Chapter 5

Polyhedral Model

5.1 Main Concepts

A polyhedral model is an abstraction of a piece of code that is used in various contexts and that therefore exists in various incarnations. There are some concepts that they all have in common, even though they may be called and represented differently.

Instance Set The instance set is the set of all "dynamic execution instances", i.e., the set of operations that are performed by the abstracted piece of code.

Dependence Relation The dependence relation is a binary relation between elements of the instance set where one of the instances depends on the other in some way. Several types of dependence relations can be considered and the exact nature of the dependence of one instance on the other depends on the type of the dependence relation. Typically, though, the dependence relation expresses that one instance needs to be executed before the other.

Schedule A schedule S defines a strict partial order $<_S$, i.e., an irreflexive and transitive relation, on the elements of the instance set that specifies the order in which they are or should be executed.

While some polyhedral compilation techniques only use a polyhedral model for *analysis* purposes, others also use it to *transform* the program fragment under consideration. These transformations are expressed through modifications of the schedule. The resulting schedules need to satisfy the following property.

Definition 5.1 (Valid Schedule)**.** *Let D be a dependence relation that expresses that the first instance needs to be executed before the second and let S be a schedule. The schedule S is said to be a* valid schedule *with respect to D, or, equivalently, to* respect *the dependences in D, if*

$$D \subseteq (<_S). \tag{5.1}$$

In order to accommodate dependences of instances on themselves, this condition may also be relaxed to

$$(D \setminus 1_{\mathrm{dom}\, D}) \subseteq (<_S). \tag{5.2}$$

Another commonly used abstraction is that of the *access relation*. This relation maps elements from the instance set to elements of some data set and expresses which data elements are or may be accessed by a given element of the instance set.

The `parse_file` operator of `iscc` can be used to extract parts of a polyhedral model from a C source file. In particular, this operator extracts a polyhedral model from the first suitable region in the source file. The operator takes a string containing the name of the source file as input and return a list containing the instance set (see Section 5.2 Instance Set), the must-write access relation, the may-write access relation, the may-read access relation (see Section 5.3 Access Relations), and a representation of the original schedule (see Section 5.6 Schedule).

The `pet_scop_extract_from_C_source` function of `pet` can be used to extract a polyhedral model from a specific function in a C source file. In particular, a polyhedral model in the form of a `pet_scop` is extracted from the first suitable region in that function. This function is exported by the `python` interface to `pet`. The function `pet_scop_get_schedule` can be used to extract the schedule from the `pet_scop`. The function `pet_scop_get_instance_set` can be used to extract the instance set from the `pet_scop`. The following functions can be used to extract access relations.

- `pet_scop_get_may_reads`,
- `pet_scop_get_may_writes`, and
- `pet_scop_get_must_writes`.

5.2 Instance Set

5.2.1 Definition and Representation

Definition 5.2 (Instance Set)**.** *The* instance set *is the set of all dynamic execution instances.*

The dynamic execution instances usually come in groups that correspond to pieces of code in the program that is being represented. The different instances in a group then correspond to the distinct executions of the corresponding piece of code at run-time. If the program is analyzed and/or transformed in source form, then these groups are typically the statements inside the analyzed code fragment, but a statement may also be decomposed into several groups or, conversely, a group may also contain several statements. If the program is analyzed in compiled form, then the groups typically correspond to the basic blocks in the internal representation of the compiler. In order to simplify

Note 5.1

```
S:      prod = 0;
        for (int i = 0; i < 100; ++i)
T:              prod += A[i] * B[i];
```

Listing 5.1: Inner product of two vectors of length 100

```
S:      prod = 0;
        for (int i = 0; i < n; ++i)
T:              prod += A[i] * B[i];
```

Listing 5.2: Inner product of two vectors of length n

the discussion, such a group, whether it represents a program statement, a basic block or something else entirely, will be called a *polyhedral statement*. Polyhedral statements are discussed in more detail in Section 5.8 Polyhedral Statements.

An instance set can be represented as a Presburger set by encoding the polyhedral statement in the name of each element and the dynamic instance of the polyhedral statement in its integer values. In particular, if the polyhedral statement is enclosed in n loops, then the dynamic instance is typically (but not necessarily) represented by n integer values, each representing the iteration count of one of the enclosing loops. It should be noted that these sequences of integers in the elements of the instance set only serve to *identify* the distinct dynamic instances and that they do not imply any particular order of execution. Also note that if the polyhedral model is only used to analyze a program, for example to determine properties of loops in the program, then the mapping between the statement instances and the loop iterations should either be implicit or it should be kept track of separately.

Example 5.3. *Consider the program fragment in Listing 5.1 for computing the inner product of two vectors* `A` *and* `B` *of length 100. There are two program statements in this fragment, one with label* `S` *and one with label* `T`*. Take these two program statements as the polyhedral statements. During the execution of this fragment, the statement with label* `S` *is executed once, while the statement with label* `T` *is executed 100 times. Each of these 100 executions can be represented by the value of the loop iterator* `i` *during the execution. That is, the instances of this program fragment can be represented by the instance set*

$$\{\, \mathtt{S}[]; \mathtt{T}[i] : 0 \leq i < 100 \,\}. \tag{5.3}$$

A program variable that is not modified inside the program fragment under analysis can be represented by a constant symbol since it has a fixed (but unknown) value. Such variables are also called *parameters*.

Example 5.4. *Consider the program fragment in Listing 5.2 for computing the inner product of two vectors* `A` *and* `B` *of length* `n`*. The only difference with*

the program fragment in Listing 5.1 on the preceding page is that the value 100 in the loop condition has been replaced by the variable `n`*. Since the value of* `n` *does not change during the execution of this program fragment, its instances can be represented by the instance set*

$$\{\, \mathtt{S}[]; \mathtt{T}[i] : 0 \leq i < n \,\}, \tag{5.4}$$

where n is a constant symbol.

Alternative 5.5 (Per-Statement Instance Set)**.** *Many approaches do not operate on a single instance set containing all instances of all polyhedral statements, but rather maintain separate instance sets per polyhedral statement.*

Alternative 5.6 (Instance Set Name)**.** *Various different names for (typically per-statement) instance sets are in common use, including iteration domain, index set and iteration space. The elements of these sets are often called iteration vectors.*

Note 5.2

Alternative 5.7 (Instance Set Representation)**.** *Many approaches use more restrictive representations for per-statement instance sets. In particular, they typically do not allow any integer divisions or quantifiers. Some do not allow any disjunctions (including negations of conjunctions) either. In this latter case, the instances of a statement are represented by the integer points in a polyhedron.*

Alternative 5.8 (Ordered Instance Set)**.** *Some approaches consider the elements of the instance set(s) to be ordered, typically lexicographically. Reordering transformations are then applied by modifying the elements of the instance set(s).*

5.2.2 Input Requirements and Approximations

In order to be able to represent the dynamic execution instances of a program fragment *exactly*, this fragment needs to satisfy certain conditions. Most importantly, the fragment needs to have *static control-flow*. That is, the control-flow needs to be known at compile time, possibly depending on the values of the constant symbols. This means that the control-flow should not depend on any input data in any other way and that moreover the compiler is able to figure out the control-flow. This typically means, for example, that the code cannot contain any `goto`s. The static control-flow requirement allows the compiler to determine at compile-time exactly which dynamic execution instances will

```
I:  sol = Initial_Solution(problem);
E1: error = Compute_Error(problem, sol);
    while (error >= threshold) {
U:          sol = Update_Solution(problem, sol);
E2:         error = Compute_Error(problem, sol);
    }
```

Listing 5.3: Pseudo code for incremental solver

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

Listing 5.4: Loop with non-constant increment

be executed at run-time. In order to be able to encode these instances in a Presburger formula, further restrictions need to be imposed. Typically, all conditions in the code are required to be (quasi-)affine expressions in the outer loop iterators and the parameters, the initial value of a loop iterator is required to be a (quasi-)affine expression in the outer loop iterators and the parameters, and the loop increment is required to be an integer constant.

Example 5.9. *Consider the pseudo code in Listing 5.3 for some incremental solver. Since the* `while`*-loop does not have an explicit loop iterator, it cannot be used to represent the instances of the two statements inside the loop. Moreover, it is impossible in general to describe the number of iterations of the loop (and hence the number of instances of the statements) as a quasi-affine expression. If the program is guaranteed to terminate, then is still possible to represent its instance set as*

$$\{\, \mathtt{I}[]; \mathtt{E1}[]; \mathtt{U}[i] : 0 \leq i < \mathrm{N}; \mathtt{E2}[i] : 0 \leq i < \mathrm{N} \,\}, \tag{5.5}$$

with N *a constant symbol that represent the unknown number of iteration of the loop. However, such an encoding is only possible for an outer* `while`*-loop since the number of iterations of a* `while`*-loop that is embedded in another loop will typically depend on the iteration of that outer loop and can therefore not be represented by a single constant symbol.*

Example 5.10. *Consider the code fragment in Listing 5.4. If the value of* `n` *is unknown, then it is not possible to describe the instances of statement* `S` *using a Presburger formula when using the value of the loop iterator* `i` *to identify each loop instance. (If the value of* `n` *is known, then those values can simply be enumerated individually as a Presburger formula.) In this example, it is still possible to describe the instance set as*

$$\{\, \mathtt{S}[j] : 0 \leq j \leq \mathrm{N} \wedge \mathrm{n} \geq 1 \,\}, \tag{5.6}$$

where N *and* n *are constant symbols that satisfy* $\mathrm{N} = \lfloor \log_2 \mathrm{n} \rfloor$ *if* $\mathrm{n} \geq 1$. *Since* `n` *is not changed during the execution of this program fragment, both* n *and* N *can indeed be used as constant symbols. However, it is more difficult for a compiler to extract such an instance set. Moreover, the relationship between* n *and* N *cannot be expressed in the instance set. This means that during further computations, the compiler may end up considering combinations of* n *and* N *that cannot occur in practice. Finally, the value of* `i` *needs to be replaced by* 2^j *in other parts of the model, which also cannot be represented in a Presburger formula.*

It should be noted though that the instance set may also be an *overapproximation* of the instances that actually get executed at run-time. Most analysis techniques are safe with respect to overapproximations such that no further adjustments are required if the polyhedral model is only used for analysis purposes. If the polyhedral model is also used to transform the input program, then additional measures need to be taken to ensure that the set of instances that are actually executed at run-time is the same for both input and output program. This usually requires keeping track of extra information in the polyhedral statements.

5.2.3 Representation in `pet`

In `pet`, a dynamic execution instance is represented as a named integer tuple where the name identifies the statement and the integer values correspond to the values of the outer loop iterators, from outermost to innermost. If the statement has a label, then that label is used as the name of the polyhedral statement. Otherwise, the polyhedral statement is assigned a generated name. In some cases, a "*virtual iterator*" is used to identify the iterations of a particular loop. This happens in particular if the loop does not have an explicit iterator or if the value of this iterator cannot be used to uniquely identify the iteration, but it can also happen in cases where using the actual iterator would lead to complicated expressions in the rest of the model.

Example 5.11. *Consider first the program in Listing 5.5 on the next page, which is a completed version of the program fragment in Listing 5.2 on page 89. The actual loop iterator* `i` *can be used to index the* `T`*-elements of the instance set without any complication. The instance set extracted by* `pet` *is therefore equal to the set in* (5.4)*, as shown below.*
`iscc` *input (*`inner.iscc`*) with source in Listing 5.5 on the next page:*

```
P := parse_file "demo/inner.c";
print P[0];
```

`iscc` *invocation:*

```
iscc < inner.iscc
```

`iscc` *output:*

```
float inner(int n, float A[const restrict static n],
        float B[const restrict static n])
{
        float prod;

S:      prod = 0;
L:      for (int i = 0; i < n; ++i)
T:              prod += A[i] * B[i];

        return prod;
}
```

Listing 5.5: Input file *inner.c*

```
int g();
void h(int);

void f()
{
        int a;

        while (1) {
A:              a = g();
B:              h(a);
        }
}
```

Listing 5.6: Input file *infinite.c*

```
[n] -> { T[i] : 0 <= i < n; S[] }
```

Example 5.12. *Consider now the program in Listing 5.6. The loop does not have a loop iterator, so* `pet` *introduces one to index the elements in the instance set, as shown below.*
`iscc` *input (*`infinite.iscc`*) with source in Listing 5.6:*

```
P := parse_file "demo/infinite.c";
print P[0];
```

`iscc` *invocation:*

```
iscc < infinite.iscc
```

`iscc` *output:*

```
{ B[t] : t >= 0; A[t] : t >= 0 }
```

```
int f()
{
        unsigned char k;
        int a;

Init:   a = 0;
        for (k = 252; (k % 9) <= 5; ++k)
Inc:            a = a + 1;
        return a;
}
```

Listing 5.7: Input file *unsigned.c*

Example 5.13. *Consider the program in Listing 5.7. The* `Inc`*-statement is executed for the following value of the loop iterator* `k`*, 252, 253, 254, 255, 0, 1, 2, 3, 4, 5, in that order. While it is perfectly possible to construct an instance set with these values for the elements, it is more convenient to use consecutive values to represent the statement instances and this is what* `pet` *does below.*
`iscc` *input (*`unsigned.iscc`*) with source in Listing 5.7:*

```
P := parse_file "demo/unsigned.c";
print P[0];
```

`iscc` *invocation:*

```
iscc < unsigned.iscc
```

`iscc` *output:*

```
{ Inc[k] : 252 <= k <= 261; Init[] }
```

If `pet` comes across any dynamic control in the analyzed program fragment, then it will either keep track of extra information on the conditions under which a given statement is executed in the corresponding polyhedral statement or it will consider the dynamic control to be embedded in the enclosing statement. The choice is controlled by the `--encapsulate-dynamic-control` command line option, which is enabled by default by `iscc`, or by calling the function `pet_options_set_encapsulate_dynamic_control`.

Example 5.14. *Consider the program in Listing 5.8 on the facing page. Since the* `if`*-condition is non-static,* `pet` *does not create a separate polyhedral statement for the* `Update` *statement, but instead includes the* `if`*-statement in the polyhedral statement. This is reflected in the label that is used to identify the statements in the output below.*
`iscc` *input (*`max.iscc`*) with source in Listing 5.8 on the next page:*

```
P := parse_file "demo/max.c";
print P[0];
```

```
float max(unsigned n,
        float A[const restrict static 1 + n])
{
        float M;

Init:   M = A[0];
        for (unsigned i = 0; i < n; ++i)
If:             if (A[1 + i] > M)
Update:                 M = A[1 + i];

        return M;
}
```

Listing 5.8: Input file *max.c*

```
int g(int);
int t(int);
void f(int n, int A[const restrict static n])
{
        int done;

Init:   done = 0;
While:  while (!done) {
Reset:          done = 1;
                for (int i = 0; i < n; ++i) {
Update:                 A[i] = g(A[i]);
Test:                   if (t(A[i]))
Reinit:                         done = 0;
                }
        }
}
```

Listing 5.9: Input file *while.c*

iscc invocation:

```
iscc < max.iscc
```

iscc output:

```
[n] -> { If[i] : 0 <= i < n; Init[] }
```

Example 5.15. *Consider the program in Listing 5.9. Since* `pet` *is unable to determine the number of iteration of the* `while`*-loop at compile-time, it considers the entire loop as an indivisible polyhedral statement. Since the loop as*

```
int g(int);
int t(int);
void f(int n, int A[const restrict static n])
{
        int done;

Init:   done = 0;
While:  while (!done) {
#pragma scop
Reset:          done = 1;
                for (int i = 0; i < n; ++i) {
Update:                 A[i] = g(A[i]);
Test:                   if (t(A[i]))
Reinit:                         done = 0;
                }
#pragma endscop
        }
}
```

Listing 5.10: Input file *while2.c*

a whole is only executed once, there is only one dynamic execution instance of this polyhedral statement, as shown below.
iscc input (while.iscc) with source in Listing 5.9 on the previous page:

```
P := parse_file "demo/while.c";
print P[0];
```

iscc invocation:

```
iscc < while.iscc
```

iscc output:

```
[n] -> { While[]; Init[] }
```

It is possible to tell `pet` *to extract a polyhedral model from the* body *of the loop by marking the body with pragmas and turning off the autodetect option as shown below. Note that in the* `pet` *library, the autodetect option is turned off by default. The state of this option may be changed by calling the* `pet_options_set_autodetect` *function.*
iscc input (while2.iscc) with source in Listing 5.10:

```
P := parse_file "demo/while2.c";
print P[0];
```

iscc invocation:

```
iscc --no-pet-autodetect < while2.iscc
```

iscc output:

```
[n] -> { Test[i] : 0 <= i < n; Reset[]; Update[i] : 0 <= i <
    ↪  n }
```

5.3 Access Relations

5.3.1 Definition and Representation

An access relation maps elements of the instance set to the data elements that are accessed by that statement. It is usually important to make a distinction between read and write accesses.

Definition 5.16 (Read Access Relation)**.** *The* read access relation *maps each dynamic execution instance to the set of data elements read by the dynamic execution instance.*

Definition 5.17 (Write Access Relation)**.** *The* write access relation *maps each dynamic execution instance to the set of data elements written by the dynamic execution instance.*

In some cases, it may be impossible or impractical to determine the exact set of accessed data elements. Furthermore, even if it is possible to determine the exact access relations, it may be impossible to represent them as a Presburger relation. The access relations may therefore need to be approximated. In case of a read, it is sufficient to determine an overapproximation of the accessed data elements. The case of a write needs a bit more consideration, however. Some uses of the write access relation, e.g., for computing the total set of elements that may be accessed by a program fragment, also allow for an overapproximation. Some other uses of the write access relation do not allow for overapproximations, but require an underapproximation instead. This leads to the following three types of access relations. Note 5.3

Definition 5.18 (May-Read Access Relation)**.** *A* may-read access relation *is a binary relation that contains the read access relation as a subset.*

Definition 5.19 (May-Write Access Relation)**.** *A* may-write access relation *is a binary relation that contains the write access relation as a subset.*

Definition 5.20 (Must-Write Access Relation)**.** *A* must-write access relation *is a binary relation that is a subset of the write access relation.*

Note that these definitions do not specify the exact contents of the relations, but only that they contain at least some pairs of elements in case of the may-read and may-write relation or that they contain at most some pairs of elements in case of the must-write relation. This flexibility is useful in cases where it is not clear at compile-time exactly which elements will be accessed

by a given dynamic execution instance, or if this information cannot be represented exactly. In those cases where this information is available and can be represented exactly, the access relations can be restricted/extended to include exactly those data elements that are accessed. The may-write access relation is then equal to the must-write access relation. In general, the must-write access relation in a subrelation of the may-write access relation.

Exploiting the fact that the three access relation do not need to be exact, they can all be represented as Presburger relations. The domain elements in these relations are elements of the instance set and therefore have the same representation. The range elements, i.e., the accessed data elements, are represented in a similar way. Like the elements of the instance set, these data elements come in groups (typically arrays) and each element is identified by the name of the group (array) and a sequence of integers that is unique with the group (the index of the array element). Note that since scalars cannot be indexed, the representation of a scalar consists of only a name and the corresponding sequence of integers is empty. That is, a scalar is treated as a zero-dimensional array.

Note 5.4

Example 5.21. *Consider once more the program in Listing 5.5. The access relations are shown in the transcript below. Note that the may-write access relation is equal to the must-write access relation because the accesses can be completely determined at compile-time and can be described using a Presburger formula. Note also that the access to* `prod` *in statement* `T` *updates* `prod` *and is therefore considered both a read and a write.*

`iscc` *input (*`inner_access.iscc`*) with source in Listing 5.5 on page 93:*

```
P := parse_file "demo/inner.c";
print "Must-write:";
print P[1];
print "May-write:";
print P[2];
print "May-read:";
print P[3];
```

`iscc` *invocation:*

```
iscc < inner_access.iscc
```

`iscc` *output:*

```
"Must-write:"
[n] -> { S[] -> prod[]; T[i] -> prod[] : 0 <= i < n }
"May-write:"
[n] -> { S[] -> prod[]; T[i] -> prod[] : 0 <= i < n }
"May-read:"
[n] -> { T[i] -> B[i] : 0 <= i < n; T[i] -> A[i] : 0 <= i <
    ↪ n; T[i] -> prod[] : 0 <= i < n }
```

```
void set_diagonal(int n,
        float A[const restrict static n][n], float v)
{
        for (int i = 0; i < n; ++i)
                A[i][i] = v;
}

void f(int n, float A[const restrict static n][n])
{
#pragma scop
S:      set_diagonal(n, A, 0.f);
        for (int i = 0; i < n; ++i)
                for (int j = i + 1; j < n; ++j)
T:                      A[i][j] += A[i][j - 1] + 1;
#pragma endscop
}
```

Listing 5.11: Input file *diagonal.c*

Note that an access relation need not be a function, either because several elements are effectively accessed directly or indirectly by the same instance of a polyhedral statement, or because it is not clear which element is being accessed such that several elements *may* be accessed. In the worst case, the entire array may be accessed, where the set of array elements is derived from the declaration of the array. If this array is a function argument of a C function, then it is important to also specify the size of the array in the outer dimension by placing the `static` keyword next to the otherwise dummy size expression. Note 5.5

Example 5.22. *The analyzed program fragment in Listing 5.11 contains two statements that access multiple elements of the same array from the same instance. In statement `S` the accesses are performed indirectly through a call to the `set_diagonal` function, while statement `T` simply contains two read accesses (one a pure read and one an update) to the same array. The complete access relations are shown in the transcript below.*
`iscc` input (`diagonal.iscc`) with source in Listing 5.11:

```
P := parse_file "demo/diagonal.c";
print "Must-write:";
print P[1];
print "May-write:";
print P[2];
print "May-read:";
print P[3];
```

`iscc` invocation:

```
iscc --no-pet-autodetect < diagonal.iscc
```

```
void f(int n, int n2, float A[const restrict static n2])
{
        for (int i = 0; i < n; ++i)
                A[i * i] = i;
}
```

Listing 5.12: Input file *square.c*

iscc output:

```
"Must-write:"
[n] -> { T[i, j] -> A[i, j] : 0 <= i < n and j > i and 0 <=
    ↪ j < n; S[] -> A[o0, o0] : 0 <= o0 < n }
"May-write:"
[n] -> { T[i, j] -> A[i, j] : 0 <= i < n and j > i and 0 <=
    ↪ j < n; S[] -> A[o0, o0] : 0 <= o0 < n }
"May-read:"
[n] -> { T[i, j] -> A[i, j] : 0 <= i < n and j > i and 0 <=
    ↪ j < n; T[i, j] -> A[i, -1 + j] : 0 <= i < n and j > i
    ↪  and 0 < j < n }
```

Example 5.23. *Listing 5.12 shows an example of a program where the index expression cannot be represented using an affine expression. The must-write access relation is therefore left empty, while the may-write access relation is defined to access the entire array, where the constraints are derived from the size of the array. The access relations derived by* `pet` *are shown below.*
`iscc` input (`square.iscc`) with source in Listing 5.12:

```
P := parse_file "demo/square.c";
print "Must-write:";
print P[1];
print "May-write:";
print P[2];
print "May-read:";
print P[3];
```

`iscc` invocation:

```
iscc < square.iscc
```

`iscc` output:

```
"Must-write:"
[n2, n] -> {  }
"May-write:"
[n2, n] -> { S_0[i] -> A[o0] : 0 <= i < n and 0 <= o0 < n2 }
"May-read:"
[n2, n] -> {  }
```

Alternative 5.24 (Exact Access Relations)**.** *Many approaches do not consider a separate may-write and must-write access relation, but simply a write-access relation. These approaches then also need to impose restrictions on the kinds of accesses that may be performed by the input program.*

Alternative 5.25 (Per-Reference Access Relations)**.** *Many approaches do not consider global access relations that describe accesses in the entire code fragment, but rather maintain separate access relations for each array reference in each polyhedral statement.*

Alternative 5.26 (Access Functions)**.** *Some approaches do not allow a reference inside a polyhedral statement instance to access more than one data element and use functions to represent the accesses rather than more general relations. These access functions are naturally defined per reference and are typically also exact. Such access functions bear some resemblance to the index expressions of Section 5.8 Polyhedral Statements, but they are not quite the same.*

5.3.2 Aliasing

In order to be able to construct the may-read access relation and may-write access relation correctly, the compiler needs to be aware of any aliasing that may be occurring in the program. For example, if a statement writes to `A` and `A` may be aliased to `B`, then the may-write access relation needs to include accesses to `B` from that statement. In the worst case, every polyhedral statement instance that writes anything may have to be considered to write to every element of every array, in which case the polyhedral model will not be very useful. It is therefore best to avoid aliasing as much as possible, which is why, in particular, most approaches to polyhedral compilation do not allow any pointer manipulations.

There are essentially three approaches to avoiding aliasing.

- Ignore aliasing

  Several tools, including `pet`, simply assume that there is no aliasing between different arrays.

- Require absence of aliasing

  One variant of this approach is to extract polyhedral models from a source language that does not permit aliasing. However, in a source language such as C, aliasing does need to be taken into account. Locally declared arrays cannot alias with each other, but arrays passed to a function are actually *pointers* to the starts of the arrays (at least in C) and it is

Note 5.6

therefore possible for such arrays to alias. The `restrict` keyword can be used on those pointers to indicate that they do not in fact alias. If an array of arrays is passed to a function, then the elements of the outer array are also pointers and they should also be required to be annotated with `restrict` to indicate that there is no aliasing among the rows of the array. However, in general it is much preferred to pass a multi-dimensional array instead. In such a multi-dimensional array, the rows are stored successively in memory and are therefore guaranteed not to alias.

- Check aliasing at run-time

  In this approach, arrays are assumed not to alias for the purpose of analyzing and transforming the code, but the groups of arrays that may potentially alias are collected as well. Extra code is then inserted into the transformed program that checks whether there is any aliasing inside those groups at run-time. If so, the original code is executed. If not, the transformed code is executed.

5.3.3 Structures

Accesses to plain structures, i.e., structures that do not contain any pointers, do not in principle require any special treatment. It is only a matter of finding the right representation for such accesses. Pointers could be allowed, but in order to avoid aliasing, they would have to be annotated with `restrict` just like function arguments or the elements of nested arrays. Recursive data structure are more challenging, however, since it is not clear how to represent them in a polyhedral framework.

Note 5.7

In `pet`, accesses to structure fields are encoded using wrapped relations. In particular, the range of the access relation is a wrapped relation with the domain identifying the structure and the range identifying the field inside the structure. The identifier of the wrapped relation is composed of the name of the outer array or scalar and the name of the field.

Example 5.27. *Consider the program in Listing 5.13 on the facing page. It contains a write access to the* `b` *field of elements of the* `c` *array, which are structures of type* `struct s`*. The transcript below prints the corresponding access relation.*
`iscc` *input (*`struct.iscc`*) with source in Listing 5.13 on the next page:*

```
P := parse_file "demo/struct.c";
print P[1];
```

`iscc` *invocation:*

```
iscc < struct.iscc
```

`iscc` *output:*

```
{ S[i] -> c_b[c[i, i] -> b[9 - i]] : 0 <= i <= 9 }
```

```
struct s {
        int a;
        int b[10];
};

void f(struct s c[static 10][10])
{
        for (int i = 0; i < 10; ++i)
S:              c[i][i].b[9 - i] = 0;
}
```

Listing 5.13: Input file *struct.c*

```
struct S {
        struct {
                int a[10];
        } f[10];
};

void foo()
{
        struct S s;

#pragma scop
        for (int i = 0; i < 10; ++i)
                for (int j = 0; j < 10; ++j)
                        s.f[i].a[j] = i * j;
#pragma endscop
}
```

Listing 5.14: Input file *struct2.c*

If the accessed field is itself a structure or an array of structures, then accesses to fields in those structure are represented as structure field accesses in a structure that is itself a structure field access. That is, the domain of the wrapped relation is itself a wrapped relation representing the access to the inner structure. This process continues recursively for any further nesting of field accessing. Note 5.8

Example 5.28. *Consider the program in Listing 5.14. It contains a write access to the* `a` *field of elements of an* `f` *array, which is itself a field of a variable of type* `struct s`*. The transcript below prints the corresponding access relation.* `iscc` *input (*`struct2.iscc`*) with source in Listing 5.14:*

```
P := parse_file "demo/struct2.c";
```

```
struct c {
        float re;
        float im;
};

void f(struct c A[const static 10])
{
S:      A[0] = A[2];
T:      A[1].re = A[0].im;
}
```

Listing 5.15: Input file *struct3.c*

```
print P[1];
```

iscc invocation:

```
iscc < struct2.iscc
```

iscc output:

```
{ S_2[i, j] -> s_f_a[s_f[s[] -> f[i]] -> a[j]] : 0 <= i <= 9
    ↪ and 0 <= j <= 9 }
```

Note that an access to an entire structure means that all fields of the structure are accessed. This is then also how it is represented in `pet`. This handling of structure accesses is similar to how accesses to entire arrays or rows of an array are handled when they are passed to a function.

Example 5.29. *Consider the program in Listing 5.15. Statement* `S` *copies an entire structure from one element of* `A` *to another element of* `A`*. This means that both fields of the structure are read and written. The corresponding access relations are shown in the transcript below.*
iscc input (struct3.iscc) with source in Listing 5.15:

```
P := parse_file "demo/struct3.c";
print "Must-write:";
print P[1];
print "May-write:";
print P[2];
print "May-read:";
print P[3];
```

iscc invocation:

```
iscc < struct3.iscc
```

iscc output:

```
"Must-write:"
{ S[] -> A_re[A[0] -> re[]]; T[] -> A_re[A[1] -> re[]]; S[]
    ↪ -> A_im[A[0] -> im[]] }
"May-write:"
{ S[] -> A_re[A[0] -> re[]]; T[] -> A_re[A[1] -> re[]]; S[]
    ↪ -> A_im[A[0] -> im[]] }
"May-read:"
{ T[] -> A_im[A[0] -> im[]]; S[] -> A_re[A[2] -> re[]]; S[]
    ↪ -> A_im[A[2] -> im[]] }
```

5.3.4 Tagged Access Relations

The standard access relations map statement instances to data elements accessed by that statement instance. However, a given statement may reference the same data structure several times and in some cases it is important to make a distinction between the individual references. For example, when `PPCG` is determining which data to copy to/from a device, it checks which of the write references produce data that is only used inside a given kernel. This requires the reference to be identifiable from the dependence relations, which in turn requires them to be encoded in the access relations.

In `pet`, unique identifiers are generated for each reference in the program fragment. These identifiers are then used to "tag" the statement instance performing the access in what are called *tagged access relations*. In particular, the domain of such a tagged access relation is a wrapped relation with as domain the statement instance and as range the reference identifier. The tags can be removed from such a tagged access relation by computing the domain product domain factor. The tagged access relations can be extracted from a `pet_scop` using the following functions.

- `pet_scop_get_tagged_may_reads`,
- `pet_scop_get_tagged_may_writes`, and
- `pet_scop_get_tagged_must_writes`.

Example 5.30. *`python` input (`tagged.py`) with source in Listing 5.11 on page 99:*

```
import isl
import pet

scop = pet.scop.extract_from_C_source("demo/diagonal.c",
                                      "f");
may_read = scop.get_may_reads()
tagged_may_read = scop.get_tagged_may_reads()
print may_read
print tagged_may_read
factor = tagged_may_read.domain_factor_domain()
```

```
print may_read.is_equal(factor)
```

python invocation:

```
python < tagged.py
```

python output:

```
[n] -> { T[i, j] -> A[i, j] : 0 <= i < n and j > i and 0 <=
    ↪ j < n; T[i, j] -> A[i, -1 + j] : 0 <= i < n and j > i
    ↪  and 0 < j < n }
[n] -> { [T[i, j] -> __pet_ref_2[]] -> A[i, j] : 0 <= i < n
    ↪ and j > i and 0 <= j < n; [T[i, j] -> __pet_ref_3[]]
    ↪ -> A[i, -1 + j] : 0 <= i < n and j > i and 0 < j < n
    ↪ }
True
```

5.4 Dependence Relations

This section only describes the general concept of dependence relations. The computation of dependence relations is described in Chapter 6 Dependence Analysis

In general, a dependence is a pair of statement instances that expresses that the second statement instance should be executed after the first instance. A dependence relation is a collection of dependences. The cause of a dependence is usually that the two statement instances involved access the same memory element.

Different types of dependences can be distinguished depending on the types of the two accesses involved.

Definition 5.31 (Read-after-Write Dependence Relation)**.** *The* read-after-write dependence relation *maps a statement instance* $\boldsymbol{i}$ *to a statement instance* $\boldsymbol{j}$ *if* $\boldsymbol{j}$ *is executed after* $\boldsymbol{i}$ *and if it reads from a data element that is written by* $\boldsymbol{i}$*.*

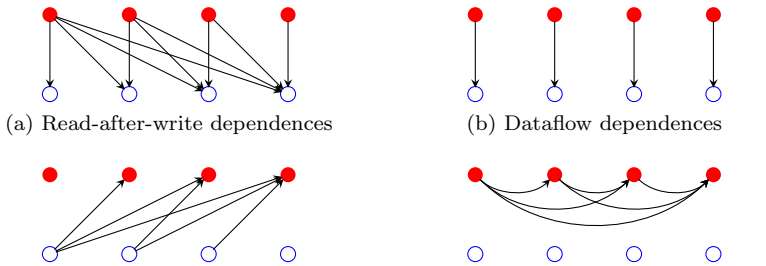
Definition 5.32 (Write-after-Read Dependence Relation)**.** *The* write-after-read dependence relation *maps a statement instance* $\boldsymbol{i}$ *to a statement instance* $\boldsymbol{j}$ *if* $\boldsymbol{j}$ *is executed after* $\boldsymbol{i}$ *and if it writes to a data element that is read by* $\boldsymbol{i}$*.*

Definition 5.33 (Write-after-Write Dependence Relation)**.** *The* write-after-write dependence relation *maps a statement instance* $\boldsymbol{i}$ *to a statement instance* $\boldsymbol{j}$ *if* $\boldsymbol{j}$ *is executed after* $\boldsymbol{i}$ *and if it writes to a data element that is written by* $\boldsymbol{i}$*.*

As in the case of access relations, it may not be possible to compute these relations exactly or to represent them exactly as Presburger relations. However, since they are only meant to express that the second statement instance should be executed after the first, it is safe to consider overapproximations. Any *read-after-write dependence relation* mentioned in the rest of this manual will then

```
for (int i = 0; i < n; ++i) {
S:      t = f1(A[i]);
T:      B[i] = f2(t);
}
```

Listing 5.16: Code with false dependences



(a) Read-after-write dependences (b) Dataflow dependences

(c) Write-after-read dependences (d) Write-after-write dependences

Figure 5.17: Dependences of the code in Listing 5.16. Instances of statement `S` are represented as ● , instances of statement `T` are represented as ○ .

actually be a may-read-after-write dependence relation, and similarly for any *write-after-read dependence relation* or *write-after-write dependence relation*.

The elements of the read-after-write dependence relation are called the *read-after-write dependences*. They are needed because the read access may read a value that was written by the write access. The elements of the write-after-read dependence relation are called the *write-after-read dependences* or the *anti-dependences*. They are needed because the write access may overwrite a value that was read by the read access. The elements of the write-after-write dependence relation are called the *write-after-write dependences* or the *output dependences*. They are needed because the second write access may overwrite a value that was written by the first write access. Enforcing the first write to be executed before the second is important to ensure that the final value written to a data element in the original program is not overwritten by a value that was written to it before in the original program.

Note that a pair of read accesses does *not* give rise to a dependence because the two reads do not influence each other. It can however still be useful to consider pairs of statement instances that read the same memory element for optimization purposes. In analogy with the actual dependences, such pairs of statement instances are sometimes called *read-after-read dependences* or *input dependences*. Note that in contrast to the case of the actual dependences, for input dependences the order of the two statement instances is of no importance.

Example 5.34. *Consider the code in Listing 5.16. Both statements access the* `t` *scalar, with* `S` *writing to the scalar and* `T` *reading from the scalar. The depen-*

dence relations are then as follows. The read-after-write dependence relation:

$$\{\, \mathtt{S}[i] \to \mathtt{T}[i'] : i' \geq i \,\}. \tag{5.7}$$

This relation is shown in Figure 5.17a. The write-after-read dependence relation:

$$\{\, \mathtt{T}[i] \to \mathtt{S}[i'] : i' > i \,\}. \tag{5.8}$$

This relation is shown in Figure 5.17c. The write-after-write dependence relation:

$$\{\, \mathtt{S}[i] \to \mathtt{S}[i'] : i' > i \,\}. \tag{5.9}$$

This relation is shown in Figure 5.17d. The computation of these dependence relation is illustrated in Example 6.1 on page 134.

The main purpose of the dependences described so far is to make sure that values are written to memory before they are read and that they are not overwritten in between. In some cases, there may be so many of these dependences that the execution order of the statement instances can hardly be changed or even not at all. For example, if a temporary scalar variable is used to store different data, then the dependences described above will serialize the statement instances accessing that scalar variable. By storing different data in different memory locations, some of these dependences are no longer required and more freedom is created for changing the execution order of the statement instances. In particular, (some) anti-dependences and output dependences can be removed and it is for this reason that they are also collectively known as the *false dependences*.

In order to be able to map different data to different memory location, it is important to determine where a new value is written and how long it needs to be stored. This information is captured by the *dataflow dependences*. In particular, there is a dataflow dependence between any write to a memory location and any later read from the same memory location that still finds the value that was written by the write access. That is, the memory location was not overwritten in between.

Definition 5.35 (Dataflow Dependence Relation)**.** *The* dataflow dependence relation *is a subset of the (exact) read-after-write dependence relation containing those pairs of statement instances for which there is no intermediate write to the same data element accessed by both statement instances.*

The dataflow dependences are also known as *value-based dependences* because the value is preserved along the dependence. In contrast, the previously described dependences are also known as *memory-based dependences* because they merely access the same memory location.

Note 5.9

As usual, it may not be possible to determine or represent the dataflow dependence relation exactly and, as in the case of write accesses, it is important to make a distinction between *potential* dataflow and *definite* dataflow. This leads to the following two types of dataflow dependence relations.

Definition 5.36 (May-Dataflow Dependence Relation)**.** *A* may-dataflow dependence relation *is a binary relation that contains the dataflow dependence relation as a subset.*

Definition 5.37 (Must-Dataflow Dependence Relation)**.** *A* must-dataflow dependence relation *is a binary relation that is a subset of the dataflow dependence relation.*

Both also come in a "tagged" form where each statement instance is accompanied by a reference identifier, as in the case of the tagged access relations of Section 5.3.4 Tagged Access Relations. These are called the *tagged may-dataflow dependence relation* and the *tagged must-dataflow dependence relation*. The must-dataflow dependence relation is a subset of the may-dataflow dependence relation. If the dataflow analysis can be performed exactly, then the two are equal to each other. The may-dataflow dependence relation is itself a subrelation of the (may-)read-after-write dependence relation. The untagged must-dataflow dependence relation is only useful if each statement contains at most one write access.

Example 5.38. *Consider once more the code in Listing 5.16 on page 107. Each value written to the scalar* `t` *by an instance of statement* `S` *is overwritten by the next instance of the same statement. This means that the value is only read by the (single) intermediate instance of statement* `T`*. That is, the dataflow dependence relation is*

$$\{\, \mathtt{S}[i] \to \mathtt{T}[i] \,\}. \tag{5.10}$$

Note that this relation is a strict subrelation of the read-after-write dependence relation of (5.7)*. It is shown in Figure 5.17b.*

Alternative 5.39 (Per-Statement-Pair Dependence Relation)**.** *Many approaches do not operate on a single dependence relation containing pairs of instances of different polyhedral statements, but instead keep track of separate dependence relations for each pair of statements. In some cases, the dependence relations are further broken up, possibly along the disjuncts of a representation in disjunctive normal form.*

Alternative 5.40 (Dependence Polyhedron)**.** *Some approaches represent dependence relations as polyhedra, where the input and output tuples are simply concatenated. This representation is called the dependence polyhedron. Since a single polyhedron cannot represent a disjunction, this implies a decomposition along disjuncts of a representation in disjunctive normal form. Furthermore, plain polyhedra do not support existentially quantified variables, meaning that in general, dataflow dependence relations cannot be represented very accurately.* Note 5.10

```
for (int i = 0; i < n; ++i) {
S:      t[i] = f1(A[i]);
T:      B[i] = f2(t[i]);
}
```

Listing 5.18: Code without false dependences

Alternative 5.41 (Depends-on Relation)**.** *Some authors prefer to consider dependences that go from a statement instance to the statement instances on which it depends. That is, the position of the two statement instances is reversed compared to the dependence relations defined above. In the case of exact dataflow dependences, this means that the dependence relation can be represented as a function since for each read operation there is at most one write operation that writes the value that is read by the read operation.*

5.5 Data-Layout Transformation

A *data-layout transformation* changes the way data is stored in memory. This may just be a reordering of the data elements, but it may also map several data elements in the original program to a single data element, or, conversely, map a single data element in the original program to multiple data elements. Data-layout transformations that map several elements to a single element are called *contractions*. Data-layout transformations that map a single element to multiple elements are called *expansions*. Note that a data-layout transformation typically also requires modifications to the variable declarations.

Note 5.11

Note 5.12

In `isl`, a data-layout transformation can be represented as a multi-space piecewise tuple of quasi-affine expressions. The range of this function corresponds to the new data elements. In simple cases, where the data is uniformly reordered, the domain of the function can correspond directly to the original data elements. If the transformation depends on the statement instance, then the domain should be a (wrapped) access relation. In particular, expansions depend on the statement instance. The transformation may also be reference specific, in which case the domain should be a (wrapped) tagged access relation.

Example 5.42. *Consider once more the code in Listing 5.16 on page 107. As explained in Example 5.34 on page 107, the code exhibits false dependences. They can be removed by expanding the* `t` *scalar to an array of a size that is equal to the number of iterations in the loop. In particular, the following expansion can be applied:*

$$\{\, [S[i] \to t[]] \to t[i]; [T[i] \to t[]] \to t[i] \,\}. \tag{5.11}$$

The result of this expansion is shown in Listing 5.18. Conversely, a contraction

of the form

$$\{\, [S[i] \to t[i]] \to t[]; [T[i] \to t[i]] \to t[] \,\} \tag{5.12}$$

or simply

$$\{\, t[i] \to t[] \,\} \tag{5.13}$$

can be applied to the code in Listing 5.18 to obtain the code in Listing 5.16.

5.6 Schedule

5.6.1 Schedule Definition and Representation

Definition 5.43 (Instance Set)**.** *A* schedule S *defines a strict partial order* $<_S$ *on the elements of the instance set.*

In particular, a schedule describes or prescribes the order in which the elements of the instance set are or should be executed. Note that some approaches do not keep track of a separate schedule but rather encode the execution order directly into the instance set(s) as explained in Alternative 5.8 Ordered Instance Set. Other approaches, at least those that perform program transformations, typically keep track of at least two schedules, one that represents the original execution order and that is called the *input schedule*, and one that represents the desired final execution order. The latter may be constructed incrementally from the input schedule or it may be computed from scratch based on the dependences as explained in Section 5.9 Operations.

A naive way of representing a schedule S would be to directly encode the strict partial order $<_S$ as a Presburger relation. However, such a relation would contain almost half of all possible pairs of statement instances and its representation would therefore be at least quadratic in the number of statements. This order is therefore invariably represented indirectly through some form of schedule.

One way of representing a schedule that is fairly close to that of an imperative program is in the form of a *schedule tree*. The two main ways of expressing execution order in an imperative program are compound statements, expressing that the constituent statements are executed in the given order, and loops, expressing the order in which different instances of the same statements are executed. The main types of nodes in a schedule tree correspond to these two mechanisms. In particular, a *sequence node* in a schedule tree expresses that its children are executed in the given order, while a *band node* expresses the order in which different instances of statements are executed. Each child of the sequence node is annotated with a Presburger set describing the statement instances represented by that child. This set is called the *filter* of the child. The order expressed by a band node is given by a tuple of multi-space piecewise quasi-affine expressions, which is called the *partial schedule* of the band. A statement instance is ordered before another statement instance by a band node if it is assigned a lexicographically smaller value by the partial schedule of the band node. For completeness, a *leaf node* is also introduced that is used

Note 5.13

Note 5.14

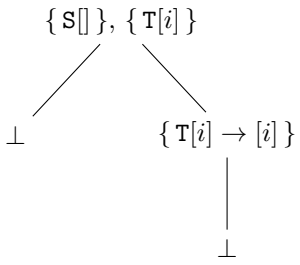


Figure 5.19: Schedule tree for the program fragment in Listing 5.2 on page 89

to represent the leaves of the schedule tree. With the introduction of such leaf nodes, all band nodes have exactly one child, while all leaf nodes have zero children. Leaf nodes are denoted by $\bot$ is some figures, but they will usually simply be omitted.

Example 5.44. *Consider once more the program fragment in Listing 5.2 on page 89. A schedule tree representation of the input schedule of this program fragment is shown in Figure 5.19. The root node is a sequence corresponding to the top-level sequence of statements in Listing 5.2, the statement* `S` *and the* `for`*-loop. The sequence node expresses that the single instance of the* `S` *statement is executed before all instances of the* `T` *statement. In this figure, the two Presburger sets describing the statement instances belonging to the two children are written inside the sequence node, separated by a comma. The first child only contains one statement instance and therefore does not need to express any further ordering. It is then simply a leaf node. The second child is a band node corresponding to the* `for`*-loop in Listing 5.2. It expresses that the instances* $\mathtt{T}[i]$ *are executed according to increasing values of* i*. Given the encoding of Example 5.4 on page 89, where this* i *corresponds to the value of the loop iterator, this means that the instances are executed in the order of the* `for`*-loop. The single child of this band node is again a leaf node.*

Algorithm 5.1 on the facing page shows how to determine the execution order of two statement instances by moving down the schedule tree. If a leaf node is reached, the schedule tree does not specify the order of the two instances. If the two instances belong to the same child of a sequence node, the algorithm moves down to that child. If the two instances belong to different children of a sequence node, then the order of these children determines the order of the statement instances. If a band node is reached, the values of the partial schedule evaluated at the two instances determines their order. If they get assigned the same value, then the algorithm moves down to the single child of the band node. Note that the execution order may not be a total order such that both $\boldsymbol{i} <_S \boldsymbol{j}$ and $\boldsymbol{j} <_S \boldsymbol{i}$ may be false. By determining the order between any pair of statements, the complete order relation can be constructed.

Input: Schedule S, statement instances $\boldsymbol{i}$ and $\boldsymbol{j}$
Output: True if $\boldsymbol{i} <_S \boldsymbol{j}$; false otherwise

```
Start at root of schedule tree S
while current node is not a leaf node do
    if current node is a sequence node then
        if i and j appear in same child then
            Move to common child
        else if i appears in earlier child then
            return true
        else
            return false
        end
    else
        Let P be the partial schedule of the current band node
        if P(i) = P(j) then
            Move to single child
        else if P(i) ≺ P(j) then
            return true
        else
            return false
        end
    end
end
return false
```

Algorithm 5.1: Execution order encoded by a schedule tree

Example 5.45. *The order relation defined by the schedule tree in Figure 5.19 is*

$$\{\, \mathtt{S}[] \to \mathtt{T}[i] : 0 \le i < n; \mathtt{T}[i] \to \mathtt{T}[i'] : 0 \le i < i' < n \,\}. \tag{5.14}$$

A schedule can also be encoded into a Presburger relation by essentially flattening the schedule tree into a single band node and then converting the resulting tuple of multi-space piecewise quasi-affine expressions into a Presburger relation. In particular, this means that the range of the relation lives in a single space and that the execution order is determined by the lexicographical order in this space. The flattening procedure is shown schematically in Algorithm 5.2 on the next page. The procedure returns either a leaf (if the input consists of only a leaf) or a band node. If the root of the input is a band node, then the result is the concatenation of that node with the flattening of the single child node. Concatenation means that essentially the flat range product of the partial schedules in computed. If the root of the input is a sequence node, then the children are assigned a sequence number that is combined with the partial schedule of the flattened child. If this flattened child is a leaf node, then it is treated as a zero-dimensional band node. If the flattened children do not all have the same number of members, then those with fewer members are padded

```
Input: Schedule (sub)tree
Output: Flattened schedule (sub)tree
if current node is a leaf node then
    return leaf
else if current node is a band node then
    flatten child of band node
    if child node is a leaf node then
        return band
    else
        concatenate current band with child
        return concatenated band
    end
else
    flatten children of sequence node
    pad lower-dimensional child bands (e.g., with zeros)
    let n be the number of children
    for i ← 0 to n − 1 do
        construct expression assigning i to statement instances of child i
        concatenate it with (padded) child schedule
    end
    take union of concatenations
    return union band
end
```

Algorithm 5.2: Flatten schedule tree

with arbitrary values, say zero. The union of all these combinations then forms the partial schedule of the flattened band node.

Example 5.46. *Consider the schedule tree in Figure 5.19 on page 112. The first child of the root node is a leaf and so does not need any further processing. From the point of view of the root node, it is treated as a zero-dimensional band node with partial schedule* $\{ \mathtt{S}[] \to [] \}$*. The second child of the root node is a band node with only a leaf node as child. This node therefore also does not require any further processing. Since the band of the first child is zero-dimensional, while that of the second child is one-dimensional, the first is padded to* $\{ \mathtt{S}[] \to [0] \}$*. The two band schedules are then prefixed by a number reflecting their position in the sequence, resulting in* $\{ \mathtt{S}[] \to [0, 0] \}$ *and* $\{ \mathtt{T}[i] \to [1, i] \}$*. Finally, the flattened sequence node has the union of these two as partial schedule, i.e.,*

$$\{ \mathtt{S}[] \to [0, 0]; \mathtt{T}[i] \to [1, i] \}. \tag{5.15}$$

Alternative 5.47 (Per-Statement Schedules)**.** *Some authors consider pet-statement schedules instead of a single schedule that describes the rel-*

```
void matmul(int M, int N, int K,
    float A[restrict static M][K],
    float B[restrict static K][N],
    float C[restrict static M][N])
{
#pragma scop
    for (int i = 0; i < M; ++i)
        for (int j = 0; j < N; ++j) {
I:          C[i][j] = 0;
            for (int k = 0; k < K; ++k)
U:              C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
#pragma endscop
}
```

Listing 5.20: Input file *matmul.c*

ative order of all statement instances. However, this only reflects a detail about how the schedule is represented. Even though different pieces of the schedule are stored across the statements, they cannot be interpreted independently of each other. Well-known representations of this type are "Kelly's abstraction" and "$2d+1$"-schedules.

Note 5.15

While flattening a schedule tree results in a Presburger relation that expresses the same ordering as the entire schedule, it is sometimes sufficient to know the ordering of statement instances at a given node in the schedule tree. In this case, the ordering imposed by the sequence nodes is irrelevant, since all statement instances that reach a given node are in the same child of each of the outer sequence nodes. The ordering is then given by the concatenation of the partial schedules of all outer band nodes. This concatenation is called the *prefix schedule* at the given node.

Example 5.48. *Consider the program shown in Listing 5.20. The prefix schedule at the band node corresponding to the outer* `for`*-loop is*

$$\{\, \mathtt{I}[i,j] \to [];\mathtt{U}[i,j,k] \to [] \,\}. \tag{5.16}$$

At the node corresponding to the second loop, it is

$$\{\, \mathtt{I}[i,j] \to [i];\mathtt{U}[i,j,k] \to [i] \,\} \tag{5.17}$$

and at the node corresponding to the third loop, it is

$$\{\, \mathtt{U}[i,j,k] \to [i,j] \,\}. \tag{5.18}$$

Finally, at the leaf corresponding to the `U` *statement, it is*

$$\{\, \mathtt{U}[i,j,k] \to [i,j,k] \,\}. \tag{5.19}$$

5.6.2 Representation in `isl`

In `isl`, a schedule tree is represented using the `isl_schedule` type. The nodes in this schedule tree may be of different types and the encoding is slightly different from that described in the previous section. In particular, band nodes appear directly in this encoding. A "sequence node", however, is represented as a pure sequence node that only expresses that its children are executed in order and *filter nodes* that describe the statement instances that are executed by any particular child. Moreover, in an `isl_schedule`, the root of the schedule tree is a *domain node* that contains the entire instance set. In a schedule extracted by `pet`, the tuple identifiers of the partial schedules of the band nodes are derived from the labels on the corresponding `for`-statements, if any.

Note 5.16

The textual representation of an `isl_schedule` is a YAML document. A node in a YAML document is either a scalar, a sequence or a mapping (associative array), where the elements in the sequence and the keys and values in the mapping are themselves YAML nodes. This document can be printed in either block style or in flow style. The individual nodes in a schedule tree are encoded in the YAML document as follows.

domain node A YAML mapping with as keys `domain` and `child`, and as corresponding values the instance set and the single child of the domain node.

band node A YAML mapping with as keys `schedule` and `child`, and as corresponding values the partial schedule of the band node and the single child of the band node. The `coincident` key is explained in Section 5.6.3 Encoding Parallelism.

sequence node A YAML mapping with as only key `sequence` and as corresponding value a YAML sequence with as entries the children of the sequence node.

filter node A YAML mapping with as keys `filter` and `child`, and as corresponding values the subset of the instance set preserved by the filter and the single child of the domain node.

If a node has a single child and if this child is a leaf node, then the child may be omitted from the textual representation. The set node and its representation are introduced in Section 5.6.3 Encoding Parallelism.

Example 5.49. *Consider the schedule below, which is the `isl` representation of the schedule in Figure 5.19 on page 112. The schedule is printed twice, once in block format and once in flow format. In order to force printing in flow format, the second instance is printed as part of a list. The structure of the `isl` representation is the same as that of the schedule tree in Figure 5.19 on page 112. The main differences are the extra domain node in the root and the omission of the leaf nodes.*
`iscc` input (`inner_schedule.iscc`) with source in Listing 5.5 on page 93:

```
P := parse_file "demo/inner.c";
print P[4];
print (P[4],0);
```

iscc invocation:

```
iscc < inner_schedule.iscc
```

iscc output:

```
domain: "[n] -> { T[i] : 0 <= i < n; S[] }"
child:
  sequence:
  - filter: "[n] -> { S[] }"
  - filter: "[n] -> { T[i] }"
    child:
      schedule: "[n] -> L[{ T[i] -> [(i)] }]"

({ domain: "[n] -> { T[i] : 0 <= i < n; S[] }", child: {
    ↪ sequence: [ { filter: "[n] -> { S[] }" }, { filter: "
    ↪ [n] -> { T[i] }", child: { schedule: "[n] -> L[{ T[i]
    ↪  -> [(i)] }]" } } ] } }, 0)
```

Navigation through a schedule tree in `isl` is done by means of an object of type `isl_schedule_node`, which points to a specific node in a tree. A pointer to root of a schedule tree can be obtained using `isl_schedule_get_root`, while `isl_schedule_node_parent` and `isl_schedule_node_child` can be used to move up and down the tree. The schedule tree into which a schedule node is pointing can be retrieved using `isl_schedule_node_get_schedule`. This is especially useful if the schedule node has been used to modify the schedule tree. The textual representation of an `isl_schedule_node` is the same as that of the tree into which it points, except that in block style the node it points to is marked with the comment "YOU ARE HERE". The `python` interface prints such objects in block style.

Example 5.50. *The transcript below illustrates how to move down a schedule tree and what the textual representation looks like.*
`python` input (`inner_schedule.py`) with source in Listing 5.5 on page 93:

```
import isl
import pet

pet.options.set_autodetect(True)
scop = pet.scop.extract_from_C_source("demo/inner.c",
                                      "inner")
schedule = scop.get_schedule()
node = schedule.get_root().child(0).child(1).child(0)
print node
```

`python` invocation:

```
python < inner_schedule.py
```

python output:

```
domain: "[n] -> { T[i] : 0 <= i < n; S[] }"
child:
  sequence:
  - filter: "[n] -> { S[] }"
  - filter: "[n] -> { T[i] }"
    child:
      # YOU ARE HERE
      schedule: "[n] -> L[{ T[i] -> [(i)] }]"
```

A Presburger relation representation of an `isl_schedule` can be obtained using the `isl_schedule_get_map` function. The result is an `isl_union_map`. In `iscc`, this operation is called `map`.

Example 5.51. *The transcript below shows how to convert the schedule tree of Example 5.49 on page 116 to the flattened representation* (5.15) *of Example 5.46 on page 114.*
`iscc` input (`flatten.iscc`) with source in Listing 5.5 on page 93:

```
P := parse_file "demo/inner.c";
print map(P[4]);
```

`iscc` invocation:

```
iscc < flatten.iscc
```

`iscc` output:

```
[n] -> { T[i] -> [1, i]; S[] -> [0, 0] }
```

The prefix schedule at a given node can be obtained in different representations through one of these functions:

- `isl_schedule_node_get_prefix_schedule_multi_union_pw_aff`
- `isl_schedule_node_get_prefix_schedule_union_pw_multi_aff`
- `isl_schedule_node_get_prefix_schedule_union_map`

Other operations include computing the pullback of a schedule with respect to a multi-space piecewise tuple of quasi-affine expressions through a call to `isl_schedule_pullback_union_pw_multi_aff`. In the `python` interface, this function is called `pullback`.

5.6.3 Encoding Parallelism

As explained in Section 5.1 Main Concepts, the order relation $<_S$ defined by a valid schedule S needs to include the dependences D. However, this order relation need not be total, meaning that there may be some pairs of statement instances for which the first is not ordered before the second and the second is not ordered before the first either. Such pairs of statement instances are then allowed to be executed simultaneously, i.e., in parallel, by the schedule. Due to the above-mentioned constraint, this can only happen if there are no dependences (directly or indirectly) between the two statement instances.

One way of exploiting such parallelism is to construct an equivalence relation E that contains the order relation $<_S$ as a subset. That is,

$$(<_S) \subseteq E. \tag{5.20}$$

The cells in the corresponding partition are completely independent, in the sense that no pair of elements from distinct cells are ordered with respect to each other. This equivalence relation is typically specified as the equivalence kernel of a function called a *placement*. In particular, the placement maps statement instances to (virtual) processors that can execute their instances in parallel with those of other processors. The placement is called trivial if all statement instances are mapped to the same processor.

Example 5.52. *Consider the dependence relation*

$$D = \{ \mathrm{S}[i,j,k] \to \mathrm{S}[i,j+1,k'] \}. \tag{5.21}$$

The dependence relation can be extended to the order relation

$$O = \{ \mathrm{S}[i,j,k] \to \mathrm{S}[i,j',k'] : j' > j \}, \tag{5.22}$$

which is therefore a valid order with respect to the dependences. The equivalence kernel of the function

$$f(\mathrm{S}[i,j,k]) \mapsto i \tag{5.23}$$

is the equivalence relation

$$\begin{aligned} E &= \{ \mathrm{S}[i,j,k] \to \mathrm{S}[i',j',k'] : f(\mathrm{S}[i,j,k]) = f(\mathrm{S}[i',j',k']) \} \\ &= \{ \mathrm{S}[i,j,k] \to \mathrm{S}[i,j',k'] \}, \end{aligned} \tag{5.24}$$

which in turn is a further extension of the order relation. That is,

$$D \subseteq O \subseteq E. \tag{5.25}$$

This means that the instances $\mathrm{S}[i,j,k]$ *can be mapped to different processors according to the value of* i *without violating any dependences. The transcript below verifies* (5.25) *and checks that* O *is indeed a strict partial order.*
`iscc` *input (*`placement.iscc`*):*

```
D := { S[i,j,k] -> S[i,j+1,k'] };
O := { S[i,j,k] -> S[i,j',k'] : j' > j };
P := { S[i,j,k] -> [i] };
E := P . P^-1;
print "Dependences form subset of order relation:";
D <= O;
print "Order relation is a strict partial order:";
(O . O <= O) * (O * O^-1 = {});
print "Order relation is subset of equivalence relation:";
O <= E;
print "Equivalence relation:";
print E;
```

iscc invocation:

```
iscc < placement.iscc
```

iscc output:

```
"Dependences form subset of order relation:"
True
"Order relation is a strict partial order:"
True
"Order relation is subset of equivalence relation:"
True
"Equivalence relation:"
{ S[i, j, k] -> S[i, j', k'] }
```

Note that a direct application of Algorithm 5.1 on page 113 to derive an execution order from a schedule tree that contains at least one band or sequence node will never produce a relation that allows for an extension to the equivalence kernel of a non-trivial placement function. Instead, if a non-trivial placement is being used, the order relation is defined to be the intersection of the order relation derived from the schedule and the equivalence kernel of the placement function. It is then this intersection that needs to contain all the dependences for the combination of placement and schedule to be valid.

The use of a (global) placement can in general only exploit some of the available parallelism. In particular, any pair of instances that are connected through an *undirected* path in the order relation cannot be executed in parallel this way. For example, two instances that both depend on a third instance cannot simply be mapped to separate processors without any synchronization or reduplication of instances since they both need to be executed after this third instance, while the latter can only be mapped to one of the two processors. If synchronization is allowed, then the two instances can still be executed in parallel. In particular, if the ancestors of a node in a schedule tree already ensure that the third instance is executed before the other two instances, then those two instances can be executed in parallel at that point in the schedule tree.

In order to exploit such local parallelism, a local placement can be constructed such that its equivalence relation satisfies (5.20), when restricted to the pairs of statement instances that are not already scheduled apart by the ancestors of the node. That is, (5.20) only needs to be satisfied for pairs of statement instances with the same value for the prefix schedule at the current node.

Example 5.53. *Consider once more the dependence relation* (5.21) *extended to the order relation* (5.22)*. Furthermore, assume that the prefix schedule at a given node in the schedule tree is*

$$\{ \mathrm{S}[i,j,k] \mapsto [j] \} \tag{5.26}$$

and take as placement the function

$$f'(\mathrm{S}[i,j,k]) \mapsto k. \tag{5.27}$$

The corresponding equivalence kernel is

$$\begin{aligned} E' &= \{ \mathrm{S}[i,j,k] \to \mathrm{S}[i',j',k'] : f'(\mathrm{S}[i,j,k]) = f'(\mathrm{S}[i',j',k']) \} \\ &= \{ \mathrm{S}[i,j,k] \to \mathrm{S}[i',j',k] \}. \end{aligned} \tag{5.28}$$

Clearly, E' does not contain O (5.22) *as a subset. However, after restriction to the pairs of statement instances with the same value for the prefix schedule, i.e., after intersecting with*

$$L = \{ \mathrm{S}[i,j,k] \mapsto \mathrm{S}[i',j,k'] \} \tag{5.29}$$

the relation $O \cap L \subseteq E' \cap L$ does hold. This computation is illustrated by the transcript below.
`iscc` input (`placement2.iscc`):

```
D := { S[i,j,k] -> S[i,j+1,k'] };
O := { S[i,j,k] -> S[i,j',k'] : j' > j };
P := { S[i,j,k] -> [k] };
E := P . P^-1;
Prefix := { S[i,j,k] -> [j] };
L := Prefix . (Prefix^-1);
print "Order relation is subset of equivalence relation:";
O <= E;
print "Locally, order relation is subset:";
(L * O) <= (L * E);
print "Local order relation:";
print (L * O);
print "Local equivalence relation:";
print (L * E);
```

`iscc` invocation:

```
iscc < placement2.iscc
```

iscc output:

```
"Order relation is subset of equivalence relation:"
False
"Locally, order relation is subset:"
True
"Local order relation:"
{  }
"Local equivalence relation:"
{ S[i, j, k] -> S[i', j, k] }
```

Since these local placements are tied to a specific position in the schedule, they are usually integrated in the schedule itself. In particular, it has become customary to construct schedules that determine a total order rather than just a partial order, but then to explicitly mark some of the schedule dimensions as representing parallel execution rather than sequential execution. That is, these schedule dimensions do not define an order, but rather define the groups of statement instances that can be executed independently of each other (at that position in the schedule).

Example 5.54. *The integrated schedule*

$$\{ \mathrm{S}[i,j,k] \mapsto [i,j,k] \}, \tag{5.30}$$

with both the outermost and the innermost dimension marked as parallel combines the placement (5.23) *of Example 5.52 on page 119 as well as the prefix schedule* (5.26) *and the local placement* (5.27) *of Example 5.53 on the previous page.*

In `isl`, a member of a band node can be marked *coincident* to indicate that the corresponding multi-space piecewise quasi-affine expression identifies groups of statement instances that are independent of each other. This property can be set using `isl_schedule_node_band_member_set_coincident` and read off using `isl_schedule_node_band_member_get_coincident`. The first takes two extra arguments, the index of the band member in the band and the new value of the property. The second takes one extra argument, the index of the band member in the band. In the YAML representation, coincidence is expressed through an (optional) additional `coincident` key on band nodes with as value a sequence of 0/1 values indicating whether the corresponding band member is considered coincident. If the key is missing, then all values are assumed to be 0, i.e., no member is considered coincident.

The sequence node has no coincident attribute. Instead another node type, the *set node*, has been introduced that expresses that its children can be executed independently of each other. Its YAML representation is as follows.

set node A YAML mapping with as only key `set` and as corresponding value a YAML sequence with as entries the children of the set node.

Example 5.55. *The following transcript illustrates how to explicitly mark some schedule band members as coincident. Since the band nodes involved all have a single member, the position of the band member is always 0 in this example.*
python input (coincident.py) with source in Listing 5.20 on page 115:

```
import isl
import pet

scop = pet.scop.extract_from_C_source("demo/matmul.c",
                                      "matmul")
schedule = scop.get_schedule()
node = schedule.get_root().child(0)
node = node.band_member_set_coincident(0, True)
node = node.child(0)
node = node.band_member_set_coincident(0, True)
print node
```

python invocation:

```
python < coincident.py
```

python output:

```
domain: "[N, M, K] -> { I[i, j] : 0 <= i < M and 0 <= j < N;
    ↪  U[i, j, k] : 0 <= i < M and 0 <= j < N and 0 <= k <
    ↪ K }"
child:
  schedule: "[M, N, K] -> L_0[{ I[i, j] -> [(i)]; U[i, j, k]
      ↪  -> [(i)] }]"
  coincident: [ 1 ]
  child:
    # YOU ARE HERE
    schedule: "[M, N, K] -> L_1[{ I[i, j] -> [(j)]; U[i, j,
        ↪ k] -> [(j)] }]"
    coincident: [ 1 ]
    child:
      sequence:
      - filter: "[M, N, K] -> { I[i, j] }"
      - filter: "[M, N, K] -> { U[i, j, k] }"
        child:
          schedule: "[M, N, K] -> L_2[{ U[i, j, k] -> [(k)]
              ↪ }]"
```

5.7 Context

The *context* is a unit set that keeps track of conditions on the constant symbols. These conditions can be used to simplify various computations during polyhedral compilation. For example, there may be a dependence between statement

instances that only occurs if the constant symbols satisfy some relation. If the context contradicts this relation, then these dependences may be ignored within that context. It can be useful to distinguish two types of contexts.

- The *known context* is non-empty for all values of the constant symbols for which the input program may be executed. That is, the input program is known not be executed for values of the constant symbols for which the known context is empty.
- The *assumed context* is non-empty for the values of the constant symbols that are considered during the analysis and/or transformation.

The known context collects constraints on the constant symbols that can be derived from the input program in the sense that this program would not run (properly) if these constraints do not hold. For example, the size expressions
Note 5.17 in an array declaration need to be non-negative for the program to make any sense, meaning that the constant symbols involved are known to satisfy this non-negativity constraint. Any values of constant symbols that certainly lead to undefined behavior can also be excluded from the known context. Examples of undefined behavior in C include division by zero, out-of-bounds accesses and (signed) integer overflow. The bounds on constant symbols derived from the types of the corresponding program variables also belong to the known context.

Example 5.56. *Consider the program in Listing 5.20 on page 115. Even though the formal arguments* `M`, `N` *and* `K` *are declared to be (possibly negative) integers, their use in the size expressions of the other arguments implies that they are all non-negative. The corresponding constant symbols may therefore also be assumed to be non-negative.*

The assumed context collects constraints that make it easier to perform polyhedral compilation. A typical use case are constraints that prevent aliasing. For example, a negative array index expression is not necessarily invalid in C, especially in the case of multi-dimensional arrays, but allowing them does mean that multiple sequences of index expressions may refer to the same array element. The same holds for index expressions with a value that is greater than the corresponding array size. The assumed context may be treated as a subset of the known context.

Example 5.57. *Consider an array that is declared as follows.*

```
int A[M][N];
```

The expression `A[2][-1]` *is perfectly valid in C, but if refers to the same element as* `A[1][N-1]`*. That is, allowing* `A[2][-1]` *would result in aliasing.*

Currently, `pet` does not make a distinction between a known context and an assumed context. The context it computes is essentially a known context, but is also includes some constraints that should in principle only be taken into account for an assumed context. The function `pet_scop_get_context`

```
void f(int n, int m, int S,
        int D[const restrict static S])
{
        for (int i = 0; i < n; i++) {
                D[i] = D[i + m];
        }
}
```

Listing 5.21: Input file *overflow.c*

can be used to extract this context from a `pet_scop`. Depending on the state of the `--signed-overflow` option, the context also includes constraints that avoid signed integer overflow. The state of this option may be changed by calling the `pet_options_set_signed_overflow` function. The possible values are `PET_OVERFLOW_AVOID` and `PET_OVERFLOW_IGNORE`.

Example 5.58. *The following transcript prints the context of the program in Listing 5.20 on page 115. The lower bounds on the constant symbols are derived from the fact that they are used as size expressions and therefore need to be non-negative. The upper bounds are derived from the types of the corresponding program variables.*
***python** input (`context.py`) with source in Listing 5.20 on page 115:*

```
import isl
import pet

scop = pet.scop.extract_from_C_source("demo/matmul.c",
                                      "matmul")
print scop.get_context()
```

***python** invocation:*

```
python < context.py
```

***python** output:*

```
[N, M, K] -> {  : 0 <= N <= 2147483647 and 0 <= M <=
    ↪ 2147483647 and 0 <= K <= 2147483647 }
```

Example 5.59. *Consider the program in Listing 5.21. The transcript below shows the context with and without taking into account that signed integers should not overflow. In the first output, the context consists of two disjuncts, one corresponding to the state where the loop is not executed (*$\mathrm{n} \leq 0$*) and one corresponding to the state where the loop is executed (*$\mathrm{n} > 0$*). In the second disjunct,* m *is also required to be non-negative because element* $i + \mathrm{m}$ *of* `D` *is accessed for* $i = 0$*. In the second output (where integer overflow is taken into account), the constraint* $\mathrm{m} \leq 2147483647$ *is further restricted to* $\mathrm{m} + \mathrm{n} \leq 2147483648$*.*

python input (overflow.py) with source in Listing 5.21 on the previous page:

```
import isl
import pet

pet.options.set_autodetect(True)
pet.options.set_signed_overflow(pet.overflow.ignore)
scop = pet.scop.extract_from_C_source("demo/overflow.c",
                                      "f")
print scop.get_context()
pet.options.set_signed_overflow(pet.overflow.avoid)
scop = pet.scop.extract_from_C_source("demo/overflow.c",
                                      "f")
print scop.get_context()
```

python invocation:

```
python < overflow.py
```

python output:

```
[S, n, m] -> {  : 0 <= S <= 2147483647 and -2147483648 <= n
    ↪ <= 2147483647 and -2147483648 <= m <= 2147483647 and
    ↪ (n <= 0 or (n > 0 and m >= 0)) }
[S, n, m] -> {  : 0 <= S <= 2147483647 and -2147483648 <= n
    ↪ <= 2147483647 and m >= -2147483648 and ((n <= 0 and m
    ↪  <= 2147483647) or (n > 0 and 0 <= m <= 2147483648 -
    ↪ n)) }
```

Note 5.18

It is also possible for the user to express some known constraints on the state of the variables using a `__pencil_assume` statement. In particular, the argument of this syntactic function call is a boolean expression that is guaranteed by the user to hold at the point in the program where it is executed. Currently, `pet` only takes into account expressions that are quasi-affine in the parameters and the outer loop iterators. Note that if `pet` were to make a distinction between the known context and the assumed context, then these constraints would end up in the known context since they are guaranteed to hold.

Example 5.60. *Consider the program in Listing 5.22 on the facing page, which is identical to that in Listing 5.21 on the previous page, except that a "call" to* `__pencil_assume` *has been added. Specifically, the user asserts that this point will only be reached when* m *is greater than* n*. Since these two variables can be treated as constant symbols, this means that this condition holds for the entire analyzed fragment and therefore ends up in the context. The transcript below prints this context.*
python input (assume.py) with source in Listing 5.22 on the facing page:

```
import isl
import pet
```

```
void f(int n, int m, int S,
        int D[const restrict static S])
{
        __pencil_assume(m > n);
        for (int i = 0; i < n; i++) {
                D[i] = D[i + m];
        }
}
```

Listing 5.22: Input file *assume.c*

```
pet.options.set_autodetect(True)
pet.options.set_signed_overflow(pet.overflow.ignore)
scop = pet.scop.extract_from_C_source("demo/assume.c", "f")
print scop.get_context()
```

python invocation:

```
python < assume.py
```

python output:

```
[S, n, m] -> {  : 0 <= S <= 2147483647 and n >= -2147483648
    ↪ and n < m <= 2147483647 }
```

5.8 Polyhedral Statements

The polyhedral statement is the basic unit of execution for the purpose of representing a piece of code using a polyhedral model. That is, the elements of the instance set represent instances of these polyhedral statements.

The choice of the basic unit of execution somewhat depends on the input from which the polyhedral model is extracted. Broadly speaking, there are three classes of inputs.

- The input language may have been specifically designed for polyhedral compilation. In this case, there is a direct correspondence between the language and the model and it is therefore up to the user to decide what constitutes a polyhedral statement. Note 5.19
- The input language may be a standard programming language such as C or Fortran, typically with restrictions on the kinds of constructs that can be used. In this case, a polyhedral statement usually corresponds to an expression statement in the source program. However, a polyhedral statement may also consist of a collection of program statements, or, conversely, a program statement may be broken up into several polyhedral statements. Note 5.20 Note 5.21

- The input may be the internal representation of a compiler. It may be slightly more difficult to extract a polyhedral representation from such an internal representation because loop structures and in particular loop induction variables may not be readily available. On the other hand, several ways of expressing the same control flow can be mapped to the same internal representation before the extraction through canonicalization of that representation. In particular, code written in different source languages can typically be treated in the same way. The polyhedral statements usually correspond to the basic blocks in the internal representation.

Note 5.22

As explained in Section 5.2.3 Representation in `pet` , the polyhedral statements extracted by `pet` are either expression statements or larger statements that contain dynamic control. For each polyhedral statement, `pet` keeps track of additional information. This additional information includes the subset of the instance set containing instances of the polyhedral statement and a representation of the corresponding program statement in the form of a tree. This tree is needed to print the bodies of the statements of a polyhedrally transformed program printed in the form of source code. For each memory access in this tree, `pet` also keeps track of the reference identifier, the access relations restricted to this access and the index expression. This *index expression* is a tuple of piecewise quasi-affine expressions that collects the expressions that appear in the access in the program text. In general, it is different from the access relations since it may reference a (single) slice of an array or an entire structure, while the access relations always reference individual data elements. The index expression is kept track of separately since it may be needed for printing the transformed code and it may be difficult or even impossible to extract from the access relations, especially if the access appears in a function call and the access relations have been set based on the accesses inside the body of the called function.

Example 5.61. *Consider the access to `A` in statement `S` of the program in Listing 5.11 on page 99. The corresponding index expression is simply*

```
[n] -> { S[] -> A[] }
```

while the (write) access relation is

```
[n] -> { S[] -> A[o0, o0] : 0 <= o0 < n }
```

as shown in Example 5.22 on page 99.

Similarly, consider the write access to `A` in statement `S` of the program in Listing 5.15 on page 104. The corresponding index expression is

```
{ S[] -> A[(0)] }
```

while the access relation is

```
{ S[] -> A_re[A[0] -> re[]];
  S[] -> A_im[A[0] -> im[]] }
```

as shown in Example 5.29 on page 104.

5.9 Operations

This section briefly describes some of the major steps in polyhedral compilation.

Extraction The extraction phase extracts a polyhedral model from the input code. The output of this step typically consists of the instance set, the access relations and the original schedule. It may also include the dependence relations if these can be readily read off from the input code or if they are computed during the extraction phase. In some cases, the input code does not have an inherent execution order, in which case no original schedule is extracted. Note 5.23

Dependence analysis Dependence analysis takes the instance set, the access relations and the original schedule as input and produces dependence relations. Dependence analysis is described in more detail in Chapter 6 Dependence Analysis. Note 5.24

Scheduling Scheduling takes the instance set and the dependence relations as input and computes a new schedule. This schedule may be computed from scratch based on the dependence relations or it may be constructed incrementally through modifications of the original schedule, which is then an additional input. Note 5.25

AST generation AST generation, also known as polyhedral scanning and code generation takes an instance set and a schedule as input and produces an Abstract Syntax Tree (AST) that executes the elements of the instance set in the order specified by the schedule. Note 5.26

Notes

5.1. Examples of polyhedral compilation frameworks where the basic execution entity is a basic block are GCC's GRAPHITE (Trifunovic et al. 2010) and LLVM's Polly (Grosser, Größlinger, et al. 2012).
5.2. These names are derived from those of similar concepts, e.g., the set of iterations of a perfectly nested loop, in precursors of polyhedral compilation. The term "iteration domain" appears to have been introduced by Irigoin and Triolet (1986). Irigoin (1987) uses "domaine d'itérations" as a translation for "index set".
5.3. The author knows of no practical use case for a must-read access relation.
5.4. Curiously, some authors, e.g., Trifunovic et al. (2010), treat scalars as one-dimensional arrays with a single integer index equal to zero. Possibly, this stems from a desire to treat scalars as actual array accesses, even at the level of the internal representation of the compiler.
5.5. Declaring a function argument of the form `type A[a][b][c]` in C is just a fancy way of writing `type (*A)[b][c]`. That is, there is no information about how many elements of type `type[b][c]` the pointer `A` points to. Adding the `static` keyword, as in `type A[static a][b][c]` means that `A` points to a

region of memory that holds at least `a` such elements. Since this specification only allows the function to access those first `a` elements, `pet` takes the declaration to mean that only those elements may effectively be accessed by the function.

5.6. Technically the `restrict` keyword only means that the annotated pointers do not alias if they are used to *write* to memory, but this is effectively the only case that the (polyhedral) compiler needs to worry about.

5.7. See Feautrier (1998) for a proposal on how to represent recursive trees in a way that was *inspired* by polyhedral compilation.

5.8. An alternative would be to start from the outer field access and to put further field accesses into the ranges of the wrapped relations. The choice is fairly arbitrary. Arguably, using n-ary relations would be a more appropriate representation, but they are not currently supported by `isl`.

5.9. The value-based and memory-based terminology was introduced by Pugh and Wonnacott (1994).

5.10. Some authors, e.g., Yang et al. (1995) and Darte and Vivien (1997), use the term dependence polyhedron to refer to a polyhedron of dependence *distances* instead.

5.11. Darte et al. (2005) present an algorithm for computing contractions and provides an overview of earlier techniques. Darte et al. (2004) provide even more details.

5.12. Feautrier (1988a) describes how to compute an expansion.

5.13. Schedule trees were introduced by Verdoolaege, Guelton, et al. (2014) and refined by Grosser, Verdoolaege, et al. (2015).

5.14. The other node types are deferred to a later version of this tutorial.

5.15. Kelly's abstraction was introduced by Kelly and Pugh (1995) and is called "my new abstraction" by Kelly (1996, Section 2.2.2). $2d + 1$-schedules were introduced by Cohen, Girbal, et al. (2004), Cohen, Sigler, et al. (2005), and Girbal et al. (2006). They do not form a "pure" schedule representation as some operations such as tiling require modifications to the instance set representation.

5.16. The YAML specification is available from Ben-Kiki and Evans (2009).

5.17. Technically, the C standard requires arrays to be of size at least one, but most compilers allow arrays of size zero.

5.18. The `__pencil_assume` predicate was introduced by Baghdadi et al. (2015) and Verdoolaege (2015b)

5.19. AlphaZ (Yuki, Gupta, et al. 2012) is an example of a language that was specifically designed with polyhedral compilation in mind.

5.20. An example of an approach where several program statements are combined into a single polyhedral statement is that of Mehta and Yew (2015). In particular, consecutive expression statements in the program text are considered as a single polyhedral statement.

5.21. An example of an approach where a single program statement gives rise to several polyhedral statements is that of Stock et al. (2014).

5.22. See Note 5.1 for examples of frameworks using basic blocks as polyhedral statements.

5.23. Commonly used tools for extracting a polyhedral model include `clan` (Bastoul 2008) and `pet` (Verdoolaege and Grosser 2012). Many polyhedral frameworks include a tailored extraction procedure.
5.24. Maslov (1994) and Klimov (2014) advocate performing dependence analysis during the extraction phase.
5.25. Feautrier (1992a) and Feautrier (1992b) describe one of earliest scheduling algorithms in polyhedral compilation. Darte, Robert, et al. (2000) provides an overview of the scheduling algorithms that were known at the time. One of the most popular scheduling algorithms in current use is the "Pluto" scheduling algorithm of Bondhugula et al. (2008) and Acharya and Bondhugula (2015).
5.26. Polyhedral scanning (Ancourt and Irigoin 1991) more precisely describes one of the core steps in AST generation, which is to generate an AST for visiting all the points in a polyhedron. The complete procedure is called scanning unions of polyhedra by Quilleré et al. (2000). The term "code generation" is used by, e.g., Kelly, Pugh, and Rosser (1995) and Bastoul (2004), but many other authors use the same term for other or more general functionality. The most recent algorithms for AST generation are described by Bastoul (2004), Chen (2012), and Grosser, Verdoolaege, et al. (2015).