a schedule.
2. given an ① iteration space ② access relations, ③ a schedule, the corresponding data dependences can be computed.

Transformations

We can group loop transformations according to what program properties are modified:

regularly
modeled
with integer
set

- The **order** in which computations are performed
- The **loop structure**, but not the order of computation
- The **data layout** and the **data locations accessed**
- The **computation** (algorithmic changes)

The first **three kinds of transformations** are **regularly modeled with integer sets**, whereas modeling algorithmic changes is more difficult, but possible in some cases [140]. Integer sets are also used for accelerator programming and vectorization, but such transformations are **mostly a combination of execution reordering and data layout transformations supplemented with the generation of specialized target instructions or library calls**. We will use such GPU code generation

Classical transformations that reorder computations are

1. fusion
2. fission
3. reversal
4. interchange
5. strip-mining
6. skewing

→ can be modeled by **schedule transformations**.

↓
combinations of such elementary transformations yield tiling or **unrolled and jam**.

1. Fusion

iteration space (named union set)

Fusion

original schedule $\mathcal{S} = \{S1(i) \rightarrow (0, i) \mid 1 \leq i < n; S2(i) \rightarrow (1, i) \mid 1 \leq i < m\}$

transformations $\mathcal{T} = \{(0, 0) \rightarrow (i, 0), (1, i) \rightarrow (i, 1)\}$

transformed schedule $\mathcal{S}_T = \{S1(i) \rightarrow (i, 0); S2(i) \rightarrow (i, 1)\}$

named union map

```
// Original loops
for (i=1; i<n; i+=1)
    S1(i);

for (i=1; i<m; i+=1)
    S2(i);

// Fused loops
for (i=1; i < min(n,m); i+=1) {
    S1(i);
    S2(i);
}

for (i=max(1,m); i<n; i+=1)
    S1(i);

for (i=max(1,n); i<m; i+=1)
    S2(i);
```

Notes for the above example.

1. There are two spaces.

2. Schedule assigns each statement instance a multi-dimensional execution time, here 2-d is used.

