✓ Integer maps

Integer maps are binary relations between integer set.

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

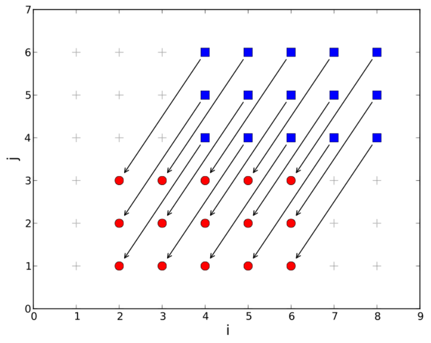

Figure 3: A two-dimensional integer map

Figure 3 illustrates the integer map $M = \{(i,j) \to (i-2, j-3)\}$ 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:

$$M = \{\underbrace{\vec{i}}_{\text{input tuples}} \to \underbrace{\vec{o}}_{\text{output tuples}} \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \underbrace{f}_{\text{Presburger formula}}(\vec{i}, \vec{o}, \vec{p})\}$$

Presburger formula that evaluates to true, *iff* $\vec{i}$ 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 "→" 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

It is possible to name integer sets and maps according to where they are used or what they represent. Named integer sets are integer sets which contain named tuples. An example is the set {S(i,j)} which is an integer set that lives in a two-dimensional space with the name "S". We define named integer maps as integer maps between two named spaces. The map $\{S(i,j) \to A(i)\}$ is an example of a named integer map. It is also possible to define integer sets that contain tuples from different spaces, e.g. $\{S1(i,j); S2(i)\}$. Such sets are called (named) union sets. Named union maps are defined accordingly. Operations that can be performed on sets and maps can often be performed on union sets and maps. In case we use such operations with non-obvious semantics, the actual semantics will be explained at the point of use. We may omit the prefix 'named' or 'union' in cases where it is either obvious or not relevant for the discussion.

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 1 execu~~tion of a statement~~ inside a loop nest), 2. their execution order as well as the 3. individual array elements access~~ed are modeled,~~ analyzed and trans~~formed, 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.

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 $\boxed{I}$ ← integer set, describes the set of statement

② schedule $\boxed{S}$ ← integer map, possibly multi-dimensional time. assigns to each statement a

③ a relation of read-accesses A_{read}

④ a relation of must-write-access A_{write}

→ defines memory locations

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];
```

$\downarrow$ we get the following model

statements

$$\boxed{I} = \{S(i, jj) \mid 1 \leqslant j \leqslant i \leqslant n\}$$

execution order $\rightarrow$

$$\boxed{S} = \{S(i,j) \rightarrow (i,j)\}$$

$$A_{\text{read}} = \{S(i,j) \rightarrow A(i-1,j); S(i,j) \rightarrow A(i,j-1)\}$$

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

$\uparrow$ memory location $\searrow$

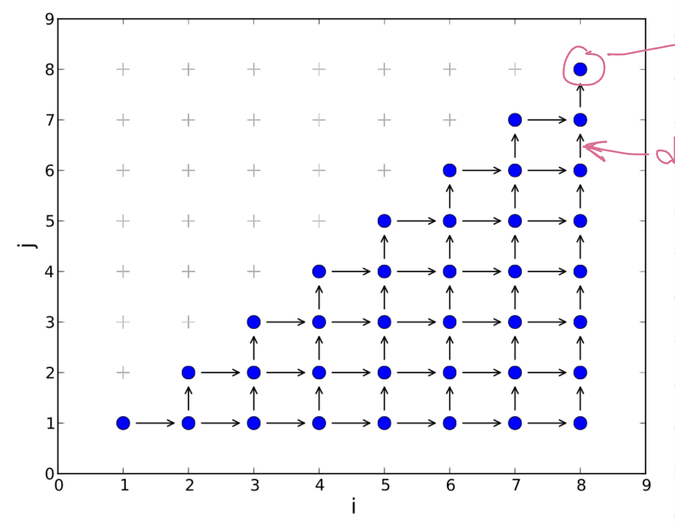

Figure 4: Iteration Space – Unmodified

The illustration of I in Figure 4 represents each statement instance with a single dot. The arrows between such statement instances illustrate data flow dependences modeled by an integer map $D = \{S(i,j) \rightarrow S(i,j+1); S(i,j) \rightarrow S(i+1,j)\}$. Those dependences relate statement instances with the statement instances they depend on. Computing precise data-flow dependences [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.

$\downarrow$

ensure that ✓1. 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.

$\downarrow$

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

$$S' = \{S(i)(j) \to (\lfloor i/3 \rfloor)(\lfloor j/3 \rfloor)(i)(j)\}$$

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. At the higher level,
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.

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.