Presburger Formulas and Polyhedral Compilation

v0.02-13-g53eb23d

Sven Verdoolaege
Polly Labs and KU Leuven

August 21, 2021

Contents

Chapter 1

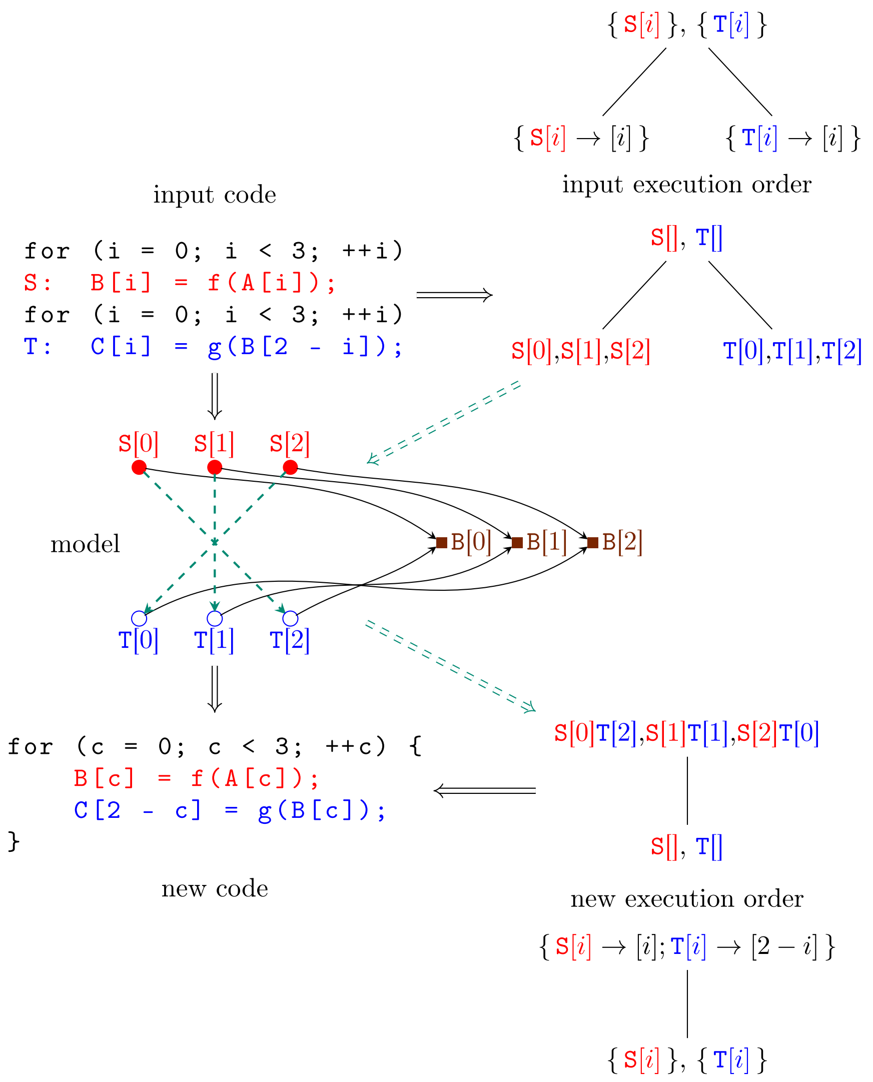
Introduction

This tutorial is intended for anyone who wants to learn about polyhedral compilation. The main focus is on the concepts and operations involved and on Note 1.1 how to use them to accomplish basic tasks. There is some variation in the definitions of the core polyhedral compilation concepts. The flavor used in this tutorial is that of `isl` based tools such as `PPCG`. However, other commonly Note 1.2 used variations will also be mentioned to help the reader understand some of the polyhedral compilation literature. These variations will be presented as **Alternatives** and may be skipped by readers only interested in the `isl` terminology. Although the tutorial tries to cover many topics in polyhedral compilation, it may be somewhat biased and it does not claim to be complete. In fact, the current preliminary version is still very incomplete.

1.1 Polyhedral Compilation

Broadly speaking, polyhedral compilation refers to a collection of program anal- Note 1.3 ysis and compilation techniques that reason about individual "dynamic execution instances" in a program as well as relations between pairs of such instances. A dynamic execution instance refers to an operation or a group of operations as it is executed at run-time, rather than as it appears in the program text. For example, if a statement appears in a loop in the program, then there would be as many instances as there are iterations in the loop. Since there can be many, possibly even an infinite number, of such instances in the program, they are typically described intensionally rather than extensionally, using mathematical objects such as polyhedra and Presburger formulas. This is explained in more detail in Chapter 3 Presburger Sets and Relations. Note that the use of polyhedra is neither required nor sufficient in polyhedral compilation, in the sense that it is perfectly possible to perform polyhedral compilation without polyhedra and that there are techniques outside polyhedral compilation such as Note 1.4 abstract interpretation and array region analysis that may also use polyhedra. Note 1.5

The following example provides a glimpse of what is to come in this tutorial. It should be noted that the example only shows one particular use case of



```
for (i = 0; i < 3; ++i)
S:  B[i] = f(A[i]);
for (i = 0; i < 3; ++i)
T:  C[i] = g(B[2 - i]);
```

```
for (c = 0; c < 3; ++c) {
    B[c] = f(A[c]);
    C[2 - c] = g(B[c]);
}
```

Figure 1.1: A typical use case of polyhedral compilation

polyhedral compilation.

Example 1.1. *Consider the code fragment near the top left of Figure 1.1 on the facing page. This is a very simple piece of code with two loops, each containing a single statement. In this example, the "dynamic execution instances" under consideration will be the executions of these two statements. Each statement is executed three times, once for every iteration of the surrounding loop. Each statement instance will be identified by the label on the statement (`S` or `T`) and an integer. In this example, this integer is equal to the value of the loop iterator when this instance is executed.*

The tree to the right of the code expresses the execution order of the statement instances. The tree comes in two forms, an intensional one on top and an (informal) extensional one below. Let us first consider the extensional one. The root node of this tree expresses that all instances of the `S` statement are executed before all instances of the `T` statement. The children of this root node then express the order of `S` and `T` statement instances respectively. In particular, `S[0]` is executed before `S[1]`, which in turn is executed before `S[2]`. In the end, the tree specifies that the statement instances are executed in the following order: `S[0]`, `S[1]`, `S[2]`, `T[0]`, `T[1]`, `T[2]`, but it specifies this order in a more structured way that matches the control of the input code. In particular, the root node corresponds to the outer sequence of the two `for` statements, while the leaves correspond to the `for` loops. The intensional tree on top provides a more abstract representation of the same information. Instead of explicitly listing the instances of the `S` statement in their execution order, this tree expresses that the instances are executed according to increasing values of the identifying integer i, which in this case corresponds to the value of the iterator of the surrounding loop. The notation in this tree will be explained in Chapter 2 Sets of Named Integer Tuples.

The graph underneath the code depicts the individual statement instances along with the elements of the `B` array that are accessed by the code fragment. An arrow is drawn between each statement instance and each array element accessed by that statement instance. For example, the statement instance `S[2]` writes to array element `B[2]` through the array reference `B[i]`. Similarly, the statement instance `T[0]` reads from the same array element through the array reference `B[2 - i]`. The dashed arrows are explained below.

Let us now assume that we want to change the order in which the statement instances are executed. There are many different orders that can be chosen. In fact, since there are 6 statement instances, there are $6! = 720$ different (sequential) orders. If groups of statement instances are allowed to be executed simultaneously, then there are even more possibilities. However, only some of these orderings preserve the semantics of the original program. In particular, an ordering that places `T[0]` before `S[2]` would not preserve the semantics of the program because `T[0]` would read `B[2]` before it is written by `S[2]` and might therefore read a value that is different from the value read in the original program. In other words, `T[0]` needs to be executed after `S[2]`. In the figure, this is expressed by a dashed arrow from `S[2]` to `T[0]`. This ordering constraint is *determined by the fact that both access the same memory element and that* `S[2]` *is executed before* `T[0]` *according to the input execution order. The computation of such ordering constraints is the subject of Chapter 6 Dependence Analysis.*

One of the possible execution orders that do preserve the semantics of the program is shown on the bottom right. As in the case of the input execution order, the new execution order is also expressed in an extensional (top) and an intensional (bottom) way. In this case, the root of the tree first orders the statement instances according to their identifying integers, where instance i of `S` *is grouped together with instance $2 - i$ of* `T`*. The leaf node then orders the* `S` *instance in the group before the* `T` *instance. In the end, this tree specifies that the statement instances are executed in the following order:* `S[0]`, `T[2]`, `S[1]`, `T[1]`, `S[2]`, `T[0]`*. The code to the left of the tree corresponds to this execution order.*

1.2 Tools

This section describes the tools that will be used to demonstrate the concepts discussed in the tutorial. The inputs to these illustrations are available as attachments to the electronic version of this tutorial. They are accessible through the file names mentioned near the illustration. The method for extracting these attachments depends on your PDF viewer. The outputs have been generated using `barvinok-0.41.5-3-g21a2c3b1f8`.

1.2.1 `pet`

Note 1.6 `pet` is a library for extracting (parts of) a polyhedral model (see Chapter 5 Polyhedral Model) from C source code. By default, polyhedral models are extracted from code fragments enclosed in a `#pragma scop` and a
Note 1.7 `#pragma endscop`. If the `--autodetect` option is turned on, then `pet` will search for appropriate fragments to extract itself, but it will detect at most one such fragment inside each function body. When `pet` is used as part of a tool, e.g., `iscc` below, then there is usually no need to compile `pet` separately.

1.2.2 `iscc`

`iscc` is an interactive tool for manipulating sets of named integer tuples (see
Note 1.8 Chapter 2 Sets of Named Integer Tuples) and related objects that is distributed along with the `barvinok` distribution. Each command applies zero or more operations to objects and is terminated by a semicolon. The operations are evaluated left to right. That is, there is no operator precedence. However, the evaluation order can be changed by placing parentheses around subexpressions. The result of the computation may be assigned to an `iscc` variable through the assignment operator, "`:=`". In order to avoid potential conflicts with operator names, it is best to only use variable names that start with a capital letter. The `iscc` variables are dynamically typed. The `typeof` operator can be used to query the type of the value currently assigned to a variable.

If the result of a computation is not assigned to a variable, then it is printed. In interactive mode, the printed result is also assigned to a numbered variable that can be reused in later computations just like a named variable. This assignment can be suppressed by using the `print` operator. The outputs shown in this tutorial are all obtained in non-interactive mode, so they do not contain any assignments to numbered variables. In the examples, `iscc` is invoked with the input taken from a file. The commands in a file can also be executed in interactive mode through the `source` operator. This operator takes a filename (a string enclosed in double quotes) as argument and executes the commands in the corresponding file.

Some operations return a list of values. The elements in such a list can be extracted by postfixing the expression that returns the list with `[`, a zero-based index and `]`. The `iscc` operations will be introduced gradually in this tutorial. On integer values, the following operators are available: `+` (addition), `-` (subtraction) and `*` (multiplication). On boolean values (`True` and `False`), the following operators are available: `+` (disjunction) and `*` (conjunction).

Example 1.2. *The following transcript contains two commands. The result of the first is assigned to a variable and therefore not printed. The result of the second is not assigned to a variable and therefore does get printed.*
`iscc` input (`assignment.iscc`):

```
A := 2 * (1 + 3);
A * A;
```

`iscc` invocation:

```
iscc < assignment.iscc
```

`iscc` output:

```
64
```

The `parse_file` operation is only available if support for `pet` had been built into `iscc`. This requires passing `--with-pet=bundled` to `configure`. Note that `iscc` turns on the autodetect option of `pet` by default. Use the `--no-pet-autodetect` option of `iscc` to turn it off again.

1.2.3 Python interface

Each of `isl`, `pet` and `barvinok` comes with a `python` interface, where the `pet` version is essentially identical to the `isl` version and where the `barvinok` version is an extended version of the `pet` version. In particular, the `barvinok` version also contains an interface to functions defined by the `barvinok` library. All three are called `isl.py`. They are included in the respective distributions, Note 1.9 but they can only be (re)built if `configure` has been explicitly told where to find `clang` using the `--with-clang` and/or `--with-clang-prefix` option. To actually build the interface, use `(cd interface; make isl.py)` for the isl version and make isl.py for the pet and the barvinok versions. The python interface is not installed automatically, so you typically need to adjust your PYTHONPATH to point to the interface subdirectory of the build tree of isl or to the build tree of pet or barvinok. You may also need to adjust LD_LIBRARY_PATH to point to the directory where libisl.so, libpet.so or libbarvinok.so is installed. Furthermore, you need to configure barvinok with the --enable-shared-barvinok option in order for libbarvinok.so to get built first. A simple illustration of how to use the python interface is shown in Example 2.7 on page 13.

The names of the classes in the python interface are derived from their isl counterparts by dropping the isl_ prefix. Since the classes are included in an isl module, this means that in practice, the first underscore needs to be replaced by a period. The names of the methods are derived from the corresponding isl function names by dropping the type name prefix. Some function names in isl have a suffix that refers to the type of the final argument. These suffixes are also dropped from the method names.

The pet distribution also contains a Python interface called pet.py. It is based on top of isl.py and requires the version from pet or from barvinok. The autodetect options is not turned on by default by the python interface to pet.

Acknowledgements

The author would like to thank Siddharth Bhat, Bazyli Klockiewicz and Jie Zhao for their feedback.

Notes

1.1. Earlier attempts to describe the concepts and operations involved in polyhedral compilation (within a specific context) are available from, e.g., Clauss, Garbervetsky, et al. (2011) and Verdoolaege (2013). Bastoul (2012, Chapter 2) also provides an overview, but from a different, more matrix oriented, perspective.

1.2. isl was first introduced by Verdoolaege, Janssens, et al. (2009) and described in more detail by Verdoolaege (2010). PPCG was introduced by Verdoolaege, Juega, et al. (2013).

1.3. The term "polyhedral compilation" appears to have originated from one particular group, first being used by Girbal et al. (2006), Pop et al. (2006), and Vasilache et al. (2006). It has later also been used in the tagline for the series of IMPACT workshops: International Workshop on Polyhedral Compilation Techniques. While the term may have originally been intended to have a more restrictive meaning, the scope of the workshop more or less corresponds to the meaning described in the text. Some authors even consider some forms of abstract interpretation and array region analysis to be part of polyhedral compilation.

1.4. Traditionally, there have been two approaches in "polyhedral compilation" research, even though this terminology was not used back then. One approach was firmly based on polyhedra and was using tools such as `PIP`, developed by Feautrier (1988b), and `PolyLib`, originally developed by Wilde (1993) and further extended by Loechner and Wilde (1997), in their implementations. The other approach was based on Presburger formulas and would use the `Omega` library, developed by Kelly, Maslov, et al. (1996) on top of the Omega test of Pugh (1992) in their implementations. One crucial feature that is shared by the two approaches is an instance-wise dataflow analysis (Feautrier 1991; Pugh and Wonnacott 1992).
1.5. The use of polyhedra as an abstract domain in abstract interpretation was introduced by Cousot and Halbwachs (1978). Array region analysis (using polyhedra) was introduced by Triolet et al. (1986) and further extended by Creusillet and Irigoin (1996).
1.6. The `pet` library was introduced by Verdoolaege and Grosser (2012).
1.7. These pragmas are inherited from `clan`, developed by Bastoul (2008).
1.8. The `iscc` tool was introduced by Verdoolaege (2011).
1.9. The `isl.py` interface should not be confused with the `islpy` library, which was introduced by Klöckner (2014) and is available from `http://documen.tician.de/islpy/`. Although both are wrappers around the `isl` library, the naming conventions and the set of exposed functionality are slightly different.

Chapter 2

Sets of Named Integer Tuples

This chapter describes the abstract elements that are used to represent various entities later on in the tutorial, sets and binary relations of such elements and operations on these sets and relations. In order to make this chapter accessible to readers who are not interested in polyhedral compilation, these concepts are treated purely abstractly. Furthermore, all sets in this chapter will be described extensionally. Intensional descriptions form the subject of Chapter 3 Presburger Sets and Relations.

2.1 Named Integer Tuples

The objects considered in this tutorial are each represented by a *named integer tuple*, consisting of an identifier (name) and a sequence of integer values. The identifier may be omitted and the sequence of integers may have a zero length. Two such named integer tuples are considered to be the same if they have the same identifier and the same sequence of integer values. Note 2.1

Notation 2.1 (Named Integer Tuple)**.** *The notation for a named integer tuple is formed by the identifier followed by a comma delimited list of the integer values enclosed in square brackets.*

Note 2.2

For example, the "named" integer tuple without identifier and with a zero-length sequence of integers is written "[]". The named integer tuple with identifier A and sequence of integers 2, 8 and 1 is written "A$[2, 8, 1]$". Named integer tuples will be extended to structured named integer tuples in Definition 2.66 on page 28.

Alternative 2.2 (Unnamed Integer Tuples)**.** *Some frameworks only deal with sequences of integers, without support for an explicit identifier. In cases where there are different types of such sequences at a conceptual level, the different types are usually encoded by means of an additional integer value that is added at the beginning or the end of the sequence.*

For example, if A is assigned the value 0, then the above named tuple could be represented as $[0, 2, 8, 1]$.

2.2 Sets

A set of named integer tuples contains zero or more named integer tuples as elements.

Notation 2.3 (Set)**.** *The notation for a* set *is formed by a semicolon delimited list of elements enclosed in braces.*

No order is defined on the elements in a set. This means in particular, that the elements in a set may be printed in a different order than the one in which it was defined. Elements in a set do not carry any multiplicity. That is, an element either belongs to a set or it does not belong to a set, but it cannot belong to the set multiple times. For example, the set

$$\{\, [];\mathrm{A}[2,8,1] \,\} \tag{2.1}$$

is equal to the set

$$\{\, \mathrm{A}[2,8,1]; []; [] \,\}. \tag{2.2}$$

Note 2.3 In `isl`, such sets are represented by an `isl_union_set`. The empty set is written $\{\,\}$ or $\emptyset$ in the text and `{ }` in `iscc`. In `isl`, an empty set can be created using `isl_union_set_empty`.

Alternative 2.4 (Fixed-dimensional Sets)**.** *Some frameworks do not allow integer tuples of different sizes to be combined into the same set. The tuples of smaller sizes are then typically padded with arbitrary integer values (say, zero). For example, if* $\mathrm{A}[2,8,1]$ *is encoded as* $[0,2,8,1]$ *as in Alternative 2.2 Unnamed Integer Tuples and if this element needs to be combined with* $\mathrm{B}[5]$ *in the same set, then the latter can be encoded as* $[1,5,0,0]$*, assuming that B is represented by the value* 1*.*

2.2.1 Basic Operations

The most basic operations are the intersection, the union and the set difference.

Operation 2.5 (Intersection of Sets)**.** *The intersection $A \cap B$ of two sets A and B contains the elements that are contained in both A and B.*

In `isl`, this operation is called `isl_union_set_intersect`. In `iscc`, this operation is written `*`.

Example 2.6. *`iscc` input (`intersection.iscc`):*

```
{ B[0]; A[2,8,1] } * { A[2,8,1]; C[5] };
```

`iscc` invocation:

```
iscc < intersection.iscc
```

iscc output:

```
{ A[2, 8, 1] }
```

Example 2.7. *The same result as in Example 2.6 can be obtained using the python interface as follows.*
python input (intersection.py):

```
import isl

s1 = isl.union_set("{ B[0]; A[2,8,1] }")
s2 = isl.union_set("{ A[2,8,1]; C[5] }")
print(s1.intersect(s2))
```

python invocation:

```
python < intersection.py
```

python output:

```
{ A[2, 8, 1] }
```

Operation 2.8 (Union of Sets)**.** *The union $A \cup B$ of two sets A and B contains the elements that are contained in either A or B.*

In `isl`, this operation is called `isl_union_set_union`. In `iscc`, this operation is written `+`.

Example 2.9. *iscc input (union.iscc):*

```
{ B[0]; A[2,8,1] } + { A[2,8,1]; C[5] };
```

iscc invocation:

```
iscc < union.iscc
```

iscc output:

```
{ A[2, 8, 1]; C[5]; B[0] }
```

Note that since no order is defined on the elements in a set, the elements in this union may be printed in a different order on your screen.

Operation 2.10 (Set Difference)**.** *The difference $A \setminus B$ of two sets A and B contains the elements that are contained in A but not in B.*

In `isl`, this operation is called `isl_union_set_subtract`. In `iscc`, this operation is written `-`.

Example 2.11. *iscc input (difference.iscc):*

```
{ B[0]; A[2,8,1] } - { A[2,8,1]; C[5] };
```

`iscc` *invocation:*

```
iscc < difference.iscc
```

`iscc` *output:*

```
{ B[0] }
```

2.2.2 Comparisons

The most basic comparison operation on sets is the equality operation.

Operation 2.12 (Equality of Sets)**.** *Two sets A and B are equal ($A = B$) if they contain the same elements.*

In `isl`, this operation is called `isl_union_set_is_equal`. In `iscc`, this operation is written `=`. See also Operation 3.26 on page 48.

Example 2.13. *The following transcript confirms that the set in* (2.1) *is equal to the set in* (2.2)*.*
`iscc` *input* (`set_equal.iscc`)*:*

```
{ []; A[2,8,1] } = { A[2,8,1]; []; [] };
```

`iscc` *invocation:*

```
iscc < set_equal.iscc
```

`iscc` *output:*

```
True
```

Example 2.14. *The same result as in Example 2.13 can be obtained using the python interface as follows.*
`python` *input* (`set_equal.py`)*:*

```
import isl

s1 = isl.union_set("{ []; A[2,8,1] }")
s2 = isl.union_set("{ A[2,8,1]; []; [] }")
print(s1.is_equal(s2))
```

`python` *invocation:*

```
python < set_equal.py
```

`python` *output:*

```
True
```

A special case of the equality operation is the test whether a set is empty.

Operation 2.15 (Emptiness of a Set)**.** *A set is empty if it does not contain any element, i.e., if it is equal to the empty set.*

In `isl`, this operation is called `isl_union_set_is_empty`. In `iscc`, this operation can be performed by comparing with the empty set. See also Operation 3.31 on page 49.

Example 2.16. *iscc input (set_empty.iscc):*

```
{ []; A[2,8,1] } = { };
```

iscc invocation:

```
iscc < set_empty.iscc
```

iscc output:

```
False
```

Other comparisons are the (strict) subset test and the (strict) superset test.

Operation 2.17 (Subset)**.** *The set A is a subset of the set B, $A \subseteq B$, if all elements of A are contained in B, i.e., if $A \setminus B = \emptyset$.*

In `isl`, this operation is called `isl_union_set_is_subset`. In `iscc`, this operation is written `<=`. See also Operation 3.34 on page 49.

Example 2.18. *iscc input (set_subset.iscc):*

```
{ []; A[2,8,1] } <= { A[2,8,1]; []; [] };
```

iscc invocation:

```
iscc < set_subset.iscc
```

iscc output:

```
True
```

Operation 2.19 (Strict Subset)**.** *The set A is a strict (or proper) subset of the set B, $A \subsetneq B$, if all elements of A are contained in B and B contains elements not in A, i.e., if $A \setminus B = \emptyset$ and $A \neq B$.*

In `isl`, this operation is called `isl_union_set_is_strict_subset`. In `iscc`, this operation is written `<`. See also Operation 3.37 on page 50.

Example 2.20. *iscc input (set_strict_subset.iscc):*

```
{ []; A[2,8,1] } < { A[2,8,1]; []; [] };
```

iscc invocation:

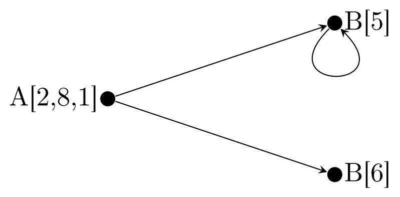


Figure 2.1: A graph representation of the binary relation in (2.3)

```
iscc < set_strict_subset.iscc
```

iscc output:

```
False
```

Operation 2.21 (Superset)**.** *The set A is a superset of the set B, $A \supseteq B$, if all elements of B are contained in A, i.e., if $B \subseteq A$.*

In `isl`, there is no specific function for this operation, but the function `isl_union_set_is_subset` can be called with the arguments reversed. In `iscc`, this operation is written `>=`. See also Operation 3.40 on page 50.

Operation 2.22 (Strict Superset)**.** *The set A is a strict (or proper) superset of the set B, $A \supsetneq B$, if all elements of B are contained in A and A contains elements not in B, i.e., if $B \subsetneq A$.*

In `isl`, there is no specific function for this operation, but the function `isl_union_set_is_strict_subset` can be called with the arguments reversed. In `iscc`, this operation is written `>`. See also Operation 3.42 on page 50.

2.3 Binary Relations

A *binary relation* is a set that contains pairs of named integer tuples.

Note 2.4 **Notation 2.23** (Pair of Elements)**.** *In `isl`, the two named integer tuples in each pair in a binary relation are separated by a `->`.*

Note that this notation does *not* imply that there would be a functional dependence from the first tuple to the second. That is, a given first tuple may appear in multiple pairs with different values for the second tuple. In fact, a binary relation can be considered as representing the edges of a graph. This graph may have loops, but no parallel edges.

Example 2.24. *Figure 2.1 shows a graph representation of the following binary relation:*

$$\{ \mathrm{A}[2,8,1] \to \mathrm{B}[5]; \mathrm{A}[2,8,1] \to \mathrm{B}[6]; \mathrm{B}[5] \to \mathrm{B}[5] \}. \tag{2.3}$$

In `isl`, such binary relations are represented by an `isl_union_map`. The empty binary relation is written $\{\,\}$ or $\emptyset$ in the text and `{ }` in `iscc`. In `isl`, an empty binary relation can be created using `isl_union_map_empty`. Note that even though a binary relation is essentially just a set of pairs of tuples, there is a strict separation in `isl` between sets and binary relations. That is, a set can only contain tuples (and no pairs of tuples) and a relation can only contain pairs of tuples (and no tuples themselves).

Alternative 2.25 (Encoding Binary Relations)**.** *Some frameworks do not have special support for binary relations. Binary relations then need to be encoded in sets of a dimension that is double the dimension of the base sets. For example, using the same encodings of* A *and* B *as in Alternative 2.4 Fixed-dimensional Sets, the binary relation in* (2.3) *can be encoded as*

$$\{\, [0,2,8,1,1,5,0,0]; [0,2,8,1,1,6,0,0]; [1,5,0,0,1,5,0,0] \,\}. \tag{2.4}$$

2.3.1 Basic Operations

Since a binary relation is essentially a set of pairs of tuples, the operations that apply to sets also apply to binary relations. Binary relations additionally admit an inverse and a composition operation.

Operation 2.26 (Intersection of Binary Relations)**.** *The intersection $A \cap B$ of two binary relations A and B contains the pairs of elements that are contained in both A and B.*

In `isl`, this operation is called `isl_union_map_intersect`. In `iscc`, this operation is written `*`.

Example 2.27. *`iscc` input (map_intersection.iscc):*

```
{ A[2,8,1] -> B[5]; B[5] -> B[5] } *
    { A[2,8,1] -> B[6]; B[5] -> B[5] };
```

`iscc` invocation:

```
iscc < map_intersection.iscc
```

`iscc` output:

```
{ B[5] -> B[5] }
```

Operation 2.28 (Union of Binary Relations)**.** *The union $A \cup B$ of two binary relations A and B contains the pairs of elements that are contained in either A or B.*

In `isl`, this operation is called `isl_union_map_union`. In `iscc`, this operation is written `+`.

Operation 2.29 (Binary Relation Difference)**.** *The difference $A \setminus B$ of two binary relations A and B contains the pairs of elements that are contained in A but not in B.*

In `isl`, this operation is called `isl_union_map_subtract`. In `iscc`, this operation is written `-`.

Operation 2.30 (Inverse of a Binary Relation)**.** *The inverse R^{-1} of a binary relation R contains the same pairs of elements as R, but with the order of the elements in the pair interchanged. That is,*

$$R^{-1} = \{\, \boldsymbol{j} \to \boldsymbol{i} : \boldsymbol{i} \to \boldsymbol{j} \in R \,\}. \tag{2.5}$$

In `isl`, this operation is called `isl_union_map_reverse`. In `iscc`, this operation is written `^-1`.

Example 2.31. *`iscc` input (`inverse.iscc`):*

```
{ A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] }^-1;
```

`iscc` invocation:

```
iscc < inverse.iscc
```

`iscc` output:

```
{ B[6] -> A[2, 8, 1]; B[5] -> A[2, 8, 1]; B[5] -> B[5] }
```

Operation 2.32 (Composition of Binary Relations)**.** *The composition $B \circ A$ of two binary relations A and B contains those pairs of elements such that the first element appears as a first element in A, the second element appears as a second element in B and such that the other element in those pairs (i.e., the second element in A and the first element in B) is the same. That is,*

$$B \circ A = \{\, \boldsymbol{i} \to \boldsymbol{j} : \exists \boldsymbol{k} : \boldsymbol{i} \to \boldsymbol{k} \in A \wedge \boldsymbol{k} \to \boldsymbol{j} \in B \,\}. \tag{2.6}$$

In `isl`, this operation is called `isl_union_map_apply_range`, with arguments A and B. In `iscc`, this operation is written `before` or "`.`" with arguments A and B; or, with the arguments reversed, `after` or `()`, in particular `B(A)`.

Example 2.33. *Consider the relations*

$$A = \{\, \mathrm{B}[6] \to \mathrm{A}[2, 8, 1]; \mathrm{B}[6] \to \mathrm{B}[5] \,\} \tag{2.7}$$

and

$$B = \{\, \mathrm{A}[2, 8, 1] \to \mathrm{B}[5]; \mathrm{A}[2, 8, 1] \to \mathrm{B}[6]; \mathrm{B}[5] \to \mathrm{B}[5] \,\}. \tag{2.8}$$

Their composition is

$$B \circ A = \{\, \mathrm{B}[6] \to \mathrm{B}[6]; \mathrm{B}[6] \to \mathrm{B}[5] \,\}. \tag{2.9}$$

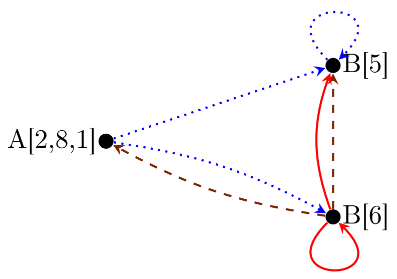


Figure 2.2: A graph representation of the binary relations in (2.7) (dashed lines) and (2.8) (dotted lines) along with their composition in (2.9) (solid lines)

Note that the composition of B and A (in the opposite order) is different:

$$A \circ B = \{ \mathrm{A}[2,8,1] \to \mathrm{B}[5]; \mathrm{A}[2,8,1] \to \mathrm{A}[2,8,1] \}. \tag{2.10}$$

iscc input (composition.iscc):

```
A := { B[6] -> A[2,8,1]; B[6] -> B[5] };
B := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
B after A;
A after B;
```

iscc invocation:

```
iscc < composition.iscc
```

iscc output:

```
{ B[6] -> B[6]; B[6] -> B[5] }
{ A[2, 8, 1] -> A[2, 8, 1]; A[2, 8, 1] -> B[5] }
```

The composition $R(R)$ of a binary relation R with itself can also be written R^2. Similarly, R^{-2} is the composition of the inverse of R with itself. More generally, the fixed power of a binary relation is defined as follows.

Operation 2.34 (Fixed Power of a Binary Relation)**.** *The fixed power R^n of a binary relation R with n a non-zero integer is equal to the composition of n copies of R if n is positive or the composition of $-n$ copies of R^{-1} if n is negative. In other words,* Note 2.5

$$R^n = \begin{cases} R & \text{if } n = 1 \\ R^{n-1}(R) & \text{if } n > 1 \\ R^{-1} & \text{if } n = -1 \\ R^{n+1}(R^{-1}) & \text{if } n < -1. \end{cases} \tag{2.11}$$

In `isl`, this operation is called `isl_union_map_fixed_power_val`. In `iscc`, this operation is written `^`.

Example 2.35. *`iscc` input (`fixed_power.iscc`):*

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
R;
R^2;
R^3;
R^-1;
R^-2;
```

`iscc` invocation:

```
iscc < fixed_power.iscc
```

`iscc` output:

```
{ B[5] -> B[5]; A[2, 8, 1] -> B[6]; A[2, 8, 1] -> B[5] }
{ A[2, 8, 1] -> B[5]; B[5] -> B[5] }
{ A[2, 8, 1] -> B[5]; B[5] -> B[5] }
{ B[6] -> A[2, 8, 1]; B[5] -> A[2, 8, 1]; B[5] -> B[5] }
{ B[5] -> A[2, 8, 1]; B[5] -> B[5] }
```

Example 2.36. *The same result as in Example 2.35 can be obtained using the python interface as follows.*
`python` input (`fixed_power.py`):

```
import isl

r = isl.union_map(
    "{ A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] }")
print(r)
print(r.fixed_power(isl.val(2)))
print(r.fixed_power(isl.val(3)))
print(r.fixed_power(isl.val(-1)))
print(r.fixed_power(isl.val(-2)))
```

`python` invocation:

```
python < fixed_power.py
```

`python` output:

```
{ B[5] -> B[5]; A[2, 8, 1] -> B[6]; A[2, 8, 1] -> B[5] }
{ A[2, 8, 1] -> B[5]; B[5] -> B[5] }
{ A[2, 8, 1] -> B[5]; B[5] -> B[5] }
{ B[6] -> A[2, 8, 1]; B[5] -> A[2, 8, 1]; B[5] -> B[5] }
{ B[5] -> A[2, 8, 1]; B[5] -> B[5] }
```

2.3.2 Comparisons

The same comparison operators that can be applied to sets can also be applied to binary relations.

Operation 2.37 (Equality of Binary Relations)**.** *Two binary relations A and B are equal ($A = B$) if they contain the same pairs of elements.*

In `isl`, this operation is called `isl_union_map_is_equal`. In `iscc`, this operation is written `=`. See also Operation 3.30 on page 49.

Operation 2.38 (Emptiness of a Binary Relation)**.** *A binary relation is empty if it does not contain any pair of elements, i.e., if it is equal to the empty binary relation.*

In `isl`, this operation is called `isl_union_map_is_empty`. In `iscc`, this operation can be performed by comparing with the empty binary relation. See also Operation 3.33 on page 49.

Operation 2.39 (Subrelation)**.** *The binary relation A is a subset of the binary relation B, $A \subseteq B$, if all pairs of elements in A are contained in B, i.e., if $A \setminus B = \emptyset$.*

In `isl`, this operation is called `isl_union_map_is_subset`. In `iscc`, this operation is written `<=`. See also Operation 3.36 on page 50.

Operation 2.40 (Strict Subrelation)**.** *The binary relation A is a strict (or proper) subset of the binary relation B, $A \subsetneq B$, if all pairs of elements of A are contained in B and B contains pairs of elements not in A, i.e., if $A \setminus B = \emptyset$ and $A \neq B$.*

In `isl`, this operation is called `isl_union_map_is_strict_subset`. In `iscc`, this operation is written `<`. See also Operation 3.39 on page 50.

Operation 2.41 (Superrelation)**.** *The binary relation A is a superset of the binary relation B, $A \supseteq B$, if all pairs of elements in B are contained in A, i.e., if $B \subseteq A$.*

In `isl`, there is no specific function for this operation, but the function `isl_union_map_is_subset` can be called with the arguments reversed. In `iscc`, this operation is written `>=`. See also Operation 3.41 on page 50.

Operation 2.42 (Strict Superrelation)**.** *The binary relation A is a strict (or proper) superset of the binary relation B, $A \supsetneq B$, if all pairs of elements in B are contained in A and A contains pairs of elements not in B, i.e., if $B \subsetneq A$.*

In `isl`, there is no specific function for this operation, but the function `isl_union_map_is_strict_subset` can be called with the arguments reversed. In `iscc`, this operation is written `>`. See also Operation 3.43 on page 50.

2.3.3 Conversions

This section describes operations that create binary relations from sets or the other way around.

Operation 2.43 (Domain of a Binary Relation)**.** *The domain* $\operatorname{dom} R$ *of a binary relation* R *consists of the elements that appear as first element in the pairs of elements of* R*. That is,*

$$\operatorname{dom} R = \{\, \boldsymbol{i} : \exists \boldsymbol{j} : \boldsymbol{i} \to \boldsymbol{j} \in R \,\}. \tag{2.12}$$

In `isl`, this operation is called `isl_union_map_domain`. In `iscc`, this operation is written `dom` or `domain`.

Example 2.44. *Consider the binary relation from Example 2.24 on page 16, shown in Figure 2.1 on page 16. The domain of this relation is*

$$\{\, \mathrm{A}[2,8,1]; \mathrm{B}[5] \,\}. \tag{2.13}$$

iscc input (domain.iscc):

```
dom { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
```

iscc invocation:

```
iscc < domain.iscc
```

iscc output:

```
{ B[5]; A[2, 8, 1] }
```

Operation 2.45 (Range of a Binary Relation)**.** *The range* $\operatorname{ran} R$ *of a binary relation* R *consists of the elements that appear as second element in the pairs of elements of* R*. That is,*

$$\operatorname{ran} R = \{\, \boldsymbol{j} : \exists \boldsymbol{i} : \boldsymbol{i} \to \boldsymbol{j} \in R \,\}. \tag{2.14}$$

In `isl`, this operation is called `isl_union_map_range`. In `iscc`, this operation is written `ran` or `range`.

Example 2.46. *Consider once more the binary relation from Example 2.24 on page 16, shown in Figure 2.1 on page 16. The range of this relation is*

$$\{\, \mathrm{B}[5]; \mathrm{B}[6] \,\}. \tag{2.15}$$

iscc input (range.iscc):

```
ran { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
```

iscc invocation:

```
iscc < range.iscc
```

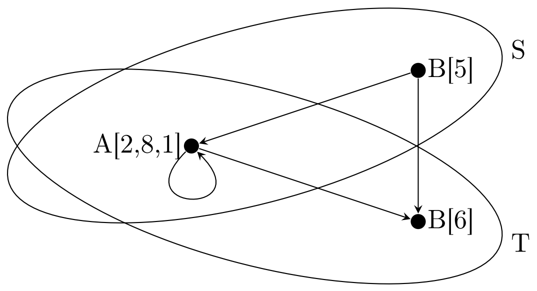


Figure 2.3: The two sets in (2.17) and their universal relation

iscc output:

```
{ B[6]; B[5] }
```

Operation 2.47 (Universal Binary Relation between Sets)**.** *The universal relation $A \to B$ between two sets A and B is the binary relation that contains the pairs of elements obtained by taking the first element from A and the second element from B. That is,*

$$A \to B = \{\, \boldsymbol{i} \to \boldsymbol{j} : \boldsymbol{i} \in A \wedge \boldsymbol{j} \in B \,\}. \tag{2.16}$$

In `isl`, this operation is called `isl_union_map_from_domain_and_range`. In `iscc`, this operation is written `->`.

Example 2.48. *The two sets*

$$S = \{\, \mathrm{A}[2,8,1]; \mathrm{B}[5] \,\} \quad \textit{and} \quad T = \{\, \mathrm{A}[2,8,1]; \mathrm{B}[6] \,\} \tag{2.17}$$

along with the universal relation constructed from the two sets are shown in Figure 2.3.
iscc input (universal.iscc):

```
S := { A[2,8,1]; B[5] };
T := { A[2,8,1]; B[6] };
S -> T;
```

iscc invocation:

```
iscc < universal.iscc
```

iscc output:

```
{ A[2, 8, 1] -> A[2, 8, 1]; A[2, 8, 1] -> B[6]; B[5] -> A[2,
    ↪ 8, 1]; B[5] -> B[6] }
```

Operation 2.49 (Identity Relation on a Set)**.** *The* identity relation 1_S *on a set S is the binary relation that contains a pair of elements for each element of S consisting of two copies of that element. That is,*

$$1_S = \{\, \boldsymbol{i} \to \boldsymbol{i} : \boldsymbol{i} \in S \,\}. \tag{2.18}$$

In `isl`, this operation is called `isl_union_set_identity`.

Example 2.50. *`python` input (`identity.py`):*

```
import isl

s = isl.union_set("{ B[6]; A[2,8,1]; B[5] }")
print(s.identity())
```

`python` invocation:

```
python < identity.py
```

`python` output:

```
{ A[2, 8, 1] -> A[2, 8, 1]; B[6] -> B[6]; B[5] -> B[5] }
```

2.3.4 Mixed Operations

This section describes some operations that combine a binary relation and a set.

Operation 2.51 (Domain Restriction)**.** *The domain restriction $R \cap_{\mathrm{dom}} S$ of a binary relation R with respect to a set S contains those pairs of elements of R of which the first in the pair belongs to S. In other words,*

$$R \cap_{\mathrm{dom}} S = R \cap (S \to (\operatorname{ran} R)) \tag{2.19}$$

In `isl`, this operation is called `isl_union_map_intersect_domain`. In `iscc`, this operation is written `*`.

Example 2.52. *`iscc` input (`intersect_domain.iscc`):*

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
S := { A[2,8,1]; C[5] };
R * S;
```

`iscc` invocation:

```
iscc < intersect_domain.iscc
```

`iscc` output:

```
{ A[2, 8, 1] -> B[6]; A[2, 8, 1] -> B[5] }
```

Operation 2.53 (Range Restriction)**.** *The range restriction $R \cap_{\text{ran}} S$ of a binary relation R with respect to a set S contains those pairs of elements of R of which the second in the pair belongs to S. In other words,*

$$R \cap_{\text{ran}} S = R \cap ((\operatorname{dom} R) \to S) \tag{2.20}$$

In `isl`, this operation is called `isl_union_map_intersect_range`. In `iscc`, this operation is written `->*`.

Example 2.54. *`iscc` input (`intersect_range.iscc`):*

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
S := { A[2,8,1]; B[5] };
R ->* S;
```

`iscc` invocation:

```
iscc < intersect_range.iscc
```

`iscc` output:

```
{ A[2, 8, 1] -> B[5]; B[5] -> B[5] }
```

Operation 2.55 (Domain Subtraction)**.** *The domain subtraction $R \setminus_{\text{dom}} S$ of a binary relation R with respect to a set S contains those pairs of elements of R of which the first in the pair does not belong to S. In other words,*

$$R \setminus_{\text{dom}} S = R \setminus (S \to (\operatorname{ran} R)) \tag{2.21}$$

In `isl`, this operation is called `isl_union_map_subtract_domain`. In `iscc`, this operation is written `-`.

Example 2.56. *`iscc` input (`subtract_domain.iscc`):*

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
S := { A[2,8,1]; C[5] };
R - S;
```

`iscc` invocation:

```
iscc < subtract_domain.iscc
```

`iscc` output:

```
{ B[5] -> B[5] }
```

Operation 2.57 (Range Subtraction)**.** *The range subtraction $R \setminus_{\text{ran}} S$ of a binary relation R with respect to a set S contains those pairs of elements of R of which the second in the pair does not belong to S. In other words,*

$$R \setminus_{\text{ran}} S = R \setminus ((\operatorname{dom} R) \to S) \tag{2.22}$$

In `isl`, this operation is called `isl_union_map_subtract_range`. In `iscc`, this operation is written `->-`.

Example 2.58. *`iscc` input (`subtract_range.iscc`):*

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
S := { A[2,8,1]; B[5] };
R ->- S;
```

`iscc` invocation:

```
iscc < subtract_range.iscc
```

`iscc` output:

```
{ A[2, 8, 1] -> B[6] }
```

Operation 2.59 (Application)**.** *The application $R(S)$ of a binary relation R to a set S is the set containing those elements that appear as the second element in a pair of elements in R while the corresponding first element is an element of S. In other words,*

$$\begin{aligned} R(S) &= \operatorname{ran}(R \cap_{\text{dom}} S) \\ &= \{\, \boldsymbol{j} : \exists \boldsymbol{i} \in S : \boldsymbol{i} \to \boldsymbol{j} \in R \,\}. \end{aligned} \tag{2.23}$$

In `isl`, this operation is called `isl_union_set_apply` with arguments S and R. In `iscc`, this operation is written `()`, in particular `R(S)`.

Example 2.60. *`iscc` input (`application.iscc`):*

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
S := { A[2,8,1]; B[5] };
R(S);
```

`iscc` invocation:

```
iscc < application.iscc
```

`iscc` output:

```
{ B[6]; B[5] }
```

2.3.5 Properties

While a binary relation does not necessarily represent a (single-valued) function, some binary relations may very well be single-valued. The following operation can be used to check for this property.

Operation 2.61 (Single-valued)**.** *A binary relation R is* single-valued, *i.e., a* function, *if every element that appears as the first element in a pair of elements in R only appears as the first element in one such pair.*

In `isl`, this operation is called `isl_union_map_is_single_valued`. The property can be evaluated by composing the inverse of R with R and checking whether the result is a subset of the identity relation on the range of R, i.e.

$$R \circ R^{-1} \subseteq 1_{\operatorname{ran} R}. \tag{2.24}$$

See also Operation 3.44 on page 51.

Example 2.62. *The relation*

$$\{ \mathrm{B}[6] \to \mathrm{A}[2,8,1]; \mathrm{B}[6] \to \mathrm{B}[5] \} \tag{2.25}$$

is not single-valued, but the relation

$$\{ \mathrm{A}[2,8,1] \to \mathrm{B}[5]; \mathrm{B}[5] \to \mathrm{B}[5] \} \tag{2.26}$$

is single-valued.
python input (singlevalued.py):

```
import isl

r1 = isl.union_map("{ B[6] -> A[2,8,1]; B[6] -> B[5] }")
r2 = isl.union_map("{ A[2,8,1] -> B[5]; B[5] -> B[5] }")
print(r1.is_single_valued())
print(r2.is_single_valued())
```

python invocation:

```
python < singlevalued.py
```

python output:

```
False
True
```

Similarly, a relation can be checked for being injective or bijective.

Operation 2.63 (Injective)**.** *A binary relation R is* injective *if every element that appears as the second element in a pair of elements in R only appears as the second element in one such pair, i.e., if its inverse is single-valued.*

In `isl`, this operation is called `isl_union_map_is_injective`. The property can be evaluated by composing R with its inverse and checking whether the result is a subset of the identity relation on the domain of R, i.e.

$$R^{-1} \circ R \subseteq 1_{\operatorname{dom} R}. \tag{2.27}$$

See also Operation 3.45 on page 51.

Example 2.64. *python input (injective.py):*

```
import isl

r1 = isl.union_map("{ B[6] -> A[2,8,1]; B[6] -> B[5] }")
r2 = isl.union_map("{ A[2,8,1] -> B[5]; B[5] -> B[5] }")
print(r1.is_injective())
print(r2.is_injective())
```

`python` *invocation:*

```
python < injective.py
```

`python` *output:*

```
True
False
```

Operation 2.65 (Bijective)**.** *A binary relation R is* bijective *if it is both single-valued and injective.*

2.4 Wrapped Relations

While sets keep track of named integer tuples and binary relations keep track of pairs of such tuples, it can sometimes be convenient to keep track of relations between more than two such tuples. `isl` currently does not support ternary or general n-ary relations. However, it does allow a pair of tuples to be combined into a single tuple, which can then again appear as the first or second tuple in a pair of tuples. This process is called “wrapping”. The result of wrapping a pair of named integer tuples is called a structured named integer tuple.

2.4.1 Structured Named Integer Tuples

Definition 2.66 (Structured Named Integer Tuple)**.** *A* structured named integer tuple *is either*

- *a named integer tuple, i.e., an identifier n along with $d \geq 0$ integers i_j for $0 \leq j < d$, written $n[i_0, i_1, \ldots, i_{d-1}]$, or,*
- *a named pair of structured named integer tuples, i.e., an identifier n along with two structured named integer tuples $\boldsymbol{i}$ and $\boldsymbol{j}$ written $n[\boldsymbol{i} \to \boldsymbol{j}]$.*

Example 2.67. *The following is a set of structured named integer tuples:*

$$\{\, \mathrm{B}[5]; \mathrm{S}[\mathrm{B}[6] \to \mathrm{A}[2,8,1]]; \mathrm{Q}[\mathrm{B}[5] \to \mathrm{S}[\mathrm{B}[6] \to \mathrm{A}[2,8,1]]] \,\}. \tag{2.28}$$

The following is a binary relation of structured named integer tuples:

$$\{\, \mathrm{B}[5] \to \mathrm{A}[2,8,1]; \mathrm{S}[\mathrm{B}[6] \to \mathrm{A}[2,8,1]] \to \mathrm{B}[5] \,\}. \tag{2.29}$$

From this point on, sets will be considered to contain structured named integer tuples rather than just named integer tuples and binary relations will be considered to contain pairs of structured named integer tuples. In fact, the concept of *named integer tuple* itself will be considered to also include structured named integer tuples. However, as before, a set cannot contain a pair of tuples and a binary relation cannot contain a single tuple.

It will be convenient to consider groups of elements that share the same structure and/or identifiers together. To this end, the following concept is defined.

Definition 2.68 (Space)**.** *The* space $\mathcal{S}\boldsymbol{i}$ *of a structured named integer tuple* $\boldsymbol{i}$ *is*

- n/d*, if* $\boldsymbol{i}$ *is of the form* $n[i_0, i_1, \ldots, i_{d-1}]$*, with* n *an identifier and* d *a non-negative integer, or,*
- $(n, \mathcal{S}(\boldsymbol{j}), \mathcal{S}(\boldsymbol{k}))$*, if* $\boldsymbol{i}$ *is of the form* $n[\boldsymbol{j} \to \boldsymbol{k}]$*, with* n *an identifier and* $\boldsymbol{j}$ *and* $\boldsymbol{k}$ *structured named integer tuples.*

The space of a pair of structured named integer tuples $\boldsymbol{i} \to \boldsymbol{j}$ *is* $(\mathcal{S}(\boldsymbol{i}), \mathcal{S}(\boldsymbol{j}))$*.*

Example 2.69. *The space of the tuple* $\mathrm{Q}[\mathrm{B}[5] \to \mathrm{S}[\mathrm{B}[6] \to \mathrm{A}[2,8,1]]]$ *is* $(\mathrm{Q}, \mathrm{B}/1, (\mathrm{S}, \mathrm{B}/1, \mathrm{A}/3))$*.*

Similarly, the values of the integers can be extracted from a structured named integer tuple.

Definition 2.70 (Value Vector)**.** *The* value vector $\mathcal{V}\boldsymbol{i}$ *of a structured named integer tuple* $\boldsymbol{i}$ *is the vector*

- $(i_0, i_1, \ldots, i_{d-1})$*, if* $\boldsymbol{i}$ *is of the form* $n[i_0, i_1, \ldots, i_{d-1}]$*, with* n *an identifier and* d *a non-negative integer, or,*
- $\mathcal{V}(\boldsymbol{j}) \| \mathcal{V}(\boldsymbol{k})$*, with* $\|$ *the concatenation of two vectors, if* $\boldsymbol{i}$ *is of the form* $n[\boldsymbol{j} \to \boldsymbol{k}]$*, with* n *an identifier and* $\boldsymbol{j}$ *and* $\boldsymbol{k}$ *structured named integer tuples.*

  The value vector of a pair of structured named integer tuples $\boldsymbol{i} \to \boldsymbol{j}$ *is* $\mathcal{V}(\boldsymbol{i}) \| \mathcal{V}(\boldsymbol{j})$*.*

Example 2.71. *The value vector of the tuple* $\mathrm{Q}[\mathrm{B}[5] \to \mathrm{S}[\mathrm{B}[6] \to \mathrm{A}[2,8,1]]]$ *is* $(5,6,2,8,1)$*.*

2.4.2 Wrapping and Unwrapping

Operation 2.72 (Wrap)**.** *The* wrap $\mathcal{W}R$ *of a binary relation* R *is a set that contains an anonymous wrapped copy of each pair of elements in* R*. That is,*

$$\mathcal{W}R = \{\, [\boldsymbol{i} \to \boldsymbol{j}] : \boldsymbol{i} \to \boldsymbol{j} \in R \,\}. \tag{2.30}$$

In `isl`, this operation is called `isl_union_map_wrap`. In `iscc`, this operation is written `wrap`.

Example 2.73. *`iscc` input (`wrap.iscc`):*

```
wrap { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
```

`iscc` invocation:

```
iscc < wrap.iscc
```

`iscc` output:

```
{ [A[2, 8, 1] -> B[6]]; [A[2, 8, 1] -> B[5]]; [B[5] -> B[5]]
   ↪  }
```

Operation 2.74 (Unwrap)**.** *The* unwrap $\mathcal{W}^{-1}S$ *of a set S is a binary relation that contains the pairs of elements of which S contains a wrapped copy. That is,*

$$\mathcal{W}^{-1}S = \{\, \boldsymbol{i} \to \boldsymbol{j} : \exists n : n[\boldsymbol{i} \to \boldsymbol{j}] \in S \,\}. \tag{2.31}$$

In `isl`, this operation is called `isl_union_set_unwrap`. In `iscc`, this operation is written `unwrap`.

Example 2.75. *`iscc` input (`unwrap.iscc`):*

```
S := { B[5]; S[B[6] -> A[2, 8, 1]];
       Q[B[5] -> S[B[6] -> A[2, 8, 1]]] };
unwrap S;
```

`iscc` invocation:

```
iscc < unwrap.iscc
```

`iscc` output:

```
{ B[5] -> S[B[6] -> A[2, 8, 1]]; B[6] -> A[2, 8, 1] }
```

Note that while $\mathcal{W}^{-1}\mathcal{W}R$, with R a binary relation, is always equal to R, $\mathcal{W}\mathcal{W}^{-1}S$, with S a set, may not be equal to S. In particular, S may contain elements that are not wrapped pairs and those will be removed. Moreover, if any wrapped pair has an identifier, then that identifier will be removed from the wrapped pair as well.

Example 2.76. *`iscc` input (`wrap_unwrap.iscc`):*

```
S := { B[5]; S[B[6] -> A[2, 8, 1]];
       Q[B[5] -> S[B[6] -> A[2, 8, 1]]] };
wrap (unwrap S);
```

`iscc` invocation:

```
iscc < wrap_unwrap.iscc
```

iscc output:

```
{ [B[6] -> A[2, 8, 1]]; [B[5] -> S[B[6] -> A[2, 8, 1]]] }
```

2.4.3 Products

A product of two sets (or binary relations) is a set (or binary relation) that combines the tuples in its arguments into wrapped relations. In case of a binary relation, it is also possible to only combine the domains or the ranges.

Operation 2.77 (Set Product)**.** *The product $A \times B$ of two sets A and B is a set that contains the wrapped pairs of elements obtained by taking the first element from A and the second element from B. In other words the product of two sets is the wrap of the universal relation between the two sets. That is,*

$$\begin{aligned} A \times B &= \mathcal{W}(A \to B) \\ &= \{ [\boldsymbol{i} \to \boldsymbol{j}] : \boldsymbol{i} \in A \wedge \boldsymbol{j} \in B \}. \end{aligned} \tag{2.32}$$

In `isl`, this operation is called `isl_union_set_product`. In `iscc`, this operation is written `cross`.

Example 2.78. *Compare the following transcript to that of Example 2.48 on page 23.*
iscc input (product.iscc):

```
S := { A[2,8,1]; B[5] };
T := { A[2,8,1]; B[6] };
S cross T;
```

iscc invocation:

```
iscc < product.iscc
```

iscc output:

```
{ [A[2, 8, 1] -> B[6]]; [A[2, 8, 1] -> A[2, 8, 1]]; [B[5] ->
    ↪  B[6]]; [B[5] -> A[2, 8, 1]] }
```

Operation 2.79 (Binary Relation Product)**.** *The product $A \times B$ of two binary relations A and B is a binary relation that contains the pairs of wrapped pairs of elements obtained by taking the first elements in the wrapped pairs from A and the second elements from B. That is,*

$$A \times B = \{ [\boldsymbol{i} \to \boldsymbol{m}] \to [\boldsymbol{j} \to \boldsymbol{n}] : \boldsymbol{i} \to \boldsymbol{j} \in A \wedge \boldsymbol{m} \to \boldsymbol{n} \in B \}. \tag{2.33}$$

In `isl`, this operation is called `isl_union_map_product`. In `iscc`, this operation is written `cross`.

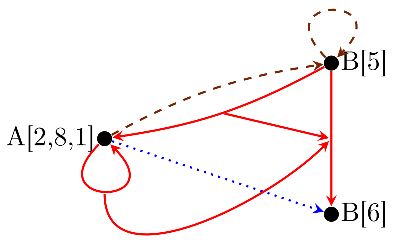


Figure 2.4: A graph representation of the binary relations in (2.34) (dashed lines) and (2.35) (dotted lines) along with their product in (2.36) (solid lines)

Example 2.80. *Consider the relations*

$$A = \{\, \mathrm{A}[2,8,1] \to \mathrm{B}[5]; \mathrm{B}[5] \to \mathrm{B}[5] \,\} \tag{2.34}$$

and

$$B = \{\, \mathrm{A}[2,8,1] \to \mathrm{B}[6] \,\}. \tag{2.35}$$

Their product is

$$\begin{aligned} A \times B = \{\, & [\mathrm{A}[2,8,1] \to \mathrm{A}[2,8,1]] \to [\mathrm{B}[5] \to \mathrm{B}[6]]; \\ & [\mathrm{B}[5] \to \mathrm{A}[2,8,1]] \to [\mathrm{B}[5] \to \mathrm{B}[6]] \,\}. \end{aligned} \tag{2.36}$$

These relations are illustrated in Figure 2.4.
iscc input (map_product.iscc):

```
A := { A[2,8,1] -> B[5]; B[5] -> B[5] };
B := { A[2,8,1] -> B[6] };
A cross B;
```

iscc invocation:

```
iscc < map_product.iscc
```

iscc output:

```
{ [A[2, 8, 1] -> A[2, 8, 1]] -> [B[5] -> B[6]]; [B[5] -> A
    ↪ [2, 8, 1]] -> [B[5] -> B[6]] }
```

Note that the product of two binary relations is different from the universal relation between the wraps of the binary relations.

Example 2.81. *iscc input (universe_wrap.iscc):*

```
A := { A[2,8,1] -> B[5]; B[5] -> B[5] };
B := { A[2,8,1] -> B[6] };
(wrap A) -> (wrap B);
```

$$\begin{tikzcd}
\boldsymbol{i} \arrow[r] \arrow[d, phantom] & \boldsymbol{j} & & \boldsymbol{i} \arrow[d] & \boldsymbol{j} \arrow[d] \\
\boldsymbol{m} \arrow[r] & \boldsymbol{n} & \Longrightarrow & \boldsymbol{m} & \boldsymbol{n}
\end{tikzcd}$$

Figure 2.5: Graphical representation of the zip operation

iscc invocation:

```
iscc < universe_wrap.iscc
```

iscc output:

```
{ [B[5] -> B[5]] -> [A[2, 8, 1] -> B[6]]; [A[2, 8, 1] -> B
    ↪ [5]] -> [A[2, 8, 1] -> B[6]] }
```

The following operation can be used to change the product of binary relations into the universal relations between the wraps of the binary relations and vice versa.

Operation 2.82 (Zip)**.** *The* zip *of a binary relation R consists of pairs of wrapped relations. The first contains the first elements in the wrapped relations in the first and second element of R. The second contains the second elements in the wrapped relations in the first and second element of R. That is,*

$$\operatorname{zip} R = \{\, [\boldsymbol{i} \to \boldsymbol{m}] \to [\boldsymbol{j} \to \boldsymbol{n}] : [\boldsymbol{i} \to \boldsymbol{j}] \to [\boldsymbol{m} \to \boldsymbol{n}] \in R \,\}. \tag{2.37}$$

In `isl`, this operation is called `isl_union_map_zip`. In `iscc`, this operation is written `zip`. The effect of this operation is shown graphically in Figure 2.5.

Example 2.83. *iscc input (zip.iscc):*

```
R := { [A[2, 8, 1] -> A[2, 8, 1]] -> [B[5] -> B[6]];
       [B[5] -> A[2, 8, 1]] -> [B[5] -> B[6]] };
zip R;
zip (zip R);
```

iscc invocation:

```
iscc < zip.iscc
```

iscc output:

```
{ [B[5] -> B[5]] -> [A[2, 8, 1] -> B[6]]; [A[2, 8, 1] -> B
    ↪ [5]] -> [A[2, 8, 1] -> B[6]] }
{ [A[2, 8, 1] -> A[2, 8, 1]] -> [B[5] -> B[6]]; [B[5] -> A
    ↪ [2, 8, 1]] -> [B[5] -> B[6]] }
```

Operation 2.84 (Domain Product)**.** *The domain product $A \rtimes B$ of two binary relations A and B is a binary relation that maps wrapped pairs of elements, the first being the first in a pair from A and the second being the first in a pair from B, to the element that appears as the second in both these pairs from A and B. That is,*

$$A \rtimes B = \{ [\boldsymbol{i} \to \boldsymbol{j}] \to \boldsymbol{k} : \boldsymbol{i} \to \boldsymbol{k} \in A \wedge \boldsymbol{j} \to \boldsymbol{k} \in B \}. \tag{2.38}$$

In `isl`, this operation is called `isl_union_map_domain_product`.

Example 2.85. *Consider once more the binary relations from Example 2.80 on page 32. The domain product of the binary relations in* (2.34) *and* (2.35) *is empty because the two relations do not have any range elements in common.*
`python` *input (`domain_product.py`):*

```
import isl

r1 = isl.union_map("{ A[2,8,1] -> B[5]; B[5] -> B[5] }")
r2 = isl.union_map("{ A[2,8,1] -> B[6] }")
print(r1.domain_product(r2))
```

`python` *invocation:*

```
python < domain_product.py
```

`python` *output:*

```
{  }
```

Operation 2.86 (Range Product)**.** *The range product $A \ltimes B$ of two binary relations A and B is a binary relation that maps elements that appear as a first element in both A and B to the wrapped pairs of corresponding second elements, the first from A and the second from B. That is,*

$$A \ltimes B = \{ \boldsymbol{i} \to [\boldsymbol{j} \to \boldsymbol{k}] : \boldsymbol{i} \to \boldsymbol{j} \in A \wedge \boldsymbol{i} \to \boldsymbol{k} \in B \}. \tag{2.39}$$

In `isl`, this operation is called `isl_union_map_range_product`.

Example 2.87. *Consider once more the binary relations from Example 2.80 on page 32. The range product of the binary relations in* (2.34) *and* (2.35) *is*

$$A \ltimes B = \{ \mathrm{A}[2,8,1] \to [\mathrm{B}[5] \to \mathrm{B}[6]] \}. \tag{2.40}$$

These relations are illustrated in Figure 2.6 on the facing page.
`python` *input (`range_product.py`):*

```
import isl

r1 = isl.union_map("{ A[2,8,1] -> B[5]; B[5] -> B[5] }")
r2 = isl.union_map("{ A[2,8,1] -> B[6] }")
print(r1.range_product(r2))
```

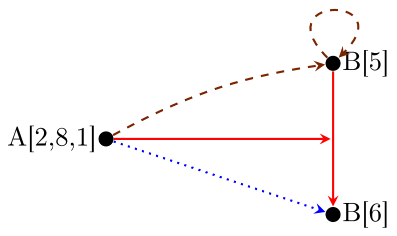


Figure 2.6: A graph representation of the binary relations in (2.34) (dashed lines) and (2.35) (dotted lines) along with their range product in (2.40) (solid lines)

python invocation:

```
python < range_product.py
```

python output:

```
{ A[2, 8, 1] -> [B[5] -> B[6]] }
```

The arguments that contributed to a product can be extracted again using the following functions. In the case of `isl_union_map_domain_product` and `isl_union_map_range_product`, only some pairs of elements in the inputs may have contributed to the product and therefore only these pairs can be extracted again.

Operation 2.88 (Domain Factor of Product)**.** *The domain factor of a binary relation R consists of those pairs of elements where the first appears as the first element in the wrapped relation in the first element of a pair of elements in R and the second appears as the first element in the wrapped relation in the second element of the same pair in R. That is, the domain factor of R is*

$$\{\, \boldsymbol{i} \to \boldsymbol{m} : \exists \boldsymbol{j}, \boldsymbol{n} : [\boldsymbol{i} \to \boldsymbol{j}] \to [\boldsymbol{m} \to \boldsymbol{n}] \in R \,\}. \tag{2.41}$$

In `isl`, this operation is called `isl_union_map_factor_domain`.

Example 2.89. *python input (factor_domain.py):*

```
import isl

r = isl.union_map(
    "{ [A[2, 8, 1] -> A[2, 8, 1]] -> [B[5] -> B[6]]; "
    "[B[5] -> A[2, 8, 1]] -> [B[5] -> B[6]] }")
print(r.factor_domain())
```

python invocation:

```
python < factor_domain.py
```

python output:

```
{ A[2, 8, 1] -> B[5]; B[5] -> B[5] }
```

Operation 2.90 (Range Factor of Product)**.** *The range factor of a binary relation R consists of those pairs of elements where the first appears as the second element in the wrapped relation in the first element of a pair of elements in R and the second appears as the second element in the wrapped relation in the second element of the same pair in R. That is, the range factor of R is*

$$\{\, \boldsymbol{j} \to \boldsymbol{n} : \exists \boldsymbol{i}, \boldsymbol{m} : [\boldsymbol{i} \to \boldsymbol{j}] \to [\boldsymbol{m} \to \boldsymbol{n}] \in R \,\}. \tag{2.42}$$

In `isl`, this operation is called `isl_union_map_factor_range`.

Example 2.91. *python input (factor_range.py):*

```
import isl

r = isl.union_map(
    "{ [A[2, 8, 1] -> A[2, 8, 1]] -> [B[5] -> B[6]]; "
    "[B[5] -> A[2, 8, 1]] -> [B[5] -> B[6]] }")
print(r.factor_range())
```

python invocation:

```
python < factor_range.py
```

python output:

```
{ A[2, 8, 1] -> B[6] }
```

Operation 2.92 (Domain Factor of Domain Product)**.** *The domain factor of a binary relation R considered as a domain product consists of those pairs of elements where the first appears as the first element in the wrapped relation in the first element of a pair of elements in R and the second is the second element of that pair in R. That is, the domain factor of R as a domain product is*

$$\{\, \boldsymbol{i} \to \boldsymbol{k} : \exists \boldsymbol{j} : [\boldsymbol{i} \to \boldsymbol{j}] \to \boldsymbol{k} \in R \,\}. \tag{2.43}$$

In `isl`, this operation is called `isl_union_map_domain_factor_domain`.

Example 2.93. *python input (domain_factor_domain.py):*

```
import isl

r = isl.union_map(
    "{ [A[2, 8, 1] -> A[2, 8, 1]] -> [B[5] -> B[6]]; "
    "[B[5] -> A[2, 8, 1]] -> [B[5] -> B[6]] }")
print(r.domain_factor_domain())
```

python invocation:

```
python < domain_factor_domain.py
```

python output:

```
{ A[2, 8, 1] -> [B[5] -> B[6]]; B[5] -> [B[5] -> B[6]] }
```

Operation 2.94 (Range Factor of Domain Product)**.** *The range factor of a binary relation R considered as a domain product consists of those pairs of elements where the first appears as the second element in the wrapped relation in the first element of a pair of elements in R and the second is the second element of that pair in R. That is, the range factor of R as a domain product is*

$$\{\, \boldsymbol{j} \to \boldsymbol{k} : \exists \boldsymbol{i} : [\boldsymbol{i} \to \boldsymbol{j}] \to \boldsymbol{k} \in R \,\}. \tag{2.44}$$

In `isl`, this operation is called `isl_union_map_domain_factor_range`.

Example 2.95. *python input (domain_factor_range.py):*

```
import isl

r = isl.union_map(
    "{ [A[2, 8, 1] -> A[2, 8, 1]] -> [B[5] -> B[6]]; "
    "[B[5] -> A[2, 8, 1]] -> [B[5] -> B[6]] }")
print(r.domain_factor_range())
```

python invocation:

```
python < domain_factor_range.py
```

python output:

```
{ A[2, 8, 1] -> [B[5] -> B[6]] }
```

Operation 2.96 (Domain Factor of Range Product)**.** *The domain factor of a binary relation R considered as a range product consists of those pairs of elements where the first appears as the first in a pair of elements in R and the second is the first element in the wrapped relation in the second element of that pair in R. That is, the domain factor of R as a range product is*

$$\{\, \boldsymbol{i} \to \boldsymbol{j} : \exists \boldsymbol{k} : \boldsymbol{i} \to [\boldsymbol{j} \to \boldsymbol{k}] \in R \,\}. \tag{2.45}$$

In `isl`, this operation is called `isl_union_map_range_factor_domain`.

Example 2.97. *python input (range_factor_domain.py):*

```
import isl

r = isl.union_map("{ A[2, 8, 1] -> [B[5] -> B[6]] }")
print(r.range_factor_domain())
```

python invocation:

```
python < range_factor_domain.py
```

python output:

```
{ A[2, 8, 1] -> B[5] }
```

Operation 2.98 (Range Factor of Range Product)**.** *The range factor of a binary relation R considered as a range product consists of those pairs of elements where the first appears as the first in a pair of elements in R and the second is the second element in the wrapped relation in the second element of that pair in R. That is, the range factor of R as a range product is*

$$\{\, \boldsymbol{i} \to \boldsymbol{k} : \exists \boldsymbol{j} : \boldsymbol{i} \to [\boldsymbol{j} \to \boldsymbol{k}] \in R \,\}. \tag{2.46}$$

In `isl`, this operation is called `isl_union_map_domain_factor_range`.

Example 2.99. *python input (range_factor_range.py):*

```
import isl

r = isl.union_map("{ A[2, 8, 1] -> [B[5] -> B[6]] }")
print(r.range_factor_range())
```

python invocation:

```
python < range_factor_range.py
```

python output:

```
{ A[2, 8, 1] -> B[6] }
```

2.4.4 Domain and Range Projection

The following operations take a binary relation as input and produce a binary relation that projects a wrapped copy of the input onto its domain or range.

Operation 2.100 (Domain Projection)**.** *The domain projection $\overrightarrow{\operatorname{dom}} R$ of a binary relation R is a binary relation that for each pair of elements in R associates a wrapped copy of that pair to the first element. That is,*

$$\underrightarrow{\operatorname{dom}} R = \{\, [\boldsymbol{i} \to \boldsymbol{j}] \to \boldsymbol{i} : \boldsymbol{i} \to \boldsymbol{j} \in R \,\}. \tag{2.47}$$

In `isl`, this operation is called `isl_union_map_domain_map`. See also Section 4.2 Creation. In `iscc`, this operation is written `domain_map`.

Example 2.101. *iscc input (domain_map.iscc):*

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
domain_map R;
```

iscc invocation:

```
iscc < domain_map.iscc
```

iscc output:

```
{ [B[5] -> B[5]] -> B[5]; [A[2, 8, 1] -> B[6]] -> A[2, 8,
    ↪ 1]; [A[2, 8, 1] -> B[5]] -> A[2, 8, 1] }
```

Operation 2.102 (Range Projection)**.** *The range projection* $\underrightarrow{\operatorname{ran}} R$ *of a binary relation* R *is a binary relation that for each pair of elements in* R *associates a wrapped copy of that pair to the second element. That is,*

$$\underrightarrow{\operatorname{ran}} R = \{ [\boldsymbol{i} \to \boldsymbol{j}] \to \boldsymbol{j} : \boldsymbol{i} \to \boldsymbol{j} \in R \}. \tag{2.48}$$

In `isl`, this operation is called `isl_union_map_range_map`. In `iscc`, this operation is written `range_map`.

Example 2.103. *iscc input (range_map.iscc):*

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
range_map R;
```

iscc invocation:

```
iscc < range_map.iscc
```

iscc output:

```
{ [B[5] -> B[5]] -> B[5]; [A[2, 8, 1] -> B[6]] -> B[6]; [A
    ↪ [2, 8, 1] -> B[5]] -> B[5] }
```

2.4.5 Difference Set Projection

The difference set of a binary relation contains the differences of the pairs of elements in the relation. The difference set projection maps the original pairs to their difference.

Operation 2.104 (Difference Set of a Binary Relation)**.** *The* difference set ΔR *of a binary relation* R *is a set containing the differences between pairs of elements in* R *that have the same space, where the difference between a pair of elements has the same space as those two elements and has values that are the difference between the values of the second and those of the first element. That is,*

$$\Delta R = \{ \boldsymbol{d} : \exists \boldsymbol{x} \to \boldsymbol{y} \in R : \mathcal{S}\boldsymbol{d} = \mathcal{S}\boldsymbol{x} = \mathcal{S}\boldsymbol{y} \wedge \mathcal{V}\boldsymbol{d} = \mathcal{V}\boldsymbol{y} - \mathcal{V}\boldsymbol{x} \}. \tag{2.49}$$

In `isl`, this operation is called `isl_union_map_deltas`. In `iscc`, this operation is written `deltas`.

Example 2.105. *iscc input (deltas.iscc):*

```
deltas { A[2,8,1] -> B[5]; B[5] -> B[6]; B[5] -> B[5] };
```

iscc invocation:

```
iscc < deltas.iscc
```

iscc output:

```
{ B[1]; B[0] }
```

Operation 2.106 (Difference Set Projection of a Binary Relation)**.** *The* difference set projection $\underset{\rightarrow}{\Delta} R$ *of a binary relation R is a binary relation mapping wrapped pairs of elements in R that have the same space to their difference. The difference between a pair of elements has the same space as those two elements and has values that are the difference between the values of the second and those of the first element. That is,*

$$\underset{\rightarrow}{\Delta} R = \{\, [\boldsymbol{x} \to \boldsymbol{y}] \to \boldsymbol{d} : \boldsymbol{x} \to \boldsymbol{y} \in R \wedge \mathcal{S}\boldsymbol{d} = \mathcal{S}\boldsymbol{x} = \mathcal{S}\boldsymbol{y} \wedge \mathcal{V}\boldsymbol{d} = \mathcal{V}\boldsymbol{y} - \mathcal{V}\boldsymbol{x} \,\}. \tag{2.50}$$

In `isl`, this operation is called `isl_union_map_deltas_map`. In `iscc`, this operation is written `deltas_map`.

Example 2.107. *iscc input (deltas_map.iscc):*

```
deltas_map { A[2,8,1] -> B[5]; B[5] -> B[6]; B[5] -> B[5] };
```

iscc invocation:

```
iscc < deltas_map.iscc
```

iscc output:

```
{ [B[5] -> B[6]] -> B[1]; [B[5] -> B[5]] -> B[0] }
```

Notes

2.1. In `isl`, the identifier of an element consists of a name and an optional user pointer. For two elements to be considered the same, they need to have the same name and the same user pointer.
2.2. This notation derives from the notation of the `Omega` library, where an element is represented by a comma delimited list of the integer values enclosed in square brackets. Identifiers were introduced in `isl` by Verdoolaege (2011). Similar notations have also been used by, e.g., Pugh (1991b) and Maslov (1994).
2.3. The reason that this type is called `isl_union_set` is that `isl` also has an `isl_set` type. See Section 3.6 Space-Local Operations.
2.4. The `->` notation is also inherited from the notation of the `Omega` library.
2.5. The 0-th power is not defined because it would have to be an identity relation on all named integer tuples, which is impossible to represent in `isl`, even intensionally.

Chapter 3

Presburger Sets and Relations

In the previous chapter, sets and binary relations were described extensionally by explicitly listing the (pairs of) element(s) contained in the set or binary relation. This chapter explains how to describe sets and binary relations intensionally through properties that need to be satisfied by the (pairs of) element(s). As in the previous chapter, sets and relations will continue to be treated purely abstractly.

3.1 Intensional Descriptions

In an intensional description, the elements of a set are described in terms of structured named integer tuple templates. These are essentially the same as structured named integer tuples, except that the integers have been replaced by variables. Compare the following definitions to Definition 2.66 on page 28, Definition 2.68 on page 29 and Definition 2.70 on page 29.

Definition 3.1 (Structured Named Integer Tuple Template)**.** *A* structured named integer tuple template *is either*

- *an identifier n along with $d \geq 0$ variables i_j for $0 \leq j < d$, written $n[i_0, i_1, \ldots, i_{d-1}]$, or,*
- *an identifier n along with two structured named integer tuple templates $\boldsymbol{i}$ and $\boldsymbol{j}$ written $n[\boldsymbol{i} \to \boldsymbol{j}]$.*

In `isl`, an *identifier* used as a variable or tuple name starts with an alphabetic character or underscore, followed by zero or more alphanumeric characters and/or underscores. A variable name may furthermore be "primed". That is, it may be followed by one or more primes ($'$). These primes serve to differentiate variables with the same name and are not considered to be part of the name.

Definition 3.2 (Space)**.** *The* space $\mathcal{S}\boldsymbol{i}$ *of a structured named integer tuple template $\boldsymbol{i}$ is*

- *n/d, if $\boldsymbol{i}$ is of the form $n[i_0, i_1, \ldots, i_{d-1}]$, with n an identifier and d a non-negative integer, or,*
- *$(n, \mathcal{S}(\boldsymbol{j}), \mathcal{S}(\boldsymbol{k}))$, if $\boldsymbol{i}$ is of the form $n[\boldsymbol{j} \to \boldsymbol{k}]$, with n an identifier and $\boldsymbol{j}$ and $\boldsymbol{k}$ structured named integer tuple templates.*

The space of a pair of structured named integer tuple templates $\boldsymbol{i} \to \boldsymbol{j}$ is $(\mathcal{S}(\boldsymbol{i}), \mathcal{S}(\boldsymbol{j}))$.

Definition 3.3 (Variable Vector)**.** *The* variable vector $\mathcal{V}\boldsymbol{i}$ *of a structured named integer tuple template $\boldsymbol{i}$ is the vector*

- *$(i_0, i_1, \ldots, i_{d-1})$, if $\boldsymbol{i}$ is of the form $n[i_0, i_1, \ldots, i_{d-1}]$, with n an identifier and d a non-negative integer, or,*
- *$\mathcal{V}(\boldsymbol{j}) \| \mathcal{V}(\boldsymbol{k})$, with $\|$ the concatenation of two vectors, if $\boldsymbol{i}$ is of the form $n[\boldsymbol{j} \to \boldsymbol{k}]$, with n an identifier and $\boldsymbol{j}$ and $\boldsymbol{k}$ structured named integer tuple templates.*

The variable vector of a pair of structured named integer tuple templates $\boldsymbol{i} \to \boldsymbol{j}$ is $\mathcal{V}(\boldsymbol{i}) \| (\mathcal{V}\boldsymbol{j})$.

The notations for sets and binary relations are then redefined in terms of these templates.

Notation 3.4 (Set)**.** *The notation for a* set *is formed by a semicolon delimited list of element descriptions enclosed in braces. An element description consists of a template followed by a colon and a formula in terms of the variables in the template.*

Within each formula the elements of the variable vector of the corresponding tuple are known as the *set variables*.

Notation 3.5 (Binary Relation)**.** *The notation for a* binary relation *is formed by a semicolon delimited list of element-pair descriptions enclosed in braces. An element-pair description consists of a pair of templates, separated by an arrow and followed by a colon and a formula in terms of the variables in the pair of templates.*

An integer tuple $\boldsymbol{i}$ belongs to a set iff the set description contains an element description such that the tuple template has the same space as $\boldsymbol{i}$ and such that the value vector of $\boldsymbol{i}$ satisfies the corresponding formula. Similarly for binary relations. The exact nature of the formulas and their satisfaction is described in Section 3.2 Presburger Formulas.

Example 3.6. *The set*

$$\{ \mathrm{B}[i] : 5 \le i \le 6; \mathrm{C}[] : \} \tag{3.1}$$

is equal to the set

$$\{ \mathrm{B}[5]; \mathrm{B}[6]; \mathrm{C}[] \} \tag{3.2}$$

in the notation of Chapter 2 Sets of Named Integer Tuples.

3.2 Presburger Formulas

A Presburger formula is a specific instance of the concept of a first order formula. This general concept is defined first.

Definition 3.7 (Language)**.** *A* language

$$\mathcal{L} = \{\, f_1/r_1, f_2/r_2, \ldots, P_1/s_1, P_2/s_2, \ldots\} \tag{3.3}$$

is a collection of function *symbols f_i and* predicate *symbols P_i, each with their own arity r_1 or s_i, i.e., the number of arguments they require. A function with arity zero is called a* constant*.*

Definition 3.8 (Term)**.** *A* term *in a language $\mathcal{L}$ is inductively defined as either*

- *v, with v a variable, or,*
- *$f_i(t_1, \ldots, t_{r_i})$, with f_i a function symbol in $\mathcal{L}$ with arity r_i and t_j terms for $1 \leq j \leq r_i$. In particular, if $r_i = 0$, then $f_i()$ is a term.*

Definition 3.9 (First Order Formula)**.** *A* first order formula *in a language $\mathcal{L}$ is inductively defined as either* Note 3.1

- true*,*
- *$P_i(t_1, \ldots t_{s_i})$, with P_i a predicate symbol in $\mathcal{L}$ with arity s_i and t_j terms in $\mathcal{L}$ for $1 \leq j \leq s_i$,*
- *$t_1 = t_2$, for t_1 and t_2 terms in $\mathcal{L}$,*
- *$F_1 \wedge F_2$, called the* conjunction *of two formulas F_1 and F_2,*
- *$F_1 \vee F_2$, called the* disjunction *of two formulas F_1 and F_2,*
- *$\neg F$, called the* negation *of formula F,*
- *$\exists v : F$, the* existential quantification *of formula F over the variable v, or,*
- *$\forall v : F$, the* universal quantification *of formula F over the variable v.*

Definition 3.10 (Free and Bound Variables)**.** *An occurrence of variable v is said to be* bound *in a formula F if F has a subformula $\exists v : F_1$ or $\forall v : F_2$ and v appears inside F_1 or F_2. The occurrence is said to be* free *otherwise.*

Note that the same variable can appear as both a free and a bound variable in the same formula, but at different occurrences.

Definition 3.11 (Closed Formula)**.** *A formula is called* closed *if it does not contain any free variables.*

Definition 3.12 (Presburger Language)**.** *The* Presburger language *is the first order language with as function symbols* Note 3.2

- $+/2$
- $-/2$
- *a constant symbol $d/0$ for each integer d*
- Note 3.3 *a unary function symbol $\lfloor \cdot / d \rfloor$ for each positive integer d*
- Note 3.4 *a set of constant symbols $c_i/0$*

and as single predicate symbol

- $\leq /2.$

Definition 3.13 (Presburger Term)**.** *A* Presburger term *is a term in the Presburger language.*

Note 3.5 **Definition 3.14** (Presburger Formula)**.** *A* Presburger formula *is a first order formula in the Presburger language.*

In order to be able to evaluate whether a first order formula is satisfied, a domain of discourse and an interpretation for all the function and predicate symbols need to be considered. The domain of discourse (or "universe") is the set of values that is used for the variables in the formulas. The interpretation maps a function or predicate symbol to an actual function or predicate. In the case of Presburger formulas, the domain of discourse is the set of integers $\mathbb{Z}$. Note 3.6

Definition 3.15 (Interpretation of Presburger Symbols)**.** *The following interpretation is given to the function and predicate symbols in Presburger formulas.*

- *the function symbol $+/2$ is mapped to the function that adds two integers;*
- *the function symbol $-/2$ is mapped to the function that subtracts the second integer argument from the first;*
- *each constant symbol $d/0$ is mapped to the corresponding integer value;*
- *each function symbol $\lfloor \cdot / d \rfloor$ is mapped to the function that returns the result of the integer division of its integer argument by d;*
- *the predicate symbol $\leq /2$ is mapped to the less-than-or-equal relation on integers.*

The constant symbols c_i are not assigned a fixed interpretation. Instead, all possible interpretations as integers are considered. The interpretation of a Presburger term is the result of recursively applying the interpretation to the function symbols that appear in the term.

The following definition defines the concept of the truth value of a general first order formula in terms of the domain of discourse (or universe) and the interpretation of the symbols. The definition makes use of substitutions of the form $F\{v \mapsto d\}$, which refers to the result of replacing every *free* occurrence of v in F by d.

Definition 3.16 (Truth Value)**.** *The truth value of a first order formula in a given universe and interpretation is determined as follows.*

- *The formula* true *is true.*
- *The formula $P_i(t_1, \ldots t_{s_i})$ is true if the interpretation of P_i applied to the interpretations of t_j is true.*
- *The formula $t_1 = t_2$ is true if the interpretations of t_1 and t_2 are equal.*
- *The formula $F_1 \wedge F_2$ is true if both F_1 and F_2 are true.*
- *The formula $F_1 \vee F_2$ is true if at least one of F_1 or F_2 is true.*
- *The formula $\neg F$ is true if F is not true.*
- *The formula $\exists v : F(v)$ is true if $F\{v \mapsto d\}$ is true for* some *d in the universe.*
- *The formula $\forall v : F(v)$ is true if $F\{v \mapsto d\}$ is true for* every *d in the universe.*

3.3 Presburger Sets and Relations

Definition 3.17 (Presburger Set)**.** *A* Presburger set *is a set in the notation of Notation 3.4 on page 42 where the formula is a Presburger formula as in Definition 3.14 on the preceding page. The only free variables allowed in this formula are the variables of the tuple template.*

Definition 3.18 (Presburger Relation)**.** *A* Presburger relation *is a binary relation in the notation of Notation 3.5 on page 42 where the formula is a Presburger formula as in Definition 3.14 on the preceding page. The only free variables allowed in this formula are the variables of the pair of tuple templates.*

As already explained in Section 3.1 Intensional Descriptions, an integer tuple $\boldsymbol{i}$ belongs to a set iff the set description contains an element description $\boldsymbol{t} : F$ such that the tuple template has the same space as $\boldsymbol{i}$, i.e., $\mathcal{S}\boldsymbol{i} = \mathcal{S}\boldsymbol{t}$, and such that the value vector of $\boldsymbol{i}$ satisfies the corresponding formula, i.e., $F\{\mathcal{V}\boldsymbol{t} \mapsto \mathcal{V}\boldsymbol{i}\}$ is true.

Example 3.19. *The set*

$$\{\, [i] : 0 \le i \wedge i \le 10 \wedge \exists \alpha : i = \alpha + \alpha \,\} \tag{3.4}$$

is equal to the set

$$\{\, [0]; [2]; [4]; [6]; [8]; [10] \,\}. \tag{3.5}$$

Example 3.20. *The set*

$$\{\,[i] : \forall i : 0 \le i \wedge i \le 10\,\} \tag{3.6}$$

is empty because the subformula $0 \le i \wedge i \le 10$ *is only true for some values of* i*, but not all integer values of* i*. This means that the formula* $\forall i : 0 \le i \wedge i \le 10$ *is not true and there are therefore no values of the tuple variable* i *for which this formula is true.*

If the formula in an element description contains any constant symbols, then the truth value of the formula may depend on the interpretation of these constant symbols. As such, a Presburger set essentially represents a family of sets, one for each value of the constant symbols.

Example 3.21. *Consider the Presburger set*

$$\{\,\mathrm{S}[i] : 0 \le i \wedge i \le \mathrm{n}\,\}. \tag{3.7}$$

Depending on the value assigned to the constant symbol n*, this description corresponds to one of the following sets.*

$$\begin{cases} \emptyset & \textit{if}\ \mathrm{n} < 0 \\ \{\,\mathrm{S}[0]\,\} & \textit{if}\ \mathrm{n} = 0 \\ \{\,\mathrm{S}[0]; \mathrm{S}[1]\,\} & \textit{if}\ \mathrm{n} = 1 \\ \{\,\mathrm{S}[0]; \mathrm{S}[1]; \mathrm{S}[2]\,\} & \textit{if}\ \mathrm{n} = 2 \\ \ldots \end{cases} \tag{3.8}$$

Table 3.1 on the facing page shows the `isl` notation for the first order logic connectives of Definition 3.9 on page 43 and the Presburger symbols of Definition 3.12 on page 43, along with some syntactic sugar that will be explained in Section 3.4 Syntactic Sugar.

Notation 3.22 (Constant Symbols)**.** *In this text, a constant symbol will by typeset in roman type. In* `isl`*, a constant symbol is called a* parameter*. A parameter has the same appearance as a variable, but it has to be declared in front of the set or binary relation description. In particular, all parameters need to be placed in a comma separated list enclosed in brackets and followed by a* `->` *in front of the set or binary relation description. The order of the parameters inside the list is immaterial.*

Example 3.23. *Consider the set in* (3.7)*. Its* `isl` *representation is as follows.*

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

In some cases, it can be convenient to reason about the values of the constant symbols for which a given set or relation is non-empty. They can be represented as a *unit set*, which is a set that does not contain any tuples, but that is still considered empty or non-empty depending on the values of the constant symbols. The notation for unit sets is similar to that of sets in Notation 3.4 on page 42, except that it does not contain any tuple templates.

$+$	`+`
$-$	`-`
$=$	`=`
$\leq$	`<=`
$<$	`<`
$\geq$	`>=`
$>$	`>`
$\neq$	`!=`
$,$	`,`
$\cdot$	`*`
$\preccurlyeq$	`<<=`
$\prec$	`<<`
$\succcurlyeq$	`>>=`
$\succ$	`>>`
mod	`mod`
true	`true`
false	`false`
$\wedge$	`and` or `&` or `&&` or `/\`
$\vee$	`or` or `|` or `||` or `\/`
$\neg$	`not` or `!`
$\Longrightarrow$	`implies`
$\exists v :$	`exists` v `:`
$\forall v :$	`not exists` v `: not`

Table 3.1: `isl` notation for Presburger formulas

Notation 3.24 (Unit Set)**.** *The notation for a* unit set *is formed by a colon and a constant formula (depending only on the symbolic constants) enclosed in braces.*

In `isl`, unit sets are called *parameter sets* and they are represented by an `isl_set`.

Example 3.25. *The conditions under which the set in* (3.7) *is non-empty can be described as*

$$\{ : \mathrm{n} \geq 0 \}, \tag{3.9}$$

or, in `isl` notation

```
[n] -> { : n >= 0 }
```

Most of the operations defined in Chapter 2 Sets of Named Integer Tuples are not affected by the presence of constant symbols. The operation is simply applied uniformly for all possible values of those constant symbols. Some operations, in particular the comparison operations, are affected, however.

Operation 3.26 (Equality of Sets)**.** *Two sets A and B are equal ($A = B$) if they contain the same elements* for every value of the constant symbols.

Example 3.27. *The set*

$$\{\, \mathrm{a}[i] : i \geq 0 \,\} \tag{3.10}$$

is not *equal to the set*

$$\{\, \mathrm{a}[i] : i \geq 0 \wedge \mathrm{n} \geq 0 \,\} \tag{3.11}$$

because the second set is empty when n *is negative, while the first one contains infinitely many elements for any value of* n.
iscc input (set_equal_constant.iscc):

```
A := [n] -> { A[i] : i >= 0 };
B := [n] -> { A[i] : i >= 0 and n >= 0 };
A = B;
```

iscc invocation:

```
iscc < set_equal_constant.iscc
```

iscc output:

```
False
```

Example 3.28. *The set*

$$\{\, \mathrm{a}[i] : 0 \leq i < \mathrm{n} \,\} \tag{3.12}$$

is not *equal to the set*

$$\{\, \mathrm{a}[i] : 0 \leq i < \mathrm{m} \,\} \tag{3.13}$$

because the constant symbols n *and* m *do not necessarily have the same value.*
iscc input (set_equal_constant2.iscc):

```
A := [n] -> { A[i] : 0 <= i < n };
B := [m] -> { A[i] : 0 <= i < m };
A = B;
```

iscc invocation:

```
iscc < set_equal_constant2.iscc
```

iscc output:

```
False
```

Example 3.29. *The set*

$$\{\, \mathrm{a}[n, i] : 0 \leq i < n \,\} \tag{3.14}$$

is *equal to the set*

$$\{\, \mathrm{a}[m, i] : 0 \leq i < m \,\} \tag{3.15}$$

because the two sets contain the same pairs of integers.
iscc input (set_equal2.iscc):

```
A := { A[n, i] : 0 <= i < n };
B := { A[m, i] : 0 <= i < m };
A = B;
```

iscc invocation:

```
iscc < set_equal2.iscc
```

iscc output:

```
True
```

Operation 3.30 (Equality of Binary Relations)**.** *Two binary relations A and B are equal ($A = B$) if they contain the same pairs of elements* for every value of the constant symbols.

Operation 3.31 (Emptiness of a Set)**.** *A set is empty if it does not contain any element* for any value of the constant symbols.

Example 3.32. *The set in* (3.15) *is only empty for some values of the constant* n*, but not for all values. It is therefore not considered to be empty.*
iscc input (set_empty_constant.iscc):

```
[n] -> { A[i] : i >= 0 and n >= 0 } = { };
```

iscc invocation:

```
iscc < set_empty_constant.iscc
```

iscc output:

```
False
```

Operation 3.33 (Emptiness of a Binary Relation)**.** *A binary relation is empty if it does not contain any pair of elements* for any value of the constant symbols.

Operation 3.34 (Subset)**.** *The set A is a subset of the set B, $A \subseteq B$, if all elements of A are contained in B* for every value of the constant symbols, *i.e., if $A \setminus B = \emptyset$.*

Example 3.35. *iscc input (set_subset_constant.iscc):*

```
A := [n] -> { A[i] : i >= 0 };
B := [n] -> { A[i] : i >= 0 and n >= 0 };
B <= A;
```

iscc invocation:

```
iscc < set_subset_constant.iscc
```

iscc output:

```
True
```

Operation 3.36 (Subrelation)**.** *The binary relation A is a subset of the binary relation B, $A \subseteq B$, if all pairs of elements in A are contained in B* for every value of the constant symbols*, i.e., if $A \setminus B = \emptyset$.*

Operation 3.37 (Strict Subset)**.** *The set A is a strict (or proper) subset of the set B, $A \subsetneq B$, if all elements of A are contained in B* for every value of the constant symbols *and B contains elements not in A* for some value of the constant symbols*, i.e., if $A \setminus B = \emptyset$ and $A \neq B$.*

Example 3.38. `iscc` *input (*`set_strict_subset_constant.iscc`*):*

```
A := [n] -> { A[i] : i >= 0 };
B := [n] -> { A[i] : i >= 0 and n >= 0 };
B < A;
```

`iscc` *invocation:*

```
iscc < set_strict_subset_constant.iscc
```

`iscc` *output:*

```
True
```

Operation 3.39 (Strict Subrelation)**.** *The binary relation A is a strict (or proper) subset of the binary relation B, $A \subsetneq B$, if all pairs of elements of A are contained in B* for every value of the constant symbols *and B contains pairs of elements not in A* for some value of the constant symbols*, i.e., if $A \setminus B = \emptyset$ and $A \neq B$.*

Operation 3.40 (Superset)**.** *The set A is a superset of the set B, $A \supseteq B$, if all elements of B are contained in A* for every value of the constant symbols*, i.e., if $B \subseteq A$.*

Operation 3.41 (Superrelation)**.** *The binary relation A is a superset of the binary relation B, $A \supseteq B$, if all pairs of elements in B are contained in A* for every value of the constant symbols*, i.e., if $B \subseteq A$.*

Operation 3.42 (Strict Superset)**.** *The set A is a strict (or proper) superset of the set B, $A \supsetneq B$, if all elements of B are contained in A* for every value of the constant symbols *and A contains elements not in B* for some value of the constant symbols*, i.e., if $B \subsetneq A$.*

Operation 3.43 (Strict Superrelation)**.** *The binary relation A is a strict (or proper) superset of the binary relation B, $A \supsetneq B$, if all pairs of elements in B are contained in A* for every value of the constant symbols *and A contains pairs of elements not in B* for some value of the constant symbols*, i.e., if $B \subsetneq A$.*

Operation 3.44 (Single-valued)**.** *A binary relation R is* single-valued*, i.e., a* function*, if every element that appears as the first element in a pair of elements in R only appears as the first element in one such pair* for every value of the constant symbols*.*

Operation 3.45 (Injective)**.** *A binary relation R is* injective *if every element that appears as the second element in a pair of elements in R only appears as the second element in one such pair* for every value of the constant symbols*, i.e., if its inverse is single-valued.*

3.4 Syntactic Sugar

Notation 3.46 (False)**.** *The formula* false *is equivalent to* ¬true*.*

Notation 3.47 (Implication)**.** *The formula $a \implies b$ is equivalent to $\neg a \vee b$.*

The formula following the tuple template in Notation 3.4 on page 42 and Notation 3.5 on page 42 is optional. If the formula is missing, then it is taken to be true.

Example 3.48. *`iscc` input (`missing_formula.iscc`):*

```
A := { A[i] : true };
B := { A[i] };
A = B;
```

`iscc` invocation:

```
iscc < missing_formula.iscc
```

`iscc` output:

```
True
```

The variables in a tuple template may be directly assigned Presburger terms that only involve variables that appear in *earlier* positions in the template.

Notation 3.49. *Let $\boldsymbol{v} = (v_1, \ldots, v_n) = \mathcal{V}\boldsymbol{t}$ be the variables of a tuple template $\boldsymbol{t}$. An element description $\boldsymbol{t} : v_k = g(v_1, \ldots, v_{k-1}) \wedge f(\boldsymbol{v})$ may be rewritten as $\boldsymbol{t}\{v_k \mapsto v_k = g(v_1, \ldots, v_{k-1})\} : f(\boldsymbol{v})$.*

Example 3.50. *The binary relation*

$$\{\, \mathrm{S}[i] \to \mathrm{S}[j = i + 1] \,\} \tag{3.16}$$

is equal to

$$\{\, \mathrm{S}[i] \to \mathrm{S}[j] : j = i + 1 \,\}. \tag{3.17}$$

Note, however, that the syntax $\{\, \mathrm{S}[i = j - 1] \to \mathrm{S}[j] \,\}$ is not allowed since the expression $j - 1$ contains variables other than those that appear in earlier positions.
`iscc` input (`tuple_equation.iscc`):

```
A := { S[i] -> S[j = i + 1] };
B := { S[i] -> S[j] : j = i + 1 };
A = B;
```

`iscc` *invocation:*

```
iscc < tuple_equation.iscc
```

`iscc` *output:*

```
True
```

The variables in a tuple template may also be simply replaced by Presburger terms that only involve variables that appear in earlier positions in the template. Note 3.7

Notation 3.51. *Let $\boldsymbol{v} = (v_1, \ldots, v_n) = \mathcal{V}\boldsymbol{t}$ be the variables of a tuple template $\boldsymbol{t}$. An element description $\boldsymbol{t} : v_k = g(v_1, \ldots, v_{k-1} \wedge f(\boldsymbol{v})$ may be rewritten as $\boldsymbol{t}\{v_k \mapsto g(v_1, \ldots, v_{k-1}\} : f(\boldsymbol{v})$.*

Example 3.52. *The binary relation*

$$\{\, \mathrm{S}[i] \to \mathrm{S}[i+1] \,\} \tag{3.18}$$

is equal to

$$\{\, \mathrm{S}[i] \to \mathrm{S}[j] : j = i + 1 \,\}. \tag{3.19}$$

Note that, as in Example 3.50 on the preceding page, the syntax $\{\, \mathrm{S}[j-1] \to \mathrm{S}[j] \,\}$ is not allowed since the expression $j - 1$ contains variables other than those that appear in earlier positions.
`iscc` *input (`tuple_expression.iscc`):*

```
A := { S[i] -> S[i + 1] };
B := { S[i] -> S[j] : j = i + 1 };
A = B;
```

`iscc` *invocation:*

```
iscc < tuple_expression.iscc
```

`iscc` *output:*

```
True
```

Notation 3.51, together with the optionality of the formula in the element description, allows the syntax of Chapter 2 Sets of Named Integer Tuples to be treated as a special case of the present syntax since a tuple $N[\boldsymbol{d}]$ can be treated as the element description $N[\boldsymbol{v}] : \boldsymbol{v} = \boldsymbol{d}$, representing the same element.

Notation 3.53. *The symbol $<$ /2 represents the less-than relation on integers. The formula $a < b$ is equivalent to $a \leq b - 1$.*

Notation 3.54. *The symbol* $\geq /2$ *represents the greater-than-or-equal relation on integers. The formula* $a \geq b$ *is equivalent to* $b \leq a$*.*

Notation 3.55. *The symbol* $> /2$ *represents the greater-than relation on integers. The formula* $a > b$ *is equivalent to* $a \geq b + 1$*.*

Notation 3.56. *The symbol* $\neq /2$ *represents the not-equal relation on integers. The formula* $a \neq b$ *is equivalent to* $\neg(a = b)$*.*

Notation 3.57. *The same comparison can be performed on multiple arguments by separating the arguments with a comma. The formula* $a, b \oplus c$*, where* $\oplus \in \{ \leq, <, \geq, >, =, \neq \}$*, is equivalent to* $a \oplus c \wedge b \oplus c$*.*

Example 3.58. *The set*

$$\{ \mathrm{S}[i, j] : i, j \geq 0 \} \tag{3.20}$$

is equal to

$$\{ \mathrm{S}[i, j] : i \geq 0 \wedge j \geq 0 \}. \tag{3.21}$$

iscc input (comma.iscc):

```
A := { S[i,j] : i,j >= 0 };
B := { S[i,j] : i >= 0 and j >= 0 };
A = B;
```

iscc invocation:

```
iscc < comma.iscc
```

iscc output:

```
True
```

Notation 3.59. *Operators can be chained. That is, a given argument may be used as both the right hand side of one operator and the left hand side of the next operator. In particular, a formula of the form* $a \oplus_1 b \oplus_2 c$*, with* $\{ \oplus_1, \oplus_2 \} \subset \{ \leq, <, \geq, >, =, \neq \}$*, is equivalent to* $a \oplus_1 b \wedge b \oplus_2 c$*.*

Example 3.60. *The set*

$$\{ \mathrm{S}[i] : 0 \leq i \leq 10 \} \tag{3.22}$$

is equal to

$$\{ \mathrm{S}[i] : 0 \leq i \wedge i \leq 10 \}. \tag{3.23}$$

iscc input (chain.iscc):

```
A := { S[i] : 0 <= i <= 10 };
B := { S[i] : 0 <= i and i <= 10 };
A = B;
```

`iscc` invocation:

```
iscc < chain.iscc
```

`iscc` output:

```
True
```

Notation 3.61. *The unary function $-/1$ represents the negation on integers. That is, the formula $-a$ is equivalent to $0 - a$.*

Notation 3.62. *The notation $\mathrm{n} \cdot e$, with n a non-negative integer constant, is a shorthand for*

$$\underbrace{e + e + \cdots + e}_{n \text{ times}}. \tag{3.24}$$

The dot $(\cdot)$ may also be omitted.

An actual multiplication, where the left-hand side is a variable or a constant symbol, is not allowed in Presburger formulas.

Example 3.63. *The set*

$$\{ [i] : 0 \le i \le 10 \wedge i = 2\mathrm{n} \} \tag{3.25}$$

is equal to the set $\{ [2\mathrm{n}] \}$ when the value assigned to the constant symbol n *satisfies $0 \le \mathrm{n} \le 5$ and is equal to the empty set otherwise.*

Notation 3.64. *The formula $a \bmod b$, with b a non-negative integer constant, is equivalent to $a - b \cdot \lfloor a/b \rfloor$.*

The following symbols deal with lexicographic order, which is defined first.

Definition 3.65 (Lexicographic Order)**.** *Given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ of equal length, $\boldsymbol{a}$ is said to be lexicographically smaller than $\boldsymbol{b}$ if it is not equal to b and if it is smaller in the first position in which it differs from $\boldsymbol{b}$. If the shared length of the two vectors is n, then this condition can be expressed as the Presburger formula* Note 3.8

$$\bigvee_{i:1 \le i \le n} \left(\left(\bigwedge_{j:1 \le j < i} a_j = b_j \right) \wedge a_i < b_i \right). \tag{3.26}$$

Notation 3.66. *The symbol $\prec /2$ represents the lexicographically smaller-than relation on equal-length sequences of integers. That is, $\boldsymbol{a} \prec \boldsymbol{b}$, with $\boldsymbol{a}$ and $\boldsymbol{b}$ two sequences of length n, if the elements in the sequences satisfy the formula in* (3.26)*.*

Example 3.67. *The binary relation*

$$\{ \mathrm{S}[i_1, i_2] \to \mathrm{S}[j_1, j_2] : i_1, i_2 \prec j_1, j_2 \} \tag{3.27}$$

is equal to

$$\{\, \mathrm{S}[i_1, i_2] \to \mathrm{S}[j_1, j_2] : i_1 < j_1 \vee (i_1 = j_1 \wedge i_2 < j_2) \,\}. \tag{3.28}$$

`iscc` *input* (`lex.iscc`):

```
A := { S[i1,i2] -> S[j1,j2] : i1,i2 << j1,j2 };
B := { S[i1,i2] -> S[j1,j2] :
        i1 < j1 or (i1 = j1 and i2 < j2) };
A = B;
```

`iscc` *invocation:*

```
iscc < lex.iscc
```

`iscc` *output:*

```
True
```

Alternative 3.68 (Extended Lexicographic Order)**.** *Some authors consider an extended form of lexicographic order that is also defined on pairs of vectors of different sizes. The shorter vector is then typically compared to the initial elements of the longer vector. This still leaves the issue of how two vectors compare if one is a proper prefix of the other. For this issue, it is either assumed that no such comparison is ever performed or some implicit order is defined.*

Notation 3.69. *The symbol* $\preccurlyeq$ */2 represents the lexicographically smaller-than-or-equal relation on equal-length sequences of integers. That is,* $\boldsymbol{a} \preccurlyeq \boldsymbol{b}$ *is equivalent to* $\boldsymbol{a} \prec \boldsymbol{b} \vee \boldsymbol{a} = \boldsymbol{b}$*.*

Notation 3.70. *The symbol* $\succ$ */2 represents the lexicographically greater-than relation on equal-length sequences of integers. That is,* $\boldsymbol{a} \succ \boldsymbol{b}$ *is equivalent to* $\boldsymbol{b} \prec \boldsymbol{a}$*.*

Notation 3.71. *The symbol* $\succcurlyeq$ */2 represents the lexicographically greater-than-or-equal relation on equal-length sequences of integers. That is,* $\boldsymbol{a} \succcurlyeq \boldsymbol{b}$ *is equivalent to* $\boldsymbol{a} \succ \boldsymbol{b} \vee \boldsymbol{a} = \boldsymbol{b}$*.*

3.5 Lexicographic Order

The lexicographic order of Definition 3.65 on the facing page is defined on a pair of vectors, but this concept can be extended to a pair of sets. The result is then a binary relation that contains pairs of elements from the two sets such that the first is lexicographically smaller than the second. Since it only makes sense to lexicographically compare two elements that have the same space, this means in particular that the pairs of elements in the result have the same space.

Operation 3.72 (Lexicographically-smaller-than Relation on Sets)**.** *The* lexicographically-smaller-than relation *$A \prec B$ on two sets A and B is a binary relation that contains pairs of elements, one from A and one from B such that the two elements have the same space and the first is lexicographically smaller than the second. That is,*

$$A \prec B = \{\, \boldsymbol{a} \to \boldsymbol{b} : \boldsymbol{a} \in A \wedge \boldsymbol{b} \in B \wedge \mathcal{S}\boldsymbol{a} = \mathcal{S}\boldsymbol{b} \wedge \mathcal{V}\boldsymbol{a} \prec \mathcal{V}\boldsymbol{b} \,\}. \tag{3.29}$$

In `isl`, this operation is called `isl_union_set_lex_lt_union_set`. In `iscc`, this operation is written `<<`.

Operation 3.73 (Lexicographically-smaller-than-or-equal Relation on Sets)**.** *The* lexicographically-smaller-than-or-equal relation *$A \preccurlyeq B$ on two sets A and B is a binary relation that contains pairs of elements, one from A and one from B such that the two elements have the same space and the first is lexicographically smaller than or equal to the second. That is,*

$$A \preccurlyeq B = \{\, \boldsymbol{a} \to \boldsymbol{b} : \boldsymbol{a} \in A \wedge \boldsymbol{b} \in B \wedge \mathcal{S}\boldsymbol{a} = \mathcal{S}\boldsymbol{b} \wedge \mathcal{V}\boldsymbol{a} \preccurlyeq \mathcal{V}\boldsymbol{b} \,\}. \tag{3.30}$$

In `isl`, this operation is called `isl_union_set_lex_le_union_set`. In `iscc`, this operation is written `<<=`.

Operation 3.74 (Lexicographically-greater-than Relation on Sets)**.** *The* lexicographically-greater-than relation *$A \succ B$ on two sets A and B is a binary relation that contains pairs of elements, one from A and one from B such that the two elements have the same space and the first is lexicographically greater than the second. That is,*

$$A \succ B = \{\, \boldsymbol{a} \to \boldsymbol{b} : \boldsymbol{a} \in A \wedge \boldsymbol{b} \in B \wedge \mathcal{S}\boldsymbol{a} = \mathcal{S}\boldsymbol{b} \wedge \mathcal{V}\boldsymbol{a} \succ \mathcal{V}\boldsymbol{b} \,\}. \tag{3.31}$$

In `isl`, this operation is called `isl_union_set_lex_gt_union_set`. In `iscc`, this operation is written `>>`.

Operation 3.75 (Lexicographically-greater-than-or-equal Relation on Sets)**.** *The* lexicographically-greater-than-or-equal relation *$A \succcurlyeq B$ on two sets A and B is a binary relation that contains pairs of elements, one from A and one from B such that the two elements have the same space and the first is lexicographically greater than or equal to the second. That is,*

$$A \succcurlyeq B = \{\, \boldsymbol{a} \to \boldsymbol{b} : \boldsymbol{a} \in A \wedge \boldsymbol{b} \in B \wedge \mathcal{S}\boldsymbol{a} = \mathcal{S}\boldsymbol{b} \wedge \mathcal{V}\boldsymbol{a} \succcurlyeq \mathcal{V}\boldsymbol{b} \,\}. \tag{3.32}$$

In `isl`, this operation is called `isl_union_set_lex_ge_union_set`. In `iscc`, this operation is written `>>=`.

Example 3.76. *The following transcript shows the different lexicographic order relations computed from the sets*

$$\{\, \mathrm{A}[i,j] : 0 \le i,j < 10; \mathrm{B}[]; \mathrm{C}[i] : 0 \le i < 100 \,\} \tag{3.33}$$

and

$$\{\, \mathrm{A}[i,j] : 0 \le i,j < 20; \mathrm{B}[] \,\}. \tag{3.34}$$

`iscc` input (`lexorder.iscc`):

```
A := { A[i,j] : 0 <= i,j < 10; B[]; C[i] : 0 <= i < 100 };
B := { A[i,j] : 0 <= i,j < 20; B[] };
A << B;
A <<= B;
A >> B;
A >>= B;
```

`iscc` *invocation:*

```
iscc < lexorder.iscc
```

`iscc` *output:*

```
{ A[i, j] -> A[i', j'] : 0 <= i <= 9 and 0 <= j <= 9 and i'
    ↪ > i and 0 <= i' <= 19 and 0 <= j' <= 19; A[i, j] -> A
    ↪ [i' = i, j'] : 0 <= i <= 9 and 0 <= j <= 9 and j' > j
    ↪  and 0 <= j' <= 19 }
{ B[] -> B[]; A[i, j] -> A[i', j'] : 0 <= i <= 9 and 0 <= j
    ↪ <= 9 and i' > i and 0 <= i' <= 19 and 0 <= j' <= 19;
    ↪ A[i, j] -> A[i' = i, j'] : 0 <= i <= 9 and 0 <= j <=
    ↪ 9 and j' >= j and 0 <= j' <= 19 }
{ A[i, j] -> A[i', j'] : 0 <= i <= 9 and 0 <= j <= 9 and 0
    ↪ <= i' <= 19 and i' < i and 0 <= j' <= 19; A[i, j] ->
    ↪ A[i' = i, j'] : 0 <= i <= 9 and 0 <= j <= 9 and 0 <=
    ↪ j' <= 19 and j' < j }
{ B[] -> B[]; A[i, j] -> A[i', j'] : 0 <= i <= 9 and 0 <= j
    ↪ <= 9 and 0 <= i' <= 19 and i' < i and 0 <= j' <= 19;
    ↪ A[i, j] -> A[i' = i, j'] : 0 <= i <= 9 and 0 <= j <=
    ↪ 9 and 0 <= j' <= 19 and j' <= j }
```

The same operations are also available on binary relations, but in this case the comparison is performed on the *range elements* of the input relations and the result collects the corresponding domain elements.

Operation 3.77 (Lexicographically-smaller-than Relation on Binary Relations)**.** *The* lexicographically-smaller-than relation $A \prec B$ *on two binary relations A and B is a binary relation that contains pairs of elements, one from the domain of A and one from the domain of B, that have corresponding range elements such that the first is lexicographically smaller than the second. That is,*

$$A \prec B = \{\, \boldsymbol{a} \to \boldsymbol{b} : \exists \boldsymbol{c}, \boldsymbol{d} : \boldsymbol{a} \to \boldsymbol{c} \in A \wedge \boldsymbol{b} \to \boldsymbol{d} \in B \wedge \mathcal{S}\boldsymbol{c} = \mathcal{S}\boldsymbol{d} \wedge \mathcal{V}\boldsymbol{c} \prec \mathcal{V}\boldsymbol{d} \,\}. \tag{3.35}$$

In `isl`, this operation is called `isl_union_map_lex_lt_union_map`. In `iscc`, this operation is written `<<`.

Operation 3.78 (Lexicographically-smaller-than-or-equal Relation on Binary Relations)**.** *The* lexicographically-smaller-than-or-equal relation $A \preccurlyeq B$ *on two* *binary relations A and B is a binary relation that contains pairs of elements, one from the domain of A and one from the domain of B, that have corresponding range elements such that the first is lexicographically smaller than or equal to the second. That is,*

$$A \preccurlyeq B = \{\, \boldsymbol{a} \to \boldsymbol{b} : \exists \boldsymbol{c}, \boldsymbol{d} : \boldsymbol{a} \to \boldsymbol{c} \in A \wedge \boldsymbol{b} \to \boldsymbol{d} \in B \wedge \mathcal{S}\boldsymbol{c} = \mathcal{S}\boldsymbol{d} \wedge \mathcal{V}\boldsymbol{c} \preccurlyeq \mathcal{V}\boldsymbol{d} \,\}. \tag{3.36}$$

In `isl`, this operation is called `isl_union_map_lex_le_union_map`. In `iscc`, this operation is written `<<=`.

Operation 3.79 (Lexicographically-greater-than Relation on Binary Relations)**.** *The* lexicographically-greater-than relation *$A \succ B$ on two binary relations A and B is a binary relation that contains pairs of elements, one from the domain of A and one from the domain of B, that have corresponding range elements such that the first is lexicographically greater than the second. That is,*

$$A \succ B = \{\, \boldsymbol{a} \to \boldsymbol{b} : \exists \boldsymbol{c}, \boldsymbol{d} : \boldsymbol{a} \to \boldsymbol{c} \in A \wedge \boldsymbol{b} \to \boldsymbol{d} \in B \wedge \mathcal{S}\boldsymbol{c} = \mathcal{S}\boldsymbol{d} \wedge \mathcal{V}\boldsymbol{c} \succ \mathcal{V}\boldsymbol{d} \,\}. \tag{3.37}$$

In `isl`, this operation is called `isl_union_map_lex_gt_union_map`. In `iscc`, this operation is written `>>`.

Operation 3.80 (Lexicographically-greater-than-or-equal Relation on Binary Relations)**.** *The* lexicographically-greater-than-or-equal relation *$A \succcurlyeq B$ on two binary relations A and B is a binary relation that contains pairs of elements, one from the domain of A and one from the domain of B, that have corresponding range elements such that the first is lexicographically greater than or equal to the second. That is,*

$$A \succcurlyeq B = \{\, \boldsymbol{a} \to \boldsymbol{b} : \exists \boldsymbol{c}, \boldsymbol{d} : \boldsymbol{a} \to \boldsymbol{c} \in A \wedge \boldsymbol{b} \to \boldsymbol{d} \in B \wedge \mathcal{S}\boldsymbol{c} = \mathcal{S}\boldsymbol{d} \wedge \mathcal{V}\boldsymbol{c} \succcurlyeq \mathcal{V}\boldsymbol{d} \,\}. \tag{3.38}$$

In `isl`, this operation is called `isl_union_map_lex_ge_union_map`. In `iscc`, this operation is written `>>=`.

Example 3.81. *`iscc` input (`lexorder_map.iscc`):*

```
A := { A[i,j] -> [i,0,j] };
B := { B[i,j] -> [j,1,i] };
A << B;
```

`iscc` invocation:

```
iscc < lexorder_map.iscc
```

`iscc` output:

```
{ A[i, j] -> B[i', j'] : j' > i; A[i, j] -> B[i', j' = i] }
```

3.6 Space-Local Operations

Some operations are performed separately for each (pair of) space(s) of elements in a set or binary relation. Such operations are called space-local. A typical example is lexicographic optimization. The basic assumption is that only elements within the same space can be compared to each other. In particular, only within a given space can two elements be compared lexicographically. This should be clear for spaces with different dimensions, barring Alternative 3.68 Extended Lexicographic Order, but it also holds for spaces with different identifiers or a different internal structure.

The space decomposition of a set partitions the elements in the set according to their spaces.

Operation 3.82 (Space Decomposition of a Set)**.** *Given a set S, its* space decomposition *$\mathcal{D}S$ is the unique collection of sets S_i such that the union of the S_i is equal to S, all elements of a given S_i have the same space and no two elements from distinct S_i have the same space. That is, let $\{ U_i \}_i \coloneqq \{ U : \exists \boldsymbol{x} \in S : U = \mathcal{S}\boldsymbol{x} \}$ be the collection of spaces of elements in S. For each i, let*

$$S_i \coloneqq \{ \boldsymbol{x} : \boldsymbol{x} \in S \wedge \mathcal{S}\boldsymbol{x} = U_i \}. \tag{3.39}$$

Then $\mathcal{D}S = \{ S_i \}_i$.

In `isl`, this operation is called `isl_union_set_foreach_set`. This function takes a set (an `isl_union_set`) and a callback that is called for each set in the space decomposition of the input set. Each such set is represented by an `isl_set`. In contrast to an `isl_union_set`, all elements of an `isl_set` have the same space. This space is then also the space of the `isl_set`. In fact, each `isl_set`, even an empty one, has its own predetermined space.

Example 3.83. *The following transcript shows an example of splitting up a set along the spaces of its elements.*
python input (sets_in_union_set.py):

```
import isl

def print_set(set):
    print(set)

s = isl.union_set("{ B[6]; A[2,8,1]; B[5] }")
s.foreach_set(print_set)
```

python invocation:

```
python < sets_in_union_set.py
```

python output:

```
{ A[2, 8, 1] }
{ B[6]; B[5] }
```

Operation 3.84 (Space Decomposition of a Binary Relation)**.** *Given a binary relation R, its* space decomposition $\mathcal{D}R$ *is the unique collection of binary relations R_i such that the union of the R_i is equal to R, all pairs of elements in a given R_i have the same pair of spaces and no two pairs of elements from distinct R_i have the same pair of spaces. That is, let $\{ U_i \to V_i \}_i := \{ U \to V : \exists \boldsymbol{x} \to \boldsymbol{y} \in R : U = \mathcal{S}\boldsymbol{x} \wedge V = \mathcal{S}\boldsymbol{y} \}$ be the collection of pairs of spaces of pairs of elements in R. For each i, let*

$$R_i := \{ \boldsymbol{x} \to \boldsymbol{y} : \boldsymbol{x} \to \boldsymbol{y} \in R \wedge \mathcal{S}\boldsymbol{x} = U_i \wedge \mathcal{S}\boldsymbol{y} = V_i \}. \tag{3.40}$$

Then $\mathcal{D}R = \{ R_i \}_i$.

In `isl`, this operation is called `isl_union_map_foreach_map`. Similarly to `isl_union_set_foreach_set` above, this function calls a callback on each binary relation in the space decomposition, where each such binary relation is represented by an `isl_map`. As in the case of an `isl_set`, all pairs of elements in an `isl_map` have the same pair of spaces. These two spaces are called the *domain space* and the *range space* of the `isl_map`.

Lexicographic optimization can now be defined in terms of the space decomposition.

Operation 3.85 (Lexiographic Maximum of a Set)**.** *The lexicographic maximum* $\operatorname{lexmax} S$ *of a set S is a subset of S that contains the lexicographically maximal element of each of the spaces with elements in S. If there is any such space with no lexicographically maximal element, then the operation is undefined. That is, let $\mathcal{D}S =: \{ S_i \}_i$. Define*

$$M_i := \{ \boldsymbol{x} : \boldsymbol{x} \in S_i \wedge \forall \boldsymbol{y} \in S_i : \mathcal{V}\boldsymbol{x} \succcurlyeq \mathcal{V}\boldsymbol{y} \}. \tag{3.41}$$

Then

$$\operatorname{lexmax} S = \bigcup_i M_i. \tag{3.42}$$

In `isl`, this operation is called `isl_union_set_lexmax`. In `iscc`, this operation is written `lexmax`.

Example 3.86. *`iscc` input (`lexmax.iscc`):*

```
S := { B[6]; A[2,8,1]; B[5] };
lexmax S;
```

`iscc` invocation:

```
iscc < lexmax.iscc
```

`iscc` output:

```
{ B[6]; A[2, 8, 1] }
```

Note that if the description of the set involves constant symbols, then the lexicographic maximum may be different for each value of the constant symbols.

Example 3.87. *iscc input (lexmax2.iscc):*

```
S := [n] -> { A[i,j] : i,j >= 0 and i + j <= n };
lexmax S;
```

iscc invocation:

```
iscc < lexmax2.iscc
```

iscc output:

```
[n] -> { A[i = n, j = 0] : n >= 0 }
```

Example 3.88. *The following set has no lexicographically maximal element:*

$$\{ \mathrm{S}[i] : i \geq 0 \}. \tag{3.43}$$

Operation 3.89 (Lexiographic Minimum of a Set)**.** *The lexicographic minimum* lexmin S *of a set S is a subset of S that contains the lexicographically minimal element of each of the spaces with elements in S. If there is any such space with no lexicographically minimal element, then the operation is undefined. That is, let $\mathcal{D}S =: \{ S_i \}_i$. Define*

$$M_i \coloneqq \{ \boldsymbol{x} : \boldsymbol{x} \in S_i \wedge \forall \boldsymbol{y} \in S_i : \mathcal{V}\boldsymbol{x} \preccurlyeq \mathcal{V}\boldsymbol{y} \}. \tag{3.44}$$

Then

$$\operatorname{lexmin} S = \bigcup_i M_i. \tag{3.45}$$

In `isl`, this operation is called `isl_union_set_lexmin`. In `iscc`, this operation is written `lexmin`.

Example 3.90. *iscc input (lexmin.iscc):*

```
S := { B[6]; A[2,8,1]; B[5] };
lexmin S;
```

iscc invocation:

```
iscc < lexmin.iscc
```

iscc output:

```
{ B[5]; A[2, 8, 1] }
```

Operation 3.91 (Lexiographic Maximum of a Binary Relation)**.** *The lexicographic maximum* lexmax R *of a binary relation R is a subset of R that contains for each first element in the pairs of elements in R and for each of the spaces of the corresponding second elements, the lexicographically maximal of those corresponding elements. If there is any such first element and space with no* *corresponding lexicographically maximal second element, then the operation is undefined. That is, let* $\mathcal{D}R =: \{ R_i \}_i$*. Define*

$$M_i := \{ \boldsymbol{x} \to \boldsymbol{y} : \boldsymbol{x} \to \boldsymbol{y} \in R_i \wedge \forall \boldsymbol{x}' \to \boldsymbol{z} \in R_i : \boldsymbol{x} = \boldsymbol{x}' \implies \mathcal{V}\boldsymbol{y} \succcurlyeq \mathcal{V}\boldsymbol{z} \}. \tag{3.46}$$

Then

$$\operatorname{lexmax} R = \bigcup_i M_i. \tag{3.47}$$

In `isl`, this operation is called `isl_union_map_lexmax`. In `iscc`, this operation is written `lexmax`.

Example 3.92. `iscc` *input* (`map_lexmax.iscc`):

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
lexmax R;
```

`iscc` *invocation:*

```
iscc < map_lexmax.iscc
```

`iscc` *output:*

```
{ A[2, 8, 1] -> B[6]; B[5] -> B[5] }
```

Operation 3.93 (Lexiographic Minimum of a Binary Relation)**.** *The lexicographic minimum* lexmin R *of a binary relation* R *is a subset of* R *that contains for each first element in the pairs of elements in* R *and for each of the spaces of the corresponding second elements, the lexicographically minimal of those corresponding elements. If there is any such first element and space with no corresponding lexicographically minimal second element, then the operation is undefined. That is, let* $\mathcal{D}R =: \{ R_i \}_i$*. Define*

$$M_i := \{ \boldsymbol{x} \to \boldsymbol{y} : \boldsymbol{x} \to \boldsymbol{y} \in R_i \wedge \forall \boldsymbol{x}' \to \boldsymbol{z} \in R_i : \boldsymbol{x} = \boldsymbol{x}' \implies \mathcal{V}\boldsymbol{y} \succcurlyeq \mathcal{V}\boldsymbol{z} \}. \tag{3.48}$$

Then

$$\operatorname{lexmin} R = \bigcup_i M_i. \tag{3.49}$$

In `isl`, this operation is called `isl_union_map_lexmin`. In `iscc`, this operation is written `lexmin`.

Example 3.94. `iscc` *input* (`map_lexmin.iscc`):

```
R := { A[2,8,1] -> B[5]; A[2,8,1] -> B[6]; B[5] -> B[5] };
lexmin R;
```

`iscc` *invocation:*

```
iscc < map_lexmin.iscc
```

`iscc` *output:*

```
{ A[2, 8, 1] -> B[5]; B[5] -> B[5] }
```

3.7 Simplification and Quantifier Elimination

For any given set, there are infinitely many ways of describing it using Presburger formulas. All operations on sets and binary relations described so far are defined in terms of the actual sets and/or binary relations and not in terms of the formulas used to describe them. The way a set or binary relation is described is therefore not all that important. Still, it may be instructive to see how `isl` describes sets and binary relations internally since it affects the way they are printed.

Most importantly, sets and binary relations are represented internally in *disjunctive normal form*, meaning that all disjunctions are moved to the outermost positions in the formula, while all conjunctions are moved innermost. In the end, each formula is of the form

$$\bigvee_i \left(\exists \boldsymbol{\alpha}_i : \left(\bigwedge_j t_{i,j}(\boldsymbol{x}, \boldsymbol{\alpha}_i) = 0 \wedge \bigwedge_k u_{i,k}(\boldsymbol{x}, \boldsymbol{\alpha}_i) \geq 0 \right) \right), \tag{3.50}$$

where $t_{i,j}$ and $u_{i,k}$ are Presburger terms and $\boldsymbol{x}$ represents the set variables.

A second transformation that is sometimes performed by `isl` on the internal representation is that of removing all existentially quantified variables. This process is called quantifier elimination and may introduce additional integer divisions and may also increase the number of disjuncts in the internal representation. Note 3.9

Operation 3.95 (Quantifier Elimination)**.** *Quantifier elimination takes a Presburger formula that may involve existentially quantified variables and rewrites it into an equivalent formula that does not involve any quantified variables.*

After quantifier elimination, each formula in a set description is of the form

$$\bigvee_i \left(\bigwedge_j t_{i,j}(\boldsymbol{x}) = 0 \wedge \bigwedge_k u_{i,k}(\boldsymbol{x}) \geq 0 \right), \tag{3.51}$$

where $t_{i,j}$ and $u_{i,k}$ are Presburger terms and $\boldsymbol{x}$ represents the set variables. When applied to sets or binary relations, quantifier elimination means that quantifier elimination is applied to the formulas in their internal representations. In `isl`, these operations are called `isl_union_set_compute_divs` and `isl_union_map_compute_divs`. There is usually no need for a user to call these functions explicitly, as `isl` will apply quantifier elimination when needed. In particular, some operations are more easily performed if the representations of the inputs do not involve any existentially quantified variables.

Example 3.96. *python input (elimination.py):*

```
import isl

s = isl.union_set("{ A[x] : exists a : x < 3a < 2x }")
```

```
print(s)
s.compute_divs()
print(s)
```

python invocation:

```
python < elimination.py
```

python output:

```
{ A[x] : exists (e0: x < 3e0 < 2x) }
{ A[x] : x >= 2 and 3*floor((-1 - x)/3) > -2x }
```

After the application of several operations, the resulting description of a set or binary relation may involve more disjuncts than strictly necessary. The following operation can be used to try and reduce this number. A description with fewer disjuncts is usually easier to understand and typically also results in faster computations.

Note 3.10

Operation 3.97 (Coalescing). *Coalescing takes a formula in disjunctive normal form and rewrites it using fewer or the same number of disjuncts.*

In `isl`, this operation is called `isl_union_set_coalesce` for sets and `isl_union_map_coalesce` for binary relations. In `iscc`, this operation is written `coalesce`.

Example 3.98. *iscc input (coalescing.iscc):*

```
S := { B[i] : 5 <= i <= 6 or 7 <= i <= 10 };
print S;
S := coalesce S;
print S;
```

iscc invocation:

```
iscc < coalescing.iscc
```

iscc output:

```
{ B[i] : 5 <= i <= 10 and (i <= 6 or i >= 7) }
{ B[i] : 5 <= i <= 10 }
```

3.8 Sampling and Scanning

When a set is described intensionally, it may not always be obvious to see whether the set is empty. Operation 3.31 on page 49 can be used to check emptiness, but in some cases it can be useful to obtain an explicit description of an element in the set. In such cases, the following operation can be used.

Operation 3.99 (Sampling a Set)**.** *Sampling of a non-empty set S determines a sequence of values for the constant symbols for which S is effectively non-empty and returns a singleton set S' that is a subset of S for those values of the constant symbols. On an empty set, sampling returns the same empty set.*

In `isl`, this operation is called `isl_union_set_sample_point`. In `iscc`, this operation is written `sample`. The function `isl_union_set_sample_point` returns an object of type `isl_point`, which is a subclass of `isl_set`. Each object of type `isl_point` is either *void* (corresponding to an empty set) or it contains a single element for a specific value of the constant symbols.

Example 3.100. *iscc input (sample.iscc):*

```
sample [n] -> { A[x,y] : 0 < x < y < n };
```

iscc invocation:

```
iscc < sample.iscc
```

iscc output:

```
[n] -> { A[x = 1, y = 2] : n = 3 }
```

Example 3.101. *python input (sample.py):*

```
import isl

s = isl.union_set("[n] -> { A[x,y] : 0 < x < y < n }")
print(s.sample_point())
```

python invocation:

```
python < sample.py
```

python output:

```
[n = 3] -> { A[1, 2] }
```

An explicit description of *all* elements in a set can be obtained using the following operation. This of course assumes that there are a finite number of them for all values of the constant symbols together.

Operation 3.102 (Scanning a Set)**.** *Given a set S that is non-empty for a finite number of values of the constant symbols and that moreover has a finite number of elements for each of those values of the constant symbols, scanning the set returns an explicit description of all these elements for all these values of the constant symbols.*

In `isl`, this operation is called `isl_union_set_foreach_point`. In `iscc`, the `scan` operation prints an explicit representation of the entire set. The function `isl_union_set_foreach_point` takes a callback that is called for each value of the constant symbols and each element in the set. Note that this operation depends on the constant symbols that the set is "aware of". That is, the sets `{ [x] : 0 <= x <= 10 }` and `[n] -> { [x] : 0 <= x <= 10 }` are equal to each other, but only the first description can be scanned since there are an infinite number of values for the constant symbol `n` for which the set in the second description is non-empty.

Example 3.103. *`iscc` input (`scan.iscc`):*

```
scan { A[x] : exists a : x < 3a < 2x < 20 };
```

`iscc` invocation:

```
iscc < scan.iscc
```

`iscc` output:

```
{ A[x = 9]; A[x = 8]; A[x = 7]; A[x = 6]; A[x = 5]; A[x =
    ↪ 4]; A[x = 2] }
```

Example 3.104. *`python` input (`scan.py`):*

```
import isl

def print_point(point):
    print(point)

s = isl.union_set("{ A[x] : exists a : x < 3a < 2x < 20 }")
s.foreach_point(print_point)
```

`python` invocation:

```
python < scan.py
```

`python` output:

```
{ A[9] }
{ A[6] }
{ A[7] }
{ A[4] }
{ A[8] }
{ A[5] }
{ A[2] }
```

3.9 Beyond Presburger Formulas

The main reason for only allowing Presburger formulas in set descriptions is that Presburger formulas are decidable. That is, for any closed Presburger formula (without constant symbols), it is possible to decide whether the formula is satisfied or not. This means in particular that is possible to check whether Note 3.11

a Presburger set is empty or not by existentially quantifying the set variables in the formulas in the set description. In this process, constant symbols can be treated as variables and existentially quantified as well. A set is then considered empty if it is empty for every possible value of the constant symbols as in Operation 3.31 on page 49. As a result of this decidability, all operations described so far can be performed exactly.

When the language that is used to describe the elements belonging to a set is extended beyond the Presburger language, this typically leads to the loss of decidability, meaning that some operations can no longer be performed exactly. Some commonly considered extensions are Note 3.12

multiplication Allowing multiplication essentially means that the constraints may involve polynomials. Some techniques for dealing with polynomials Note 3.13 consider special cases and exploit the fact that the sets only contain integers. Other techniques consider the problem over the rationals or even the reals, meaning that a set may be considered to be non-empty even though it does not contain any integer values.

uninterpreted function symbols Constant symbols, i.e., function symbols of arity zero, are already part of the Presburger language of Definition 3.12 on page 43. Some extensions allow function symbols of any Note 3.14 arity, although typically with some limitations. Just like the special case of constant symbols, these function symbols are uninterpreted because they do not have a predefined interpretation.

Notes

3.1. Note that this definition is not minimal, in the sense that some types of formulas could be defined in terms of others.

3.2. This set of function and predicate symbols is not minimal. The function $-$ and the integer constants other than 0 and 1 can be defined in terms of the other symbols.

3.3. It is more customary to introduce congruence predicates, as is done by, e.g., Presburger (1929), instead of these integer division functions. Both can be defined in terms of the others.

3.4. Traditional Presburger formulas do not have such constant symbols. Both Feautrier (1988b) and Pugh (1991a) use such constant symbols in the context of integer linear programming. Feautrier (1988b) calls them *parameters*. Pugh and Wonnacott (1995) use them in the context of Presburger formulas.

3.5. Note that the term "Presburger formula" is sometimes used in a more restrictive meaning. For example, Leservot (1996) does not allow any quantifiers in his Presburger formulas.

3.6. Some authors consider Presburger arithmetic over the natural numbers rather than the set of all integers. Note, however, that Presburger (1929) uses integers.

3.7. The convention of allowing expressions in tuples that depend on variables in earlier positions is also inherited from the `Omega` library. An alternative would be to equate all expressions that appear in a tuple to a fresh sequence of set variables and to consider all variables that appear in the original expressions as implicitly existentially quantified. An expression of the form $\{\, \mathrm{S}[j-1] \to \mathrm{S}[j] \,\}$ would then be allowed as it would be interpreted as

$$\{\, \mathrm{S}[v_1] \to \mathrm{S}[v_2] : \exists j : v_1 = j - 1 \wedge v_1 = j \,\}. \tag{3.52}$$

3.8. The formula in (3.26) is technically speaking not itself a Presburger formula, but for each value of n it can be expanded into a Presburger formula.
3.9. The need for quantifier elimination is one of the main reasons for introducing integer division symbols in the Presburger language. In `isl`, quantifier elimination is performed by applying parametric integer programming (Feautrier 1988b) on the existentially quantified variables, i.e., by finding the minimal value of those variables in terms of the other variables and constant symbols. The "new parameters" (Feautrier 1988b) in the result correspond to the additional integer divisions in the result of the quantifier elimination. Note that it is more customary to perform quantifier elimination by introducing congruence predicates. See also Note 3.3.
3.10. Verdoolaege (2015a) describes how coalescing is implemented in `isl`.
3.11. While it is always possible to determine the truth value of such a Presburger formula, the worst-case complexity is fairly high, see Oppen (1978).
3.12. Another extension is formed by encodings in finite automata. In particular, an encoding where the automaton accepts the "digits" of an integer in a base r can also represent the function V_r mapping a non-zero integer z to the greatest power of r dividing z (Bruyère 1985). See, e.g., Boigelot (1999, Section 8.1.4) for an overview. Finite automata representations may not be very practical for polyhedral compilation due to the difficulty of extracting a constraint representation from the automaton (Latour 2004).
3.13. Within the context of polyhedral compilation, several techniques have been considered to handle some forms of polynomial constraints. This note lists some of them.

Maslov and Pugh (1994) describe how to replace some forms of quadratic constraints over integers by a disjunction of conjunctions of linear constraints. Combined with factorization and projection of variables not involved in non-linear terms, they use this technique to check whether a polynomial constraint conflicts with other constraints and, if not, to simplify the polynomial constraint. This technique specifically exploits the fact that the quadratic constraint is defined over integers. The main motivation of the authors is dependence analysis.

Clauss and Tchoupaeva (2004) consider the problem of computing lower and/or upper bounds of a polynomial f defined over a (rational) parametric box. The resulting bounds are expressed as the minimum and maximum of a collection of polynomials in the constant symbols. If, say, these latter polynomials can be proven to all be negative (possibly through a recursive application), then the constraint $f \geq 0$ is known to conflict with the constraints of the parametric box. The parametric box itself has a lower and upper bound on each dimension, where these lower and upper bounds may involve the constant symbols and the previous dimensions. The bounds may themselves be polynomials, but the technique requires that the difference between an upper and a lower bound is never zero, which in turn requires the determination of the roots of this difference.

Clauss, Fernandez, et al. (2009) describe a related technique for computing lower and/or upper bounds of a polynomial defined over a convex polytope. The resulting bounds are again expressed as the minimum and maximum of a collection of polynomials in the constant symbols.

Größlinger (2009) mainly describes two techniques for handling polynomial constraints, one where multiplication is only allowed with a single symbolic constant and one where general multiplication is allowed. For the first case, the author presents a method for solving an equality constraint exactly, which is then used for dependence analysis. In the second case, the author resorts to cylindrical algebraic decomposition techniques (Arnon et al. 1984), solving over the reals.

Feautrier (2015) explores how to show that a (template) polynomial is non-negative over a set bounded by polynomial inequalities in the reals, by writing the template polynomial as a product of a fixed number M of polynomials from the inequalities. If the linear inequalities among the polynomial inequalities define a polytope, then this process is guaranteed to find a solution for some M. The author presents applications in dependence analysis and the computation of polynomial schedules.

3.14. There is some support in the `Omega` library for uninterpreted function symbols, but this support is fairly limited. In particular, the arguments of the uninterpreted function symbols are required to form a prefix of the set variables. This means that any given uninterpreted function symbol can only appear with a single set of arguments within a given set description, allowing each uninterpreted function symbol (together with its arguments) to be handled in the same way as a constant symbol. Unfortunately, this restriction is so severe, that it cannot be maintained across all operations on sets and relations. Instead, any constraint containing an uninterpreted function symbol that would no longer satisfy the requirements is replaced by an “UNKNOWN” constraint.

The Sparse Polyhedral Framework of Strout et al. (2012) follows a completely different approach. Uninterpreted function symbols are allowed to have arbitrary arguments in this framework, but very few operations are supported on sets and relations. Essentially, existential quantification is not allowed and only those operations are supported that can be implemented in a way that avoids existential quantification.

An earlier version of the Sparse Polyhedral Framework, described by LaMielle and Strout (2010), does allow existential quantification and describes techniques for exploiting information about the uninterpreted function symbols, e.g., their bijectivity, to eliminate existentially quantified variables. After application of the simplification techniques, no existentially quantified variables are allowed in the arguments of the function symbols. Otherwise, the approach fails. The remaining existentially quantified variables are eliminated using Fourier-Motkzin elimination (Schrijver 1986), meaning that the elements in the set are treated as rational values rather than as integers.

Chapter 4

Piecewise Quasi-Affine Expressions

While a Presburger relation is perfectly capable of representing a function as a special case, it can sometimes be more convenient to deal with an explicit representation of a function. The piecewise quasi-affine expressions of this chapter can represent the same functions as those that can be represented as a Presburger relation.

4.1 Quasi-Affine Expressions

Definition 4.1 (Quasi-Affine Expression)**.** *A* quasi-affine expression f *is a function that maps a named integer tuple with a given space S to a rational value, where the function is specified as a Presburger term in the variables of the tuple, optionally divided by an integer constant. The space S is called the domain space of f and is written $\mathcal{S}^{\mathrm{dom}} f$. The domain of a quasi-affine expression is the set of all elements with space S. As a special case, the quasi-affine expression may also be a (symbolic) constant expression, in which case there is no domain space, written $\mathcal{S}^{\mathrm{dom}} f = \perp$, and the domain is a unit set.*

Such an expression is called *quasi*-affine because it may involve integer divisions. In `isl`, a quasi-affine expression is represented by an `isl_aff`. The domain space of an `isl_aff` is the space of the input integer tuple. The range space of an `isl_aff` is fixed to the anonymous single-dimensional space. In case the quasi-affine expression is a constant expression, there is no input tuple and the space of the `isl_aff` is the anonymous single-dimensional space. Note 4.1

Notation 4.2 (Quasi-Affine Expression)**.** *In `isl`, a quasi-affine expression is written as a structured named integer tuple template, followed by `->` and the quasi-affine expression in the variables of the structured named integer tuple template enclosed in brackets, with the entire expression enclosed in braces. The structured named integer tuple template and the `->` are omitted if the quasi-affine expression does not have a domain space. If any constant symbols are involved, then they need to be declared as in Notation 3.22 on page 46.*

As in the case of sets and binary relation, the way a quasi-affine expression is printed may be different from the way it was originally described by the user.

Example 4.3. *python input (affine.py):*

```
import isl

a = isl.aff("{ [x,y] -> [floor((2*x+4*floor(y/2))/3)] }")
print(a)
```

python invocation:

```
python < affine.py
```

python output:

```
{ [x, y] -> [(x + floor((y)/2) + floor((-2x + y)/6))] }
```

Clearly, pure quasi-affine expressions are not enough to represent any possible single-valued Presburger relation. Instead, several such expressions need to be combined in a structured way. In particular, the following type constructors are applied to the type of quasi-affine expressions to obtain more expressive types: a tuple constructor to combine several expressions into a tuple; a piecewise constructor to combine several expressions defined over disjoint parts of a space; and a union constructor to combine several expressions defined over different spaces.

Definition 4.4 (Tuple of Expressions)**.** *A* tuple of expressions *combines zero or more base expressions of the same type and with the same domain space (or no domain space) into a multi-dimensional expression that shares this domain space and that has a prescribed range space. In particular, it is either,*

- *an identifier n along with $d \geq 0$ base expressions e_j for $0 \leq j < d$, written $n[e_0, e_1, \ldots, e_{d-1}]$, or,*
- *an identifier n along with two tuples of expressions $\boldsymbol{e}$ and $\boldsymbol{f}$ written $n[\boldsymbol{e} \to \boldsymbol{f}]$.*

The domain of a tuple of one or more expressions is the intersection *of the domains of the expressions. The domain of a tuple of zero expressions is undefined.*

In particular, a tuple of quasi-affine expressions is the result of constructing a tuple expression from quasi-affine expressions.

Definition 4.5 (Tuple of Quasi-Affine Expressions)**.** *A* tuple of quasi-affine expressions *is the result of applying Definition 4.4 to quasi-affine expressions.*

In `isl`, a tuple of quasi-affine expressions is represented by an `isl_multi_aff`.

Notation 4.6 (Tuple of Quasi-Affine Expressions)**.** *In* `isl`*, a tuple of quasi-affine expressions is written in the same way as a quasi-affine expression in Notation 4.2 on page 71, except that the quasi-affine expression enclosed in brackets is generalized to a structured named integer tuple template with the variables replaced by quasi-affine expressions in the variable of the input structured named integer tuple template.*

Example 4.7. *Here is an example of a tuple of quasi-affine expressions:*

$$\{\, \mathrm{S}[i,j] \to \mathrm{A\_x}[\mathrm{A}[i+1,j-1] \to \mathrm{x}[]] \,\}. \tag{4.1}$$

Definition 4.8 (Space of a Tuple of Expressions)**.** *The* space $\mathcal{S}\boldsymbol{e}$ *of a tuple of expressions* $\boldsymbol{e}$ *is*

- n/d*, if* $\boldsymbol{e}$ *is of the form* $n[e_0, e_1, \ldots, e_{d-1}]$*, with* n *an identifier and* d *a non-negative integer, or,*
- $(n, \mathcal{S}(\boldsymbol{f}), \mathcal{S}(\boldsymbol{g}))$*, if* $\boldsymbol{e}$ *is of the form* $n[\boldsymbol{f} \to \boldsymbol{g}]$*, with* n *an identifier and* $\boldsymbol{f}$ *and* $\boldsymbol{g}$ *tuples of expressions.*

In `isl` the space of a tuple of quasi-affine expressions is called the *range space* of its `isl_multi_aff` representation.

Definition 4.9 (Expression Vector)**.** *The* expression vector $\mathcal{E}\boldsymbol{e}$ *of a tuple of expressions* $\boldsymbol{e}$ *is the vector*

- $(e_0, e_1, \ldots, e_{d-1})$*, if* $\boldsymbol{e}$ *is of the form* $n[e_0, e_1, \ldots, e_{d-1}]$*, with* n *an identifier and* d *a non-negative integer, or,*
- $\mathcal{E}(\boldsymbol{f}) \| \mathcal{E}(\boldsymbol{g})$*, with* $\|$ *the concatenation of two vectors, if* $\boldsymbol{e}$ *is of the form* $n[\boldsymbol{f} \to \boldsymbol{g}]$*, with* n *an identifier and* $\boldsymbol{f}$ *and* $\boldsymbol{g}$ *tuples of expressions.*

Definition 4.10 (Piecewise Expression)**.** *A* piecewise expression *combines* $n \geq 0$ *pairs of fixed-space sets* S_i *and base expressions* E_i *into a single expression. The spaces of the* S_i *and the domain spaces of the* E_i *all need to be the same. Similarly, the range spaces of the* E_i *also need to be the same. Furthermore, the* S_i *need to be pairwise disjoint. The domain and range spaces of the piecewise expression are the same as those of the* E_i*. The domain of the piecewise expression is the union of the* S_i*. The value of the piecewise expression at an integer tuple* $\boldsymbol{x}$ *is* $E_i(\boldsymbol{x})$ *if* $\boldsymbol{x} \in S_i$ *for some* i*. Otherwise, the value is undefined.*

This constructor can again be applied to produce new types.

Definition 4.11 (Piecewise Quasi-Affine Expression)**.** *A* piecewise quasi-affine expression *is the result of applying Definition 4.10 to quasi-affine expressions.*

In `isl`, a piecewise quasi-affine expression is represented by an `isl_pw_aff`.

Notation 4.12 (Piecewise Quasi-Affine Expression)**.** *In* `isl`*, a piecewise quasi-affine expression is written as a sequence of conditional quasi-affine expressions separated by a semicolon and enclosed in braces. Each conditional quasi-affine expression consists of the notation from Notation 4.2 on page 71 (without the braces) for the quasi-affine expression E_i, followed by a colon and the constraints of S_i.*

Example 4.13. *Here is an example of a piecewise quasi-affine expression:*

$$\{\, [x] \to [x+1] : 0 \le x < n-1; [x] \to [0] : x = n-1 \,\}. \tag{4.2}$$

Definition 4.14 (Piecewise Tuple of Quasi-Affine Expressions)**.** *A* piecewise tuple of quasi-affine expression *is the result of applying Definition 4.10 on the previous page to tuples of quasi-affine expressions.*

In `isl`, a piecewise tuple of quasi-affine expressions is represented by an `isl_pw_multi_aff`.

Note 4.2

Alternative 4.15 (Quasts)**.** *Quasi-affine selection trees, also known as quasts, form an alternative representation of piecewise tuples of quasi-affine expressions. A quasi-affine section tree is a tree with as leaves either a tuple of quasi-affine expressions or $\perp$ and as internal nodes a quasi-affine expression. Each internal node has two children, one that should be followed if the quasi-affine expression in the node is non-negative and one that should be followed if the expression is negative. The value of the quasi-affine section tree is the value of the tuple of quasi-affine expressions that is reached on evaluating the internal nodes on the input. If this process ends up in $\perp$, then the value is undefined.*

Notation 4.16 (Piecewise Tuple of Quasi-Affine Expressions)**.** *In* `isl`*, a piecewise tuple of quasi-affine expressions is written as a sequence of conditional tuples of quasi-affine expressions separated by a semicolon and enclosed in braces. Each conditional tuple of quasi-affine expressions consists of the notation from Notation 4.6 on the preceding page (without the braces) for the tuple of quasi-affine expressions E_i, followed by a colon and the constraints of S_i.*

Each piecewise expression has a given domain and range space. The following constructor allows them to be combined over multiple spaces.

Definition 4.17 (Multi-Space Expression)**.** *A* multi-space expression *combines piecewise expressions with different domain and/or range spaces, but with pair-wise disjoint domains into a single expression. A multi-space expression does not have a specific domain or range space, even if all constituent piecewise expressions happen to have the same domain or range space. The domain of a multi-space expression is the union of the domains of the combined piecewise expressions. The value of a multi-space expression at an integer tuple $\boldsymbol{x}$ is the value of the piecewise expression at $\boldsymbol{x}$ that contains $\boldsymbol{x}$ in its domain, if any.*

Definition 4.18 (Multi-Space Piecewise Quasi-Affine Expression)**.** *A* multi-space piecewise quasi-affine expression *is the result of applying Definition 4.17 on the preceding page to piecewise quasi-affine expressions.*

In `isl`, a multi-space piecewise quasi-affine expression is represented by an `isl_union_pw_aff`.

Notation 4.19 (Multi-Space Piecewise Quasi-Affine Expression)**.** *A multi-space piecewise quasi-affine expression is written in the same way as a piecewise quasi-affine expression (see Notation 4.12 on the facing page). The only difference is that the domains may have different spaces.*

Definition 4.20 (Multi-Space Piecewise Tuple of Quasi-Affine Expressions)**.** *A* multi-space piecewise tuple of quasi-affine expressions *is the result of applying Definition 4.17 on the preceding page to piecewise tuples of quasi-affine expressions.*

In `isl`, a multi-space piecewise tuple of quasi-affine expressions is represented by an `isl_union_pw_multi_aff`.

Notation 4.21 (Multi-Space Piecewise Tuple of Quasi-Affine Expressions)**.** *A multi-space piecewise tuple of quasi-affine expressions is written in the same way as a piecewise tuple of quasi-affine expressions (see Notation 4.16 on the facing page). The only difference is that the domains and the tuples may have different spaces.*

Tuples can also be constructed from piecewise expressions or multi-space expressions.

Definition 4.22 (Tuple of Piecewise Quasi-Affine Expressions)**.** *A* tuple of piecewise quasi-affine expressions *is the result of applying Definition 4.4 on page 72 to piecewise quasi-affine expressions.*

In `isl`, a tuple of piecewise quasi-affine expressions is represented by an `isl_multi_pw_aff`.

Notation 4.23 (Tuple of Piecewise Quasi-Affine Expressions)**.** *In `isl`, a tuple of piecewise quasi-affine expressions is written in the same way as a tuple of quasi-affine expressions in Notation 4.6 on page 73, except that each quasi-affine expression may be replaced by a semicolon separated sequence of a quasi-affine expression, a colon and the constraints of the corresponding set. Each such sequence needs to be enclosed in parentheses to prevent the comma that is used to separate the sequences from being considered as separating expressions inside the constraints.*

Although piecewise tuples of quasi-affine expressions and tuples of piecewise quasi-affine expressions are very similar, they are handled in slightly different ways since the first is a piecewise expression, while the second is a tuple. There is also a difference in what these two types of expressions can represent. In particular, in a tuple of piecewise quasi-affine expressions, each element is a piecewise expression that may be undefined in different parts of the domain space. In a piecewise tuple of quasi-affine expressions, on the other hand, the entire tuple is either defined or undefined at any particular point in the domain space.

Example 4.24. *The following tuple of piecewise quasi-affine expressions cannot be represented as a piecewise tuple of quasi-affine expressions:*

$$\{\, [i] \to [(i : i \geq 0), (i - 1 : i \geq 1)] \,\}. \tag{4.3}$$

In particular, the first piecewise quasi-affine expression has domain $\{\, [i] : i \geq 0 \,\}$*, while the second has domain* $\{\, [i] : i \geq 1 \,\}$*.*

Example 4.25. *The following transcript converts a piecewise tuple of quasi-affine expressions to a tuple of piecewise quasi-affine expressions, which can always be performed without loss of information.*
python input (multi_pw_aff.py):

```
import isl

a = isl.pw_multi_aff("{ [i] -> [i, i - 1] : i - 1 >= 0 }")
print(a)
a = isl.multi_pw_aff(a)
print(a)
```

python invocation:

```
python < multi_pw_aff.py
```

python output:

```
{ [i] -> [(i), (-1 + i)] : i > 0 }
{ [i] -> [((i) : i > 0), ((-1 + i) : i > 0)] }
```

Definition 4.26 (Tuple of Multi-Space Piecewise Quasi-Affine Expressions)**.** *A* tuple of multi-space piecewise quasi-affine expressions *is the result of applying Definition 4.4 on page 72 to multi-space piecewise quasi-affine expressions. Since a multi-space piecewise quasi-affine expression does not have a domain space, neither does a tuple of multi-space piecewise quasi-affine expressions.*

In `isl`, a tuple of multi-space piecewise quasi-affine expressions is represented by an `isl_multi_union_pw_aff`.

Notation 4.27 (Tuple of Multi-Space Piecewise Quasi-Affine Expressions)**.** *In* `isl`*, a tuple of multi-space piecewise quasi-affine expressions is written as a structured named integer tuple template with the variables replaced by multi-space piecewise quasi-affine expressions in the notation of Notation 4.21 on the preceding page, except that the constant symbols are declared outside of the structured named integer tuple template.*

Example 4.28. *The following transcript shows an example of a tuple of multi-space piecewise quasi-affine expressions.*
`python` *input (*`multi_union_pw_aff.py` *):*

```
import isl

a = isl.multi_union_pw_aff(
    "[n] -> A[{ S1[] -> [n]; S2[i,j] -> [i] }, "
    "{ S1[] -> [0]; S2[i,j] -> [j] }]")
print(a)
```

`python` *invocation:*

```
python < multi_union_pw_aff.py
```

`python` *output:*

```
[n] -> A[{ S2[i, j] -> [(i)]; S1[] -> [(n)] }, { S2[i, j] ->
    ↪  [(j)]; S1[] -> [(0)] }]
```

4.2 Creation

Some operations on binary relations are also available in a form that produces a multi-space piecewise tuple of quasi-affine expressions. In particular, the domain projection operation of Operation 2.100 on page 38 is also available in `isl` as `isl_union_map_domain_map_union_pw_multi_aff`.

4.3 Operations

Every binary operation takes either two expressions with a domain space as input or two expressions with no domain space. Some binary operations have additional constraints.

4.3.1 Sum

The sum of two functions is defined in the obvious way.

Definition 4.29 (Sum of Quasi-Affine Expressions)**.** *The sum $f + g$ of two quasi-affine expressions f and g with the same domain space is a function with the same domain space and as value the sum of the values of f and g.*

In `isl`, this operation is available as `isl_aff_add`.

Definition 4.30 (Sum of Tuples of Expressions)**.** *The sum $\boldsymbol{f} + \boldsymbol{g}$ of two tuples of expressions with the same domain and range spaces is a tuple of expressions with the same domain and range spaces and as expressions the pair-wise sums of the expressions of $\boldsymbol{f}$ and $\boldsymbol{g}$.*

In `isl`, this operation is available as

- `isl_multi_aff_add`,
- `isl_multi_pw_aff_add`, and
- `isl_multi_union_pw_aff_add`.

Definition 4.31 (Sum of Piecewise Expressions)**.** *The sum $f+g$ of two piecewise expressions with the same domain and range spaces is a tuple of expressions with the same domain and range spaces that evaluates to the sum of f and g on their shared domain. In particular, let f be composed of m sets U_i with base expressions F_i and f of n sets V_j with base expressions G_j. Define*

$$\begin{aligned} S_{ij} &= U_i \cap V_j \\ E_{ij} &= F_i + G_j, \end{aligned} \tag{4.4}$$

for $1 \le i \le m$ and $1 \le j \le n$. The sum $f+g$ consists of the non-empty sets S_{ij} along with the corresponding base expressions E_{ij}.

In `isl`, this operation is available as

- `isl_pw_aff_add` and
- `isl_pw_multi_aff_add`.

Definition 4.32 (Sum of Multi-Space Expressions)**.** *The sum $f+g$ of two multi-space expressions is a multi-space expression that combines the sums of the pairs of constituent expressions of f and g that have the same domain and range spaces. If f and g have constituent expressions that have the same domain space, but a different range space, then the domains of these two constituent expressions are required to be disjoint.*

In `isl`, this operation is available as

- `isl_union_pw_aff_add` and
- `isl_union_pw_multi_aff_add`.

Example 4.33. *`python` input (`aff_sum.py`):*

```
import isl

a1 = isl.union_pw_multi_aff(
 "{ A[i] -> B[i] : i > 0 }")
a2 = isl.union_pw_multi_aff(
 "{ A[i] -> B[3] : i >= 0; A[i] -> C[2] : i < 0 }")
print(a1.add(a2))
```

`python` invocation:

```
python < aff_sum.py
```

`python` output:

```
{ A[i] -> B[(3 + i)] : i > 0 }
```

4.3.2 Union

The union of two expressions with disjoint domains combines them into a single expression defined over the union of the domains.

Definition 4.34 (Union of Piecewise Expressions)**.** *The union of two piecewise expressions f and g with the same domain and range spaces but disjoint domains is a piecewise expression that evaluates to f on the domain of f and to g on the domain of g.*

Definition 4.35 (Union of Multi-Space Expressions)**.** *The union of two multi-space expressions f and g with disjoint domains is a multi-space expression that evaluates to f on the domain of f and to g on the domain of g.*

The union operation is not directly available in `isl`. Instead, `isl` provides an operation that computes the sum of the intersection of the domains of the two arguments and the union on the symmetric difference of those domains. The two arguments then also need to agree on the range space on the intersection of their domains. If the arguments are known to be disjoint, then this operation can be used as a union operation. It comes in the following flavors:

- `isl_pw_aff_union_add`,
- `isl_pw_multi_aff_union_add`,
- `isl_union_pw_aff_union_add`,
- `isl_union_pw_multi_aff_union_add`, and
- `isl_multi_union_pw_aff_union_add`.

Example 4.36. *python input (aff_union_sum.py):*

```
import isl

a1 = isl.union_pw_multi_aff(
 "{ A[i] -> B[i] : i > 0 }")
a2 = isl.union_pw_multi_aff(
 "{ A[i] -> B[3] : i >= 0; A[i] -> C[2] : i < 0 }")
print(a1.union_add(a2))
```

python invocation:

```
python < aff_union_sum.py
```

python output:

```
{ A[i] -> B[(3 + i)] : i > 0; A[i] -> B[(3)] : i = 0; A[i]
    ↪ -> C[(2)] : i < 0 }
```

4.3.3 Product

The product of two functions (or constant expressions) is defined in an analogous way to the product of two set (see Operation 2.77 on page 31) or two binary relations (see Operation 2.79 on page 31).

Operation 4.37 (Product of Tuple Expressions)**.** *The product $\boldsymbol{f} \times \boldsymbol{g}$ of two tuple expressions $\boldsymbol{f}$ and $\boldsymbol{g}$ is the tuple expression obtained by forming the wrapped pair of the tuple expressions defined over the wrapped pair of domain spaces (or no domain space if $\boldsymbol{f}$ and $\boldsymbol{g}$ have no domain space). That is, let $\mathcal{E}\boldsymbol{f} =: (f_1, \ldots, f_m)$ and $\mathcal{E}\boldsymbol{g} =: (g_1, \ldots, g_n)$. Then*

$$\begin{aligned} \mathcal{S}(\boldsymbol{f} \times \boldsymbol{g}) &= [\mathcal{S}\boldsymbol{f} \to \mathcal{S}\boldsymbol{g}] \\ \mathcal{S}^{\text{dom}}(\boldsymbol{f} \times \boldsymbol{g}) &= [\mathcal{S}^{\text{dom}}\boldsymbol{f} \to \mathcal{S}^{\text{dom}}\boldsymbol{g}] \\ \mathcal{E}(\boldsymbol{f} \times \boldsymbol{g}) &= (f'_1, \ldots, f'_m, g'_1, \ldots, g'_n) \end{aligned} \tag{4.5}$$

with

$$\begin{aligned} f'_i : \mathcal{S}^{\text{dom}}(\boldsymbol{f} \times \boldsymbol{g}) &\to \mathbb{Q} : \\ [\boldsymbol{a} \to \boldsymbol{b}] &\mapsto f_i(\boldsymbol{a}) \qquad \textit{if } \boldsymbol{a} \in \operatorname{dom} f_i \end{aligned} \tag{4.6}$$

for $1 \le i \le m$ and

$$\begin{aligned} g'_j : \mathcal{S}^{\text{dom}}(\boldsymbol{f} \times \boldsymbol{g}) &\to \mathbb{Q} : \\ [\boldsymbol{a} \to \boldsymbol{b}] &\mapsto g_j(\boldsymbol{b}) \qquad \textit{if } \boldsymbol{b} \in \operatorname{dom} g_j \end{aligned} \tag{4.7}$$

for $1 \le j \le n$. If $\mathcal{S}^{\text{dom}}\boldsymbol{f} = \perp$ (and therefore also $\mathcal{S}^{\text{dom}}\boldsymbol{g} = \perp$), then $\mathcal{S}^{\text{dom}}(\boldsymbol{f} \times \boldsymbol{g}) = \perp$ too and $\mathcal{E}(\boldsymbol{f} \times \boldsymbol{g}) = \mathcal{E}\boldsymbol{f} \| \mathcal{E}\boldsymbol{g}$, with $\|$ the concatenation of two vectors.

In `isl`, this operation is available as `isl_multi_aff_product` and also as `isl_multi_pw_aff_product`. The operation is also available on types derived from tuple expressions, in particular as `isl_pw_multi_aff_product`.

Example 4.38. `python` *input (*`aff_product.py`*):*

```python
import isl

a1 = isl.multi_aff("{ A[i,j] -> R[i + j] }")
a2 = isl.multi_aff("{ B[x] -> R[2x] }")
print(a1.product(a2))
```

`python` *invocation:*

```
python < aff_product.py
```

`python` *output:*

```
{ [A[i, j] -> B[x]] -> [R[(i + j)] -> R[(2x)]] }
```

A range product of two functions is also defined in an analogous way to that of binary relations (see Operation 2.86 on page 34). No domain product can be defined on functions since a function defined in terms of one domain cannot be coerced into being defined on another domain.

Operation 4.39 (Range Product of Tuple Expressions)**.** *The range product* $\boldsymbol{f} \ltimes \boldsymbol{g}$ *of two tuple expressions* $\boldsymbol{f}$ *and* $\boldsymbol{g}$ *defined over the same space (or no domain space) is the tuple expression obtained by forming the wrapped pair of the tuple expressions. The range product is defined over the same space as* $\boldsymbol{f}$ *and* $\boldsymbol{g}$*. That is,*

$$\begin{aligned} \mathcal{S}(\boldsymbol{f} \ltimes \boldsymbol{g}) &= [\mathcal{S}\boldsymbol{f} \to \mathcal{S}\boldsymbol{g}] \\ \mathcal{S}^{\mathrm{dom}}(\boldsymbol{f} \ltimes \boldsymbol{g}) &= \mathcal{S}^{\mathrm{dom}}\boldsymbol{f} = \mathcal{S}^{\mathrm{dom}}\boldsymbol{g} \\ \mathcal{E}(\boldsymbol{f} \ltimes \boldsymbol{g}) &= \mathcal{E}\boldsymbol{f} \| \mathcal{E}\boldsymbol{g}, \end{aligned} \tag{4.8}$$

with $\|$ *the concatenation of two vectors.*

Note that for tuples of expressions with no domain space, the product and the range product are the same. In `isl`, this operation is available as

- `isl_multi_aff_range_product`,
- `isl_multi_pw_aff_range_product`, and
- `isl_multi_union_pw_aff_range_product`.

The operation is also available on types derived from tuple expressions, in particular as `isl_pw_multi_aff_range_product`.

Example 4.40. *python input (aff_range_product.py):*

```
import isl

a1 = isl.multi_union_pw_aff(
    "R[{ A[i,j] -> [i + j]; B[x] -> [2x] }]")
a2 = isl.multi_union_pw_aff(
    "S[{ A[i,j] -> [2]; B[x] -> [0]; C[] -> [1] }]")
print(a1.range_product(a2))
```

python invocation:

```
python < aff_range_product.py
```

python output:

```
[R[{ A[i, j] -> [(i + j)]; B[x] -> [(2x)] }] -> S[{ A[i, j]
    ↪ -> [(2)]; B[x] -> [(0)]; C[] -> [(1)] }]]
```

In some cases, it may be more convenient to simply concatenate two tuples of expressions, without keeping track of the original spaces of these tuples. To this end, a flattening of a space is first defined as follows.

Definition 4.41 (Flattening of a Space)**.** *Given a space S, its flattening $\mathcal{F}S$ is the space*

- *n/d, if $S = n/d$ or,*
- *n/d if $S = (n, S_1, S_2)$, $\mathcal{F}S_1 = n_1/d_1$, $\mathcal{F}S_2 = n_2/d_2$ and $d = d_1 + d_2$.*

Operation 4.42 (Flat Range Product of Tuple Expressions)**.** *The flat range product $\boldsymbol{h}$ of two tuple expressions $\boldsymbol{f}$ and $\boldsymbol{g}$ defined over the same space (or no domain space) is the tuple expression obtained by forming the flattened concatenation of the tuple expressions. The flat range product is defined over the same space as $\boldsymbol{f}$ and $\boldsymbol{g}$. That is,*

$$\begin{aligned} \mathcal{S}\boldsymbol{h} &= \mathcal{F}([\mathcal{S}\boldsymbol{f} \to \mathcal{S}\boldsymbol{g}]) \\ \mathcal{S}^{\text{dom}}\boldsymbol{h} &= \mathcal{S}^{\text{dom}}\boldsymbol{f} = \mathcal{S}^{\text{dom}}\boldsymbol{g} \\ \mathcal{E}\boldsymbol{h} &= \mathcal{E}\boldsymbol{f} \| \mathcal{E}\boldsymbol{g}, \end{aligned} \tag{4.9}$$

with $\|$ the concatenation of two vectors.

In `isl`, this operation is available as

- `isl_multi_aff_flat_range_product`,
- `isl_multi_pw_aff_flat_range_product`, and
- `isl_multi_union_pw_aff_flat_range_product`.

The operation is also available on types derived from tuple expressions, in particular as

- `isl_pw_multi_aff_flat_range_product`, and
- `isl_union_pw_multi_aff_flat_range_product`.

Example 4.43. `python` *input (*`aff_flat_range_product.py`*):*

```
import isl

a1 = isl.multi_union_pw_aff(
    "R[{ A[i,j] -> [i + j]; B[x] -> [2x] }]")
a2 = isl.multi_union_pw_aff(
    "S[{ A[i,j] -> [2]; B[x] -> [0]; C[] -> [1] }]")
print(a1.flat_range_product(a2))
```

`python` *invocation:*

```
python < aff_flat_range_product.py
```

`python` *output:*

```
[{ A[i, j] -> [((i + j))]; B[x] -> [((2x))] }, { A[i, j] -> [(2)
   ↪ ]; B[x] -> [(0)]; C[] -> [(1)] }]
```

4.3.4 Pullback

The precomposition of two functions f and g is called the *pullback* of the function f by the function g. In particular, the domain of f is pulled back to the domain of g. Essentially, this means that the function g is plugged into f.

Operation 4.44 (Pullback)**.** *The pullback $f \circ g$ of the function f by the function g is the function that maps domain elements of g to the result of applying f to the result of g. If f has a domain space, then it needs to be equal to the range space of g. That is,*

$$\begin{aligned} f \circ g : \mathcal{S}^{\mathrm{dom}} g \to \mathbb{Q} : & \\ \boldsymbol{a} \mapsto f(g(\boldsymbol{a})) & \quad \text{if } g(\boldsymbol{a}) \in \operatorname{dom} f. \end{aligned} \tag{4.10}$$

If $\mathcal{S}^{\mathrm{dom}} \boldsymbol{g} = \perp$, then this degenerates to $f \circ g = f(g)$ if $g \in \operatorname{dom} f$. If f is a tuple expression, then the pullback of f by g is the tuple in the same space as f with as tuple expressions the pullback of the tuple expressions of f by g.

In `isl`, this operation is available as

- `isl_aff_pullback_multi_aff`
- `isl_pw_aff_pullback_multi_aff`
- `isl_pw_aff_pullback_pw_multi_aff`
- `isl_pw_aff_pullback_multi_pw_aff`
- `isl_multi_aff_pullback_multi_aff`
- `isl_pw_multi_aff_pullback_multi_aff`
- `isl_pw_multi_aff_pullback_pw_multi_aff`
- `isl_union_pw_multi_aff_pullback_union_pw_multi_aff`
- `isl_multi_pw_aff_pullback_multi_aff`
- `isl_multi_pw_aff_pullback_pw_multi_aff`
- `isl_multi_pw_aff_pullback_multi_pw_aff`
- `isl_union_pw_aff_pullback_union_pw_multi_aff`
- `isl_multi_union_pw_aff_pullback_union_pw_multi_aff`

The final parts of these functions names refer to the type of the final argument. In the `python` interface, this part of the function name is dropped. The above functions are therefore all called `pullback`.

Example 4.45. *`python` input (`pullback.py`):*

```
import isl

a1 = isl.multi_union_pw_aff(
    "R[{ A[i,j] -> [i + j]; B[x] -> [2x] }]")
a2 = isl.union_pw_multi_aff(
    "{ C[x] -> A[x,x]; D[x] -> E[2x] }")
print(a1.pullback(a2))
```

python invocation:

```
python < pullback.py
```

python output:

```
R[{ C[x] -> [(2x)] }]
```

4.4 Conversions

A function f of any of the types derived from a quasi-affine expressions can be converted to a Presburger relation as long as the domain of a function is not a unit set. In particular, the resulting relation is of the form

$$\{\, \boldsymbol{a} \to \boldsymbol{b} : \boldsymbol{a} \in \operatorname{dom} f \wedge \boldsymbol{b} = f(\boldsymbol{a}) \,\}. \tag{4.11}$$

This conversion is available as `isl_union_map_from_union_pw_multi_aff` and `isl_union_map_from_multi_union_pw_aff` in `isl`. Recall that the domain of a tuple of multi-space piecewise quasi-affine expressions is the *intersection* of the domains of the constituent expressions, so some information may be lost in the conversion. This means in particular that if these multi-space piecewise quasi-affine expressions have disjoint domains, that then the resulting Presburger relation is empty. In the `python` interface, these functions are called `convert_from` because `from` is a Python keyword.

Example 4.46. *Here is an example of a conversion where no information is lost.*
python input (union_map_from_multi_union_pw_aff.py):

```
import isl

a = isl.multi_union_pw_aff(
    "[n] -> A[{ S1[] -> [n]; S2[i,j] -> [i] }, "
    "{ S1[] -> [0]; S2[i,j] -> [j] }]")
r = isl.union_map.convert_from(a)
print(r)
```

python invocation:

```
python < union_map_from_multi_union_pw_aff.py
```

python output:

```
[n] -> { S2[i, j] -> A[i, j]; S1[] -> A[n, 0] }
```

Example 4.47. *Here is an example of a conversion where some information is lost.*
`python` *input (*`union_map_from_multi_union_pw_aff2.py`*):*

```
import isl

a = isl.multi_union_pw_aff(
    "A[{ S[i] -> [i] : i >= 0; S[i] -> [-i] : i < 0 }, "
    "{ S[i] -> [i - 1] : i >= 1 }]")
r = isl.union_map.convert_from(a)
print(r)
```

`python` *invocation:*

```
python < union_map_from_multi_union_pw_aff2.py
```

`python` *output:*

```
{ S[i] -> A[i, -1 + i] : i > 0 }
```

Notes

4.1. The “quasi-affine” terminology appears to have been coined by Quinton (1984), although it originally only allowed for an outer integer division. Feautrier (1991) allows integer divisions throughout the expression.
4.2. Quasi-affine selection trees were introduced by Feautrier (1991). Operations on such trees are described by Alias et al. (2012) and Guda (2013).

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

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

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*

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

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 previous 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.*

Note 5.2

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.*

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*

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

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*

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

where N *and* n *are constant symbols that satisfy* $\mathrm{N} = \lfloor \log_2 \mathrm{n} \rfloor$ *if* $\mathrm{n} \ge 1$. *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 facing 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 facing 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:

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

```
int f()
{
        int a;

Init:   a = 0;
        for (unsigned char 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:*

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

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 next 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 facing 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 preceding 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; Update[i] : 0 <= i < n; Reset
    ↪ [] }
```

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, Note 5.3 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.

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] -> prod[] : 0 <= i < n; T[i] -> A[i] : 0 <= i
    ↪ < n; T[i] -> B[i] : 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] -> { S[] -> A[o0, o0] : 0 <= o0 < n; T[i, j] -> A[i, j]
    ↪ : 0 <= i < n and j > i and 0 <= j < n }
"May-write:"
[n] -> { S[] -> A[o0, o0] : 0 <= o0 < n; T[i, j] -> A[i, j]
    ↪ : 0 <= i < n and j > i and 0 <= j < 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 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. Note 5.6

- 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 next 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 facing 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:"
{ T[] -> A_re[A[1] -> re[]]; S[] -> A_im[A[0] -> im[]]; S[]
    ↪ -> A_re[A[0] -> re[]] }
"May-write:"
{ T[] -> A_re[A[1] -> re[]]; S[] -> A_im[A[0] -> im[]]; S[]
    ↪ -> A_re[A[0] -> re[]] }
"May-read:"
{ S[] -> A_im[A[2] -> im[]]; S[] -> A_re[A[2] -> re[]]; T[]
    ↪ -> A_im[A[0] -> 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 $\boldsymbol{i}$.*

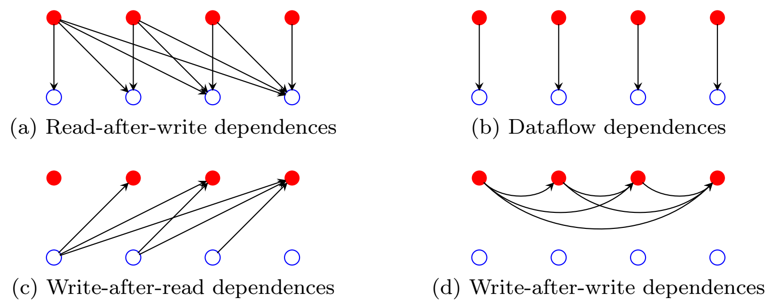
Definition 5.32 (Write-after-Read Dependence Relation)**.** *The* write-after-read dependence relation *maps a statement instance $\boldsymbol{i}$ to a statement instance $\boldsymbol{j}$ if $\boldsymbol{j}$ is executed after $\boldsymbol{i}$ and if it writes to a data element that is read by $\boldsymbol{i}$.*

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 Note 5.13 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 Note 5.14 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

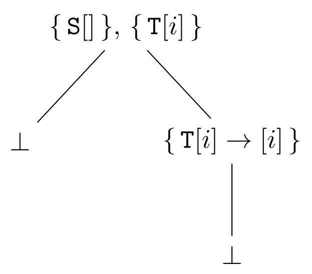


Figure 5.19: Schedule tree for the program fragment in Listing 5.2 on page 89

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 $\perp$ 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 next 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 $\boldsymbol{i}$ and $\boldsymbol{j}$ appear in same child then
            Move to common child
        else if $\boldsymbol{i}$ appears in earlier child then
            return true
        else
            return false
        end
    else
        Let $P$ be the partial schedule of the current band node
        if $P(\boldsymbol{i}) = P(\boldsymbol{j})$ then
            Move to single child
        else if $P(\boldsymbol{i}) \prec P(\boldsymbol{j})$ 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 following 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 $i \leftarrow 0$ 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 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' = 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' = j, k' = 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 preceding 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.member_set_coincident(0, True)
node = node.child(0)
node = node.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 to 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 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. Note 5.17

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* ($\mathrm{n} \le 0$) *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* $\mathrm{m} \le 2147483647$ *is further restricted to* $\mathrm{m} + \mathrm{n} \le 2147483648$*.*

python input (overflow.py) with source in Listing 5.21 on the preceding 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)) }
```

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. Note 5.18

Example 5.60. *Consider the program in Listing 5.22 on the next page, which is identical to that in Listing 5.21 on the preceding 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 next 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, m, n] -> {  : 0 <= S <= 2147483647 and -2147483648 <= m
    ↪ <= 2147483647 and -2147483648 <= n <= 2147483647 and
    ↪ n < m }
```

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, Note 5.20

conversely, a program statement may be broken up into several polyhedral statements. 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, Schreiber, et al. (2005) present an algorithm for computing contractions and provides an overview of earlier techniques. Darte, Schreiber, 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).

Chapter 6

Dependence Analysis

6.1 Dependence Analysis

Recall from Section 5.4 Dependence Relations that there is a dependence between two statement instances if they both (may) access the same memory element, at least one of which by writing to the memory element, and the first is executed before the second in the input program.

Computing dependence relations from the access relations and the schedule is fairly easy. Let us consider the read-after-write dependence relation as an example. Let W refer to the may-write access relation and let R refer to the may-read access relation. The first step is to compute the pairs of statement instances, one performing a write and one performing a read, that may access the same data element. This relation can be obtained by composing the may-write access relation with the inverse of the may-read access relation, i.e.,

$$R^{-1} \circ W. \tag{6.1}$$

Finally, only those pairs of statement instances should be retained where the first instance is executed before the second. That is, the pair needs to be a member of the order relation defined by the input schedule S. In other words, (6.1) needs to be intersected with this order relation, i.e.,

$$\left(R^{-1} \circ W\right) \cap <_S . \tag{6.2}$$

The write-after-read dependence relation and the write-after-write dependence relation can be similarly computed as

$$\left(W^{-1} \circ R\right) \cap <_S \tag{6.3}$$

and

$$\left(W^{-1} \circ W\right) \cap <_S . \tag{6.4}$$

A direct evaluation of the above expressions for the dependence relations requires the execution order relation $<_S$ to be computed from the schedule S first. This order relation can be easily computed from a Presburger relation

```
float f1(float);
float f2(float);

void f(int n, float A[restrict static n],
        float B[restrict static n])
{
        float t;

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

Listing 6.1: Input file *false.c*

representation of the schedule since it is a relation between pairs of domain elements in this flat schedule such that the corresponding range element are in lexicographic order. The execution order relation is then none other than the lexicographically-smaller-than relation between the flat schedule and itself. That is,

$$<_S = S \prec S, \tag{6.5}$$

with S a Presburger relation representation of the schedule. Note that, as explained in Section 5.6.1 Schedule Definition and Representation, an explicit computation of the order relation results in a data structure that is quadratic in the number of statements. This computation can be avoided by using the approximate dataflow analysis of Section 6.3 Approximate Dataflow Analysis below.

Example 6.1. *Consider the program in Listing 6.1, which is a completed version of the code in Listing 5.16 on page 107. The transcript below first computes the corresponding order relation and then derives the dependence relations. The resulting dependence relations are those shown in Example 5.34 on page 107, specialized to the set of instances that perform the relevant accesses.*
`iscc` input (`false.iscc`) with source in Listing 6.1:

```
P := parse_file "demo/false.c";
Write := P[2];
Read := P[3];
Schedule := map(P[4]);
Order := Schedule << Schedule;
print "Order relation:";
print Order;
print "Read-after-write dependence relation:";
(Write . Read^-1) * Order;
print "Write-after-read dependence relation:";
```

```
(Read . Write^-1) * Order;
print "Write-after-write dependence relation:";
(Write . Write^-1) * Order;
```

iscc invocation:

```
iscc < false.iscc
```

iscc output:

```
"Order relation:"
[n] -> { S[i] -> T[i'] : i' > i; S[i] -> T[i' = i]; S[i] ->
    ↪ S[i'] : i' > i; T[i] -> T[i'] : i' > i; T[i] -> S[i']
    ↪  : i' > i }
"Read-after-write dependence relation:"
[n] -> { S[i] -> T[i'] : 0 <= i < n and i' > i and 0 <= i' <
    ↪  n; S[i] -> T[i' = i] : 0 <= i < n }
"Write-after-read dependence relation:"
[n] -> { T[i] -> S[i'] : 0 <= i < n and i' > i and 0 <= i' <
    ↪  n }
"Write-after-write dependence relation:"
[n] -> { S[i] -> S[i'] : 0 <= i < n and i' > i and 0 <= i' <
    ↪  n }
```

6.2 Dataflow Analysis

The computation of dataflow dependences is slightly more complicated. Assume first that they can be computed exactly. This means in particular that the write and read access relations are known and represented exactly. Several ways of computing the dataflow dependences are described in order to illustrate the use of some of the operations on sets and binary relations and to point out some possible pitfalls of the different alternatives. Note 6.1

As explained in Section 5.4 Dependence Relations, a dataflow dependence is a read-after-write dependence for which there is no intermediate write to the same memory location. One way of computing dataflow dependences is then to remove those read-after-write dependences for which there is such an intermediate write. The removed dependences are said to be *killed* by the intermediate write. In order to be able to match the intermediate writes with the right read-after-write dependences, the dependences need to keep track of the memory element involved. For this purpose, the read access relation R, mapping statement instances to data elements, is replaced by its range projection

$$R_1 = \underrightarrow{\operatorname{ran}} R, \tag{6.6}$$

mapping pairs of statement instances and data elements to the data elements. Similarly, the write access relation W is replaced by

$$W_1 = \underrightarrow{\operatorname{ran}} W. \tag{6.7}$$

Only replacing R by R_1 and W by W_1 does not produce the desired result because $<_S$ still contains pairs of statement instances rather than pairs of pairs of statement instances and data elements. The data elements can be introduced into the schedule by setting

$$S_1 = S \circ \left(\underrightarrow{\operatorname{dom}}\,(R \cup W)\right). \tag{6.8}$$

The read-after-write dependences with associated array elements can then be computed as

$$D_1 = \left(R_1^{-1} \circ W_1\right) \cap <_{S_1}, \tag{6.9}$$

with $<_{S_1}$ computed as in (6.5). Similarly, the write-after-write dependences with associated array elements can be computed as

$$O_1 = \left(W_1^{-1} \circ W_1\right) \cap <_{S_1}. \tag{6.10}$$

The flow dependences are then the read-after-write dependences, apart from those that can be obtained as a combination of write-after-write dependence and a read-after-write dependence, i.e.,

$$F_1 = D_1 \setminus (D_1 \circ O_1). \tag{6.11}$$

The reference to the array elements can be removed from F_1 by taking its domain factor.

Example 6.2. *The transcript below illustrates the computation of the dataflow dependences of the code in Listing 6.1 on page 134. The final step of computing the domain factor is performed by taking the domain of the zipped relation. The resulting dataflow dependence relation is equal to the one from Example 5.38 on page 109, specialized to the set of instances that perform the relevant accesses.* iscc *input* (dataflow.iscc) *with source in Listing 6.1 on page 134:*

```
P := parse_file "demo/false.c";
Write := P[2];
Read := P[3];
Schedule := map(P[4]);
Write1 := range_map Write;
Read1 := range_map Read;
Schedule1 := (domain_map (Read + Write)) . Schedule;
Order1 := Schedule1 << Schedule1;
RAW := (Write1 . Read1^-1) * Order1;
WAW := (Write1 . Write1^-1) * Order1;
Flow := RAW - (WAW . RAW);
print "Write access relation:";
print Write1;
print "Read access relation:";
print Read1;
print "Schedule:";
print Schedule1;
print "Order relation:";
print Order1;
print "Read-after-write dependence relation:";
print RAW;
print "Write-after-write dependence relation:";
print WAW;
print "Flow dependence relation:";
print Flow;
unwrap (domain (zip Flow));
```

`iscc` *invocation:*

```
iscc < dataflow.iscc
```

`iscc` *output:*

```
"Write access relation:"
[n] -> { [S[i] -> t[]] -> t[] : 0 <= i < n; [T[i] -> B[i]]
   ↪ -> B[i] : 0 <= i < n }
"Read access relation:"
[n] -> { [T[i] -> t[]] -> t[] : 0 <= i < n; [S[i] -> A[i]]
   ↪ -> A[i] : 0 <= i < n }
"Schedule:"
[n] -> { [T[i] -> t[]] -> [i, 1] : 0 <= i < n; [S[i] -> t[]]
   ↪  -> [i, 0] : 0 <= i < n; [T[i] -> B[i]] -> [i, 1] : 0
   ↪  <= i < n; [S[i] -> A[i]] -> [i, 0] : 0 <= i < n }
"Order relation:"
[n] -> { [S[i] -> t[]] -> [S[i'] -> A[i']] : 0 <= i < n and
   ↪ i' > i and 0 <= i' < n; [T[i] -> B[i]] -> [T[i'] -> t
   ↪ []] : 0 <= i < n and i' > i and 0 <= i' < n; [T[i] ->
   ↪  B[i]] -> [S[i'] -> A[i']] : 0 <= i < n and i' > i
   ↪ and 0 <= i' < n; [T[i] -> t[]] -> [T[i'] -> t[]] : 0
   ↪ <= i < n and i' > i and 0 <= i' < n; [S[i] -> A[i]]
   ↪ -> [S[i'] -> A[i']] : 0 <= i < n and i' > i and 0 <=
   ↪ i' < n; [T[i] -> t[]] -> [T[i'] -> B[i']] : 0 <= i <
   ↪ n and i' > i and 0 <= i' < n; [S[i] -> t[]] -> [T[i']
   ↪  -> t[]] : i >= 0 and i < i' < n; [S[i] -> t[]] -> [T
   ↪ [i' = i] -> t[]] : 0 <= i < n; [S[i] -> A[i]] -> [T[i
   ↪ '] -> B[i']] : 0 <= i < n and i' > i and 0 <= i' < n;
   ↪  [S[i] -> A[i]] -> [T[i' = i] -> B[i]] : 0 <= i < n;
   ↪ [S[i] -> t[]] -> [S[i'] -> t[]] : i >= 0 and i < i' <
   ↪  n; [T[i] -> t[]] -> [S[i'] -> A[i']] : 0 <= i < n
   ↪ and i' > i and 0 <= i' < n; [S[i] -> t[]] -> [T[i']
   ↪ -> B[i']] : 0 <= i < n and i' > i and 0 <= i' < n; [S
   ↪ [i] -> t[]] -> [T[i' = i] -> B[i]] : 0 <= i < n; [T[i
   ↪ ] -> t[]] -> [S[i'] -> t[]] : 0 <= i < n and i' > i
   ↪ and 0 <= i' < n; [T[i] -> B[i]] -> [T[i'] -> B[i']] :
   ↪  i >= 0 and i < i' < n; [T[i] -> B[i]] -> [S[i'] -> t
   ↪ []] : 0 <= i < n and i' > i and 0 <= i' < n; [S[i] ->
   ↪  A[i]] -> [S[i'] -> t[]] : 0 <= i < n and i' > i and
   ↪ 0 <= i' < n; [S[i] -> A[i]] -> [T[i'] -> t[]] : 0 <=
    ↪ i < n and i' > i and 0 <= i' < n; [S[i] -> A[i]] -> [
    ↪ T[i' = i] -> t[]] : 0 <= i < n }
"Read-after-write dependence relation:"
[n] -> { [S[i] -> t[]] -> [T[i'] -> t[]] : i >= 0 and i < i'
    ↪  < n; [S[i] -> t[]] -> [T[i' = i] -> t[]] : 0 <= i <
    ↪ n }
"Write-after-write dependence relation:"
[n] -> { [S[i] -> t[]] -> [S[i'] -> t[]] : 0 <= i < n and i'
    ↪  > i and 0 <= i' < n }
"Flow dependence relation:"
[n] -> { [S[i] -> t[]] -> [T[i' = i] -> t[]] : 0 <= i < n }
[n] -> { S[i] -> T[i' = i] : 0 <= i < n }
```

An alternative to modifying the schedule in (6.8) and using the order defined by this modified schedule is to use the order defined by the original schedule and to apply it to the right part of $R_1^{-1} \circ W_1$. In particular, (6.9) can be replaced by

$$D_1 = \operatorname{zip}\left(\left(\operatorname{zip}\left(R_1^{-1} \circ W_1\right)\right) \cap_{\text{dom}} \mathcal{W}\left(<_S\right)\right), \tag{6.12}$$

while (6.10) can be replaced by

$$O_1 = \operatorname{zip}\left(\left(\operatorname{zip}\left(W_1^{-1} \circ W_1\right)\right) \cap_{\text{dom}} \mathcal{W}\left(<_S\right)\right), \tag{6.13}$$

Example 6.3. *The transcript below illustrates that this alternative way of computing D_1 and O_1 produces the same results.*
`iscc` *input (*`dataflow2.iscc`*) with source in Listing 6.1 on page 134:*

```
P := parse_file "demo/false.c";
Write := P[2];
Read := P[3];
Schedule := map(P[4]);
Write1 := range_map Write;
Read1 := range_map Read;
Order := Schedule << Schedule;
RAW := zip ((zip (Write1 . Read1^-1)) * wrap(Order));
WAW := zip ((zip (Write1 . Write1^-1)) * wrap(Order));
print "Order relation:";
print Order;
print "Read-after-write dependence relation:";
print RAW;
print "Write-after-write dependence relation:";
print WAW;
```

`iscc` *invocation:*

```
iscc < dataflow2.iscc
```

`iscc` *output:*

```
"Order relation:"
```

```
[n] -> { S[i] -> T[i'] : i' > i; S[i] -> T[i' = i]; S[i] ->
    ↪ S[i'] : i' > i; T[i] -> T[i'] : i' > i; T[i] -> S[i']
    ↪  : i' > i }
"Read-after-write dependence relation:"
[n] -> { [S[i] -> t[]] -> [T[i'] -> t[]] : 0 <= i < n and i'
    ↪  > i and 0 <= i' < n; [S[i] -> t[]] -> [T[i' = i] ->
    ↪ t[]] : 0 <= i < n }
"Write-after-write dependence relation:"
[n] -> { [S[i] -> t[]] -> [S[i'] -> t[]] : 0 <= i < n and i'
    ↪  > i and 0 <= i' < n }
```

The main problem with the way of computing dataflow dependences described above is that it cannot easily deal with approximations. In particular, D_1 is used on both sides of a subtraction operation in (6.11). This means that if D_1 is an overapproximation, then the result is not guaranteed to be an overapproximation or an underapproximation, while most practical uses depend on such a guarantee.

An alternative way of computing dataflow dependences is to not think of them as the read-after-write dependences with no intermediate write, but rather as pairing off each read with the *last* preceding write to the same memory element. The first step in this computation is to pair off each read with *all* preceding writes to the same memory element:

$$A = \left(W^{-1} \circ R\right) \cap (<_S)^{-1}, \tag{6.14}$$

with W the write access relation, R the read access relation and S the schedule. To simplify the exposition, it is assumed here that every statement instance reads or writes at most one data element. Otherwise, it would be required to also keep track of the relevant data element(s). Having collected all statement instances that previously wrote to the memory element read by a given statement instance, the last of these writing statement instances now needs to be computed. In particular, the statement instance that is assigned the lexicographically greatest value by the (Presburger relation representation of) the schedule needs to be computed. The writing instances are then first mapped to their positions in the schedule, the lexicographically greatest position is selected and then the positions are mapped back to the writing instances. The result maps read instances to write instances and so the dataflow dependence relation is the inverse of this result. That is,

$$F = \left(S^{-1} \circ \operatorname{lexmax}\left(S \circ A\right)\right)^{-1}. \tag{6.15}$$

Note that this computation assumes that the order defined by the schedule is total, i.e., that the schedule assigns a different position to each statement instance, such that the statement instance can be recovered from the position. Keeping track of an extra copy of the write statement instance before composition with the schedule does not help because the lexicographical maximum would then be computed as a function of both the read and the write statement instance.

Example 6.4. *The transcript below uses the method described above to compute the same flow dependences as in Example 6.2 on page 136.*
`iscc` *input (*`dataflow3.iscc`*) with source in Listing 6.1 on page 134:*

```
P := parse_file "demo/false.c";
Write := P[2];
Read := P[3];
Schedule := map(P[4]);
Order := Schedule << Schedule;
A := (Read . Write^-1) * (Order^-1);
F := ((lexmax (A . Schedule)) . Schedule^-1)^-1;
print "Reads mapped to all previous writes:";
print A;
print "Flow dependences:";
print F;
```

`iscc` *invocation:*

```
iscc < dataflow3.iscc
```

`iscc` *output:*

```
"Reads mapped to all previous writes:"
[n] -> { T[i] -> S[i'] : 0 <= i < n and 0 <= i' < i and i' <
    ↪  n; T[i] -> S[i' = i] : 0 <= i < n }
"Flow dependences:"
[n] -> { S[i] -> T[i' = i] : 0 <= i < n }
```

6.3 Approximate Dataflow Analysis

The previous section described how to perform (simplified forms of) dataflow analysis exactly. However, the access relations may not always be known exactly since the accesses may depend on run-time information or they may not be representable as Presburger relations. In such cases, dataflow analysis can only be performed approximately. Broadly speaking, there are two approaches for computing approximate dataflow.

- One approach is to operate on may and must versions of access relations and to compute approximate dataflow directly.

- Note 6.2 The other is to keep track of additional run-time information and to derive an exact, but run-time-dependent dataflow dependence relation. An approximate dataflow relation can then be derived by projecting out all the run-time-dependent information, possibly after a simplification based on exploiting some known properties of the run-time-dependent information. Because this approach keeps track of more information, it is in general capable of computing more accurate approximations.

This section is devoted to a direct computation of approximate dataflow dependences.

Recall that the main difference between memory-based dependence analysis and value-based dependence analysis is that in value-based dependence analysis, a write kills all dependences between another write preceding the first write and a read following the first write. The main idea behind the approximate dataflow analysis described in this section is then to only allow must-writes to kill any dependences. In the worst case, none of the may-writes are also must-writes and then the result of the dependence analysis is the same as that of a memory-based dependence analysis. In order not to be too specific to the standard dataflow analysis, the operation is formulated in terms of may-sources, must-sources and sinks rather than in terms of may-writes, must-writes and reads.

Operation 6.5 (Approximate Dataflow Analysis)**.** Approximate dataflow analysis *takes as input three binary relations and a schedule on the domains of the binary relations. The three binary relations are called the* sink K*, the* may-source Y *and the* must-source T*. The schedule S is used to evaluate the predicates "last", "before" and "after" in the following sentences. For each domain element $\boldsymbol{i}$ of the sink and for each corresponding range element $\boldsymbol{a}$, approximate dataflow analysis determines the* last *domain element $\boldsymbol{j}$ of the* must-source *that is executed before $\boldsymbol{i}$ and also has $\boldsymbol{a}$ as corresponding range element. Furthermore, the analysis collects* all *domain elements $\boldsymbol{k}$ of the* may-source *that are executed before $\boldsymbol{i}$ and after $\boldsymbol{j}$ and that also have $\boldsymbol{a}$ as corresponding range element. If no such $\boldsymbol{j}$ can be found for a particular combination of $\boldsymbol{i}$ and $\boldsymbol{a}$, then the "after $\boldsymbol{j}$" condition is dropped. In other words, for each domain element of the sink and for each corresponding range element, the previously executed domain elements of the must-source and may-source that share this range element are collected until a domain element of the must-source is found. The collection of all such triplets $\boldsymbol{j} \to (\boldsymbol{i} \to \boldsymbol{a})$ and $\boldsymbol{k} \to (\boldsymbol{i} \to \boldsymbol{a})$ forms the* may-dependence relation*. The subset of the $\boldsymbol{j} \to (\boldsymbol{i} \to \boldsymbol{a})$ for which there are no intermediate $\boldsymbol{k}$ forms the* must-dependence relation*. The subset of the sink for which no corresponding domain element $\boldsymbol{j}$ can be found forms the* may-no-source relation*. The subset of the sink for which no corresponding domain elements $\boldsymbol{j}$ or $\boldsymbol{k}$ can be found forms the* must-no-source relation*. That is, the may-dependence relation is*

$$\begin{aligned}\{\, \boldsymbol{k} \to (\boldsymbol{i} \to \boldsymbol{a}) : \boldsymbol{i} \to \boldsymbol{a} \in K \wedge \boldsymbol{k} \to \boldsymbol{a} \in (T \cup Y) \wedge \boldsymbol{k} <_S \boldsymbol{i} \wedge \\ \neg\,(\exists \boldsymbol{j} : \boldsymbol{j} \to \boldsymbol{a} \in T \wedge \boldsymbol{k} <_S \boldsymbol{j} <_S \boldsymbol{i})\,\},\end{aligned} \tag{6.16}$$

the must-dependence relation is

$$\begin{aligned}\{\, \boldsymbol{k} \to (\boldsymbol{i} \to \boldsymbol{a}) : \boldsymbol{i} \to \boldsymbol{a} \in K \wedge \boldsymbol{k} \to \boldsymbol{a} \in T \wedge \boldsymbol{k} <_S \boldsymbol{i} \wedge \\ \neg\,(\exists \boldsymbol{j} : \boldsymbol{j} \to \boldsymbol{a} \in (T \cup Y) \wedge \boldsymbol{k} <_S \boldsymbol{j} <_S \boldsymbol{i})\,\},\end{aligned} \tag{6.17}$$

the may-no-source relation is

$$\{\, \boldsymbol{i} \to \boldsymbol{a} \in K : \neg\,(\exists \boldsymbol{j} : \boldsymbol{j} \to \boldsymbol{a} \in T \wedge \boldsymbol{j} <_S \boldsymbol{i})\,\}, \tag{6.18}$$

and the must-no-source relation is

$$\{\, \boldsymbol{i} \to \boldsymbol{a} \in K : \neg\,(\exists \boldsymbol{j} : \boldsymbol{j} \to \boldsymbol{a} \in (T \cup Y) \wedge \boldsymbol{j} <_S \boldsymbol{i})\,\}. \tag{6.19}$$

In `isl`, the approximate dataflow analysis functionality is available through the `isl_union_access_info_compute_flow` function. This function takes an `isl_union_access_info` object (describing the sink, the must-source, the may-source and the schedule) as input, and it produces an `isl_union_flow` object (describing the may-dependence relation, the must-dependence relation, the may-no-source relation and the must-no-source relation) as output. The implementation combines techniques described in the previous sections, but it avoids constructing a global order relation $<_S$. The `isl_union_access_info` object is constructed from the sink relation by performing a call to the function `isl_union_access_info_from_sink`. The must-source, the may-source and the schedule (in the form of either a schedule tree or a Presburger relation) can be set using the following functions.

- `isl_union_access_info_set_must_source`
- `isl_union_access_info_set_may_source`
- `isl_union_access_info_set_schedule`, in case the schedule is represented as a schedule tree.
- `isl_union_access_info_set_schedule_map`, in case the schedule is represented as a Presburger relation.

If the must-source and/or may-source is not set, then they are assumed to be empty. The schedule is required to be set. The may-dependence relation, the must-dependence relation, the may-no-source relation and the must-no-source relation can be extracted from an `isl_union_flow` object using the following functions.

- `isl_union_flow_get_full_may_dependence` returns the complete may-dependence relation of (6.16).
- `isl_union_flow_get_full_must_dependence` returns the complete must-dependence relation of (6.17).
- `isl_union_flow_get_may_dependence` returns the may-dependence relation with the accessed element projected out. That is, it is the result of computing the range factor of the complete may-dependence relation considered as a range product.
- `isl_union_flow_get_must_dependence` returns the must-dependence relation with the accessed element projected out. That is, it is the result of computing the range factor of the complete must-dependence relation considered as a range product.
- `isl_union_flow_get_may_no_source`

- `isl_union_flow_get_must_no_source`

In `iscc`, the approximate dataflow analysis is available as the `last-any-before-under` operation. In particular, the argument of `last` specifies the must-source, the argument of `any` specifies the may-source, the argument of `before` specifies the sink, and the argument of `under` specifies the schedule, in either schedule tree or Presburger relation representation. One of `last` or `any` (together with its argument) may be omitted. If only `last` is used, then the result is a list containing the must-dependence relation and the must-no-source relation. Otherwise, the output is the may-dependence relation.

6.4 Applications of Approximate Dataflow Analysis

The most obvious application of the approximate dataflow analysis operation is to compute dataflow dependence relations. In particular, the sink is set to the may-read access relation, the may-source is set to the may-write access relation and the must-source is set to the must-write access relation. The resulting may-dependence relation represents the may-dataflow dependence relation, while the must-dependence relation represents the must-dataflow dependence relation. Furthermore, the may-no-source relation can be used as a *may-live-in relation*. This is a relation that contains the read accesses that may read a value that was not written inside the analyzed program fragment. If the write accesses are known exactly, then only the must-source needs to be specified and the must-no-source relation represents the exact live-in relation.

Example 6.6. *The transcript below illustrates the computation of dataflow dependences using approximate dataflow analysis in the exact case. The result is the same as what was computed in Example 6.2 on page 136 and Example 6.4 on page 140, which also assumed exact write access relations. Additionally, the live-in accesses are also computed. In particular, all reads from `A` in statement `S` are live-in.*
`iscc` input (`dataflow4.iscc`) with source in Listing 6.1 on page 134:

```
P := parse_file "demo/false.c";
Write := P[2];
Read := P[3];
Schedule := P[4];
F := last Write before Read under Schedule;
print "Flow dependences:";
print F[0];
print "Live-in accesses:";
print F[1];
```

`iscc` invocation:

```
iscc < dataflow4.iscc
```

`iscc` output:

```
"Flow dependences:"
[n] -> { S[i] -> T[i' = i] : 0 <= i < n }
"Live-in accesses:"
[n] -> { S[i] -> A[i] : 0 <= i < n }
```

Example 6.7. *The transcript below performs the same computation as in Example 6.6 on the previous page, except that it does not assume that the write access relation is exact. Since the write access relation is effectively exact, the results are the same.*
`python` *input (*`dataflow.py` *) with source in Listing 6.1 on page 134:*

```
import isl
import pet

pet.options.set_autodetect(True)
scop = pet.scop.extract_from_C_source("demo/false.c", "f")
schedule = scop.get_schedule()
may_read = scop.get_may_reads()
may_write = scop.get_may_writes()
must_write = scop.get_must_writes()

access = isl.union_access_info(may_read)
access = access.set_may_source(may_write)
access = access.set_must_source(must_write)
access = access.set_schedule(schedule)
flow = access.compute_flow()
print("May-flow dependences:")
print(flow.get_may_dependence())
print("May-live-in accesses:")
print(flow.get_may_no_source())
```

`python` *invocation:*

```
python < dataflow.py
```

`python` *output:*

```
May-flow dependences:
[n] -> { S[i] -> T[i' = i] : 0 <= i < n }
May-live-in accesses:
[n] -> { S[i] -> A[i] : 0 <= i < n }
```

Whereas exact dataflow analysis represents an extreme case of approximate dataflow analysis where only the must-source is specified, the dependence analysis of Section 6.1 Dependence Analysis is at the other extreme where only the may-source is specified. In particular, the read-after-write dependence relation is computed by setting the may-source to the may-write access relation and the sink to the may-read access relation. Similarly, the write-after-read dependence relation is computed by setting the may-source to the may-read access relation and the sink to the may-write access relation, while the write-after-write dependence relation is computed by setting both the may-source and the sink to the may-write access relation.

Example 6.8. *The transcript below repeats the computation of Example 6.1 on page 134 using the approximate dataflow analysis functionality. The results are the same.*
iscc input (false2.iscc) with source in Listing 6.1 on page 134:

```
P := parse_file "demo/false.c";
Write := P[2];
Read := P[3];
Schedule := P[4];
print "Read-after-write dependence relation:";
any Write before Read under Schedule;
print "Write-after-read dependence relation:";
any Read before Write under Schedule;
print "Write-after-write dependence relation:";
any Write before Write under Schedule;
```

iscc invocation:

```
iscc < false2.iscc
```

iscc output:

```
"Read-after-write dependence relation:"
[n] -> { S[i] -> T[i'] : i >= 0 and i <= i' < n }
"Write-after-read dependence relation:"
[n] -> { T[i] -> S[i'] : i >= 0 and i < i' < n }
"Write-after-write dependence relation:"
[n] -> { S[i] -> S[i'] : i >= 0 and i < i' < n }
```

In order to compute a tagged may-dataflow dependence relation or a tagged must-dataflow dependence relation, the approximate dataflow analysis needs to be applied to the tagged access relations. The domains of these relations are not a subset of the instance set, however, so they do not match the domain of the schedule. The schedule can however be pulled back to apply to the domains of the tagged access relations.

Example 6.9. *python input (tagged_dataflow.py) with source in Listing 6.1 on page 134:*

```
import isl
import pet

pet.options.set_autodetect(True)
scop = pet.scop.extract_from_C_source("demo/false.c", "f")
schedule = scop.get_schedule()
may_read = scop.get_tagged_may_reads()
may_write = scop.get_tagged_may_writes()
```

```
must_write = scop.get_tagged_must_writes()
tagged_instances = may_write.union(may_read).domain()
tagged_instances = tagged_instances.unwrap()
drop_tags = tagged_instances.domain_map_union_pw_multi_aff()
schedule = schedule.pullback(drop_tags)

access = isl.union_access_info(may_read)
access = access.set_may_source(may_write)
access = access.set_must_source(must_write)
access = access.set_schedule(schedule)
flow = access.compute_flow()
print("Tagged may-read access relation:")
print(may_read)
print("Tagged may-write access relation:")
print(may_write)
print("Tagged may-flow dependences:")
print(flow.get_may_dependence())
print("Tagged may-live-in accesses:")
print(flow.get_may_no_source())
```

python invocation:

```
python < tagged_dataflow.py
```

python output:

```
Tagged may-read access relation:
[n] -> { [S[i] -> __pet_ref_2[]] -> A[i] : 0 <= i < n; [T[i]
    ↪  -> __pet_ref_4[]] -> t[] : 0 <= i < n }
Tagged may-write access relation:
[n] -> { [S[i] -> __pet_ref_1[]] -> t[] : 0 <= i < n; [T[i]
    ↪ -> __pet_ref_3[]] -> B[i] : 0 <= i < n }
Tagged may-flow dependences:
[n] -> { [S[i] -> __pet_ref_1[]] -> [T[i' = i] ->
    ↪ __pet_ref_4[]] : 0 <= i < n }
Tagged may-live-in accesses:
[n] -> { [S[i] -> __pet_ref_2[]] -> A[i] : 0 <= i < n }
```

6.5 Kills

If there are many may-writes that are not also must-writes, then the approximate dataflow analysis can be fairly inaccurate. In some cases, the user may be able to tell that no dataflow can occur through a given piece of memory and a certain point in the program text. The user can communicate this information by inserting a `__pencil_kill` statement at that point in the program text, killing the given piece of memory. In particular, a `__pencil_kill` statement consists of a "call" to `__pencil_kill` with as arguments the data elements through which no data can flow at that point. These kills can then be used to kill dependences by including them in the must-source of an approximate

Note 6.3

```
int f(int);
void g(int N, int perm[restrict static N],
    int A[restrict static N])
{
S:  A[0] = 1;
K:  __pencil_kill(A);
    for (int i = 0; i < N; ++i)
T:      A[perm[i]] = f(i);
U:  A[0] = f(A[0]);
}
```

Listing 6.2: Input file *kill.c*

dataflow analysis. The output may then contain "dependences" emanating from these kills, but they can simply be removed.

The kills are collected by `pet` and can be extracted using a call to the functions `pet_scop_get_must_kills` or `pet_scop_get_tagged_must_kills`. Note that `pet` introduces a separate kill statement for each argument of the `__pencil_kill` call. The label on the program statement may therefore not be preserved. In `iscc`, the kills are not available in the output of `parse_file` and the statements performing the kills are filtered out from the remaining output.

Example 6.10. *Consider the code in Listing 6.2 and assume that `perm` is a permutation of the integers 0 to $N - 1$. This means that no dataflow can occur between statement `S` and statement `U` because `A[0]` is overwritten by* some *instance of statement `T`. However, the compiler does not know that `perm` is a permutation. The user has therefore added a call to `__pencil_kill` indicating that the entire array `A` is killed (by the instances of `T`). The transcript below shows the effect of taking into account the kills during dataflow analysis. In particular, when kills are not taken into account, there is an additional dependence from statement `S` to statement `U`.*
`python` input (`kill.py`) with source in Listing 6.2:

```
import isl
import pet

pet.options.set_autodetect(True)
scop = pet.scop.extract_from_C_source("demo/kill.c", "g")
schedule = scop.get_schedule()
may_read = scop.get_may_reads()
may_write = scop.get_may_writes()
must_write = scop.get_must_writes()
kill = scop.get_must_kills()

access = isl.union_access_info(may_read)
access = access.set_may_source(may_write)
access = access.set_must_source(must_write)
access = access.set_schedule(schedule)
flow1 = access.compute_flow()
access = access.set_must_source(must_write.union(kill))
flow2 = access.compute_flow()

print("May-flow dependences without kills:")
print(flow1.get_may_dependence())
print("Kills:")
print(kill)
print("May-flow dependences with kills:")
f = flow2.get_may_dependence()
f = f.subtract_domain(kill.domain())
print(f)
```

`python` *invocation:*

```
python < kill.py
```

`python` *output:*

```
May-flow dependences without kills:
[N] -> { T[i] -> U[] : 0 <= i < N; S[] -> U[] : N > 0 }
Kills:
[N] -> { S_1[] -> A[o0] : 0 <= o0 < N }
May-flow dependences with kills:
[N] -> { T[i] -> U[] : 0 <= i < N }
```

Besides keeping track of the kills manually introduced by the user, `pet` also introduces its own kills. In particular, for any local variable that is declared inside the analyzed region, two kill statements are automatically inserted, one at the point where the variable is declared and one at the point where the variable goes out of scope. The latter is omitted when the variable does not go out of scope within the analyzed region.

Example 6.11. *The code shown in Listing 6.3 on the facing page is a variation of the code shown in Listing 5.20 on page 115 with a local variable. The transcript below shows the kills that have been introduced by* `pet` *and the locations where they are introduced in the schedule.*
`python` *input (*`local.py` *) with source in Listing 6.3 on the facing page:*

```
import isl
import pet

scop = pet.scop.extract_from_C_source("demo/matmul2.c",
                                      "matmul")
schedule = scop.get_schedule()
kill = scop.get_must_kills()
print(kill)
print(schedule.get_root())
```

`python` *invocation:*

```
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) {
            float t;
I:          t = 0;
            for (int k = 0; k < K; ++k)
U:              t = t + A[i][k] * B[k][j];
S:          C[i][j] = t;
        }
#pragma endscop
}
```

Listing 6.3: Input file *matmul2.c*

```
python < local.py
```

python output:

```
[N, M, K] -> { S_1[i, j] -> t[] : 0 <= i < M and 0 <= j < N;
    ↪  S_0[i, j] -> t[] : 0 <= i < M and 0 <= j < N }
# YOU ARE HERE
domain: "[N, M, K] -> { S_1[i, j] : 0 <= i < M and 0 <= j <
    ↪ N; I[i, j] : 0 <= i < M and 0 <= j < N; S_0[i, j] : 0
    ↪  <= i < M and 0 <= j < N; S[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[{ S_1[i, j] -> [(i)]; I[i, j]
      ↪ -> [(i)]; S_0[i, j] -> [(i)]; S[i, j] -> [(i)]; U[i
      ↪ , j, k] -> [(i)] }]"
  child:
    schedule: "[M, N, K] -> L_1[{ S_1[i, j] -> [(j)]; I[i, j
        ↪ ] -> [(j)]; S_0[i, j] -> [(j)]; S[i, j] -> [(j)];
        ↪  U[i, j, k] -> [(j)] }]"
    child:
      sequence:
      - filter: "[M, N, K] -> { S_0[i, j] }"
      - 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)]
              ↪ }]"
      - filter: "[M, N, K] -> { S[i, j] }"
```

```
int f(int);

void g(int n, int A[const restrict static n])
{
#pragma scop
S:      A[f(0)] = 1;
T:      A[0] = 0;
#pragma endscop
}
```

Listing 6.4: Input file *live-out.c*

```
    - filter: "[M, N, K] -> { S_1[i, j] }"
```

6.6 Live-out Accesses

Approximate dataflow analysis can also be used to compute a *may-live-out relation*. This is a relation that contains the may-write accesses that may still have corresponding reads outside of the program fragment under analysis. In particular, it consists of all the may-write accesses that are *not* killed by a later must-write access or a kill. This set of killed may-write accesses can be computed by performing dataflow analysis with as sinks the must-writes and the kills and as may-sources the may-writes. The domain of the resulting may-dependence relation can then be removed from the set of all accesses. Note that it is important to consider the complete may-dependence relation (including the corresponding accessed elements) since a kill of one element accessed by a may-write should not be considered to be a kill of *all* elements accessed by that write.

Example 6.12. *Consider the program in Listing 6.4. Statement `S` may write to any element of `A`. The write in `T` only kills the write in `S` if the latter writes to element 0 of the array. The transcript below illustrates the computation of the correponding may-live-out access relation.*
`python` input (`live-out.py`) with source in Listing 6.4:

```
import isl
import pet

scop = pet.scop.extract_from_C_source("demo/live-out.c",
                                      "g")
schedule = scop.get_schedule()
may_write = scop.get_may_writes()
must_write = scop.get_must_writes()
kill = scop.get_must_kills()
```

```
access = isl.union_access_info(must_write.union(kill))
access = access.set_may_source(may_write)
access = access.set_schedule(schedule)
flow = access.compute_flow()
dep = flow.get_full_may_dependence()
killed = dep.range_factor_range()
live_out = may_write.subtract(killed)
print(live_out)
```

python invocation:

```
python < live-out.py
```

python output:

```
[n] -> { S[] -> A[o0] : 0 < o0 < n; T[] -> A[0] : n > 0 }
```

Notes

6.1. Exact dataflow analysis is described by Feautrier (1991) and, using a different algorithm, by Pugh and Wonnacott (1994). Yuki, Feautrier, et al. (2013) describe an extension to X10 programs.
6.2. One technique that follows this approach is the “fuzzy array dataflow analysis technique of Barthou et al. (1997). Verdoolaege, Nikolov, et al. (2013) present a related technique.
6.3. The `__pencil_kill` statement was introduced by Baghdadi et al. (2015) and was inspired by the KILL statement of Vandierendonck et al. (2010).

Bibliography

Acharya, Aravind and Uday Bondhugula (2015). “PLUTO+: Near-complete Modeling of Affine Transformations for Parallelism and Locality”. In: *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP 2015. San Francisco, CA, USA: ACM, pp. 54–64. DOI: 10.1145/2688500.2688512. [131]

Alias, Christophe, Alain Darte, and Alexandru Plesco (Jan. 2012). “Optimizing Remote Accesses for Offloaded Kernels: Application to High-Level Synthesis for FPGA”. In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12).* Paris, France. [85]

Ancourt, Corinne and François Irigoin (Apr. 1991). “Scanning polyhedra with DO loops”. In: *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming.* Williamsburg, VA, pp. 39–50. DOI: 10.1145/109625.109631. [131]

Arnon, Dennis S., George E. Collins, and Scott McCallum (1984). “Cylindrical algebraic decomposition I: The basic algorithm”. In: *SIAM Journal on Computing* 13.4, pp. 865–877. DOI: 10.1137/0213054. [69]

Baghdadi, Riyadh, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Róbert Dávid, and Elnar Hajiyev (Oct. 2015). “PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming”. In: *Proc. Parallel Architectures and Compilation Techniques (PACT'15).* [130, 151]

Barthou, Denis, Jean-François Collard, and Paul Feautrier (1997). “Fuzzy Array Dataflow Analysis”. In: *J. Parallel Distrib. Comput.* 40.2, pp. 210–226. DOI: 10.1006/jpdc.1996.1261. [151]

Bastoul, Cédric (2004). “Code Generation in the Polyhedral Model Is Easier Than You Think”. In: *PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques.* Washington, DC, USA: IEEE Computer Society, pp. 7–16. DOI: 10.1109/PACT.2004.11. [131]

Bastoul, Cédric (May 2008). *Extracting polyhedral representation from high level languages.* Tech. rep. LRI, Paris-Sud University. [9, 131]

Bastoul, Cédric (Dec. 2012). “Contributions to High-Level Program Optimization”. Habilitation. Paris-Sud University, France. [8]

Ben-Kiki, Oren and Clark Evans (2009). *YAML Ain't Markup Language (YAML™) Version 1.2.* [130]

Boigelot, Bernard (1999). "Symbolic Methods for Exploring Infinite State Spaces". PhD thesis. Faculté des Sciences Appliquées de l'Université de Liège. [68]

Bondhugula, Uday, Albert Hartono, J. Ramanujam, and P. Sadayappan (2008). "A practical automatic polyhedral parallelizer and locality optimizer". In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation.* PLDI '08. Tucson, AZ, USA: ACM, pp. 101–113. DOI: 10.1145/1375581.1375595. [131]

Bruyère, Véronique (1985). "Entiers et automates finis". Mémoire de fin d'études. University of Mons, Belgium. [68]

Chen, Chun (June 2012). "Polyhedra scanning revisited". In: *SIGPLAN Not.* 47.6, pp. 499–508. DOI: 10.1145/2345156.2254123. [131]

Clauss, Philippe, Federico Javier Fernandez, Diego Garbervetsky, and Sven Verdoolaege (Aug. 2009). "Symbolic polynomial maximization over convex sets and its application to memory requirement estimation". In: *IEEE Transactions on VLSI Systems* 17.8, pp. 983–996. DOI: 10.1109/TVLSI.2008.2002049. [69]

Clauss, Philippe, Diego Garbervetsky, Vincent Loechner, and Sven Verdoolaege (2011). "Polyhedral Techniques for Parametric Memory Requirement Estimation". In: *Energy-Aware Memory Management for Embedded Multimedia Systems: A Computer-Aided Design Approach.* Ed. by F. Balasa and Dhiraj K. Pradhan. Chapman and Hall/CRC, pp. 117–149. DOI: 10.1201/b11418-5. [8]

Clauss, Philippe and Irina Tchoupaeva (2004). "A symbolic approach to bernstein expansion for program analysis and optimization". In: *Compiler Construction.* Springer, pp. 120–133. DOI: 10.1007/978-3-540-24723-4_9. [68]

Cohen, Albert, Sylvain Girbal, and Olivier Temam (2004). "A Polyhedral Approach to Ease the Composition of Program Transformations". In: *Euro-Par (Euro-Par)*, pp. 292–303. DOI: 10.1007/978-3-540-27866-5_38. [130]

Cohen, Albert, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache (2005). "Facilitating the search for compositions of program transformations". In: *Proceedings of the 19th annual international conference on Supercomputing.* ICS '05. Cambridge, Massachusetts: ACM, pp. 151–160. DOI: 10.1145/1088149.1088169. [130]

Cousot, Patrick and Nicolas Halbwachs (1978). "Automatic Discovery of Linear Restraints among Variables of a Program". In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages.* Tucson, Arizona: ACM Press, pp. 84–96. DOI: 10.1145/512760.512770. [9]

Creusillet, Béatrice and François Irigoin (Dec. 1996). "Interprocedural array region analyses". In: *Int. J. Parallel Program.* 24 (6), pp. 513–546. DOI: 10.1007/BFb0014191. [9]

Darte, Alain, Yves Robert, and Frédéric Vivien (2000). *Scheduling and Automatic Parallelization.* Birkhauser Boston. [131]

Darte, Alain, Robert Schreiber, and Gilles Villard (2004). *Lattice based memory allocation.* Research report 2004-23. Ecole Normale Supérieure de Lyon, 46 Allée d'Italie, 69364 Lyon Cedex 07: Laboratoire de l'Informatique du Parallélisme. [130]

Darte, Alain, Robert Schreiber, and Gilles Villard (2005). "Lattice-Based Memory Allocation". In: *IEEE Trans. Comput.* 54.10, pp. 1242–1257. DOI: 10.1109/TC.2005.167. [130]

Darte, Alain and Frédéric Vivien (1997). "Optimal Fine and Medium Grain Parallelism Detection in Polyhedral Reduced Dependence Graphs". In: *International Journal of Parallel Programming* 25.6, pp. 447–496. DOI: 10.1023/A:1025168022993. [130]

Feautrier, Paul (1988a). "Array expansion". In: *ICS '88: Proceedings of the 2nd international conference on Supercomputing.* St. Malo, France: ACM Press, pp. 429–441. DOI: 10.1145/55364.55406. [130]

Feautrier, Paul (1988b). "Parametric Integer Programming". In: *RAIRO Recherche Opérationnelle* 22.3, pp. 243–268. [9, 67, 68]

Feautrier, Paul (1991). "Dataflow analysis of array and scalar references". In: *International Journal of Parallel Programming* 20.1, pp. 23–53. DOI: 10.1007/BF01407931. [9, 85, 151]

Feautrier, Paul (Oct. 1992a). "Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time". In: *International Journal of Parallel Programming* 21.5, pp. 313–348. DOI: 10.1007/BF01407835. [131]

Feautrier, Paul (Dec. 1992b). "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time". In: *International Journal of Parallel Programming* 21.6, pp. 389–420. DOI: 10.1007/BF01379404. [131]

Feautrier, Paul (1998). "A Parallelization Framework for Recursive Tree Programs". In: *Euro-Par.* Ed. by David J. Pritchard and Jeff Reeve. Vol. 1470. Lecture Notes in Computer Science. Springer, pp. 470–479. [130]

Feautrier, Paul (Jan. 2015). "The Power of Polynomials". In: *5th International Workshop on Polyhedral Comilation Techniques.* [69]

Girbal, Sylvain, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam (June 2006). "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies". In: *Int. J. Parallel Program.* 34.3, pp. 261–317. DOI: 10.1007/s10766-006-0012-3. [8, 130]

Grosser, Tobias, Armin Größlinger, and Christian Lengauer (2012). "Polly - Performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04. DOI: 10.1142/S0129626412500107. [129]

Grosser, Tobias, Sven Verdoolaege, and Albert Cohen (July 2015). "Polyhedral AST generation is more than scanning polyhedra". In: *ACM Transactions on Programming Languages and Systems* 37.4, 12:1–12:50. DOI: 10.1145/2743016. [130, 131]

Größlinger, Armin (2009). "The Challenges of Non-linear Parameters and Variables in Automatic Loop Parallelisation". PhD thesis. Universität Passau. [69]

Guda, S. A. (2013). “Operations on the tree representations of piecewise quasi-affine functions”. In: *Inform. Primen.* 7.1, pp. 58–69. [85]

Irigoin, François (June 1987). “Partitionnement Des Boucles Imbeiquees: Une Technique D’optimisation four les Programmes Scientifiques”. PhD thesis. École nationale supérieure des mines de Paris. [129]

Irigoin, François and Rémi Triolet (Aug. 1986). *Supernodes and Alliant FX/8 Minisupercomputer.* Tech. rep. ENSMP-CAI-86-E81. [129]

Kelly, Wayne (1996). *Optimization within a Unified Transformation Framework.* Tech. rep. CS-TR-3725. Dept. of CS, Univ. of Maryland, College Park. [130]

Kelly, Wayne, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott (Nov. 1996). *The Omega Library.* Tech. rep. University of Maryland. [9]

Kelly, Wayne and William Pugh (1995). “A unifying framework for iteration reordering transformations”. In: *IEEE First International Conference on Algorithms and Architectures for Parallel Processing, 1995. ICAPP 95. IEEE First ICA/sup 3/PP.* Vol. 1, pp. 153–162. DOI: 10.1109/ICAPP.1995.472180. [130]

Kelly, Wayne, William Pugh, and Evan Rosser (1995). “Code Generation for Multiple Mappings”. In: *Frontiers ’95: The 5th Symposium on the Frontiers of Massively Parallel Computation.* McLean, VA. DOI: 10.1109/FMPC.1995.380437. [131]

Klimov, Arkady V. (July 2014). “Construction of Exact Polyhedral Model for Affine Programs with Data Dependent Conditions”. In: *Proceedings of the Fourth International Valentin Turchin Workshop on Metacomputation*, pp. 136–160. [131]

Klöckner, Andreas (2014). “Loo.Py: Transformation-based Code Generation for GPUs and CPUs”. In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming.* ARRAY’14. Edinburgh, United Kingdom: ACM, 82:82–82:87. DOI: 10.1145/2627373.2627387. [9]

LaMielle, Alan and Michelle Mills Strout (Mar. 2010). *Enabling Code Generation within the Sparse Polyhedral Framework.* Tech. rep. Technical Report CS-10-102 Colorado State University. [69]

Latour, Louis (July 2004). “From Automata to Formulas: Convex Integer Polyhedra”. In: *19th IEEE Symposium on Logic in Computer Science (LICS 2004).* IEEE Computer Society, pp. 120–129. DOI: 10.1109/LICS.2004.1319606. [68]

Leservot, Arnauld (1996). “Analyses interprocédurales du flot des données”. PhD thesis. Université Paris VI, Paris, France. [67]

Loechner, Vincent and Doran K. Wilde (Dec. 1997). “Parameterized Polyhedra and Their Vertices”. In: *International Journal of Parallel Programming* 25.6, pp. 525–549. DOI: 10.1023/A:1025117523902. [9]

Maslov, Vadim (1994). “Lazy Array Data-Flow Dependence Analysis”. In: *POPL.* Ed. by Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin. ACM Press, pp. 311–325. DOI: 10.1145/174675.177911. [40, 131]

Maslov, Vadim and William Pugh (1994). “Simplifying polynomial constraints over integers to make dependence analysis more precise”. In: *Parallel Processing: CONPAR 94—VAPP VI*, pp. 737–748. DOI: 10.1007/3-540-58430-7_64. [68]

Mehta, Sanyam and Pen-Chung Yew (2015). “Improving Compiler Scalability: Optimizing Large Programs at Small Price”. In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2015. Portland, OR, USA: ACM, pp. 143–152. DOI: 10.1145/2737924.2737954. [130]

Oppen, Derek C. (1978). “A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic”. In: *Journal of Computer and System Sciences* 16, pp. 323–332. DOI: 10.1016/0022-0000(78)90021-1. [68]

Pop, Sebastian, G.-A. Silber, Albert Cohen, Cédric Bastoul, Sylvain Girbal, and Nicolas Vasilache (2006). *GRAPHITE: Polyhedral Analyses and Optimizations for GCC*. Tech. rep. A/378/CRI. Contribution to the GNU Compilers Collection Developers Summit 2006 (GCC Summit 06), Ottawa, Canada, June 28–30, 2006. Fontainebleau, France: Centre de Recherche en Informatique, École des Mines de Paris. [8]

Presburger, M. (1929). “Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt”. In: *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*. English translation available from Stansifer (1984). Warszawa, pp. 92–101. [67]

Pugh, William (1991a). “The Omega test: a fast and practical integer programming algorithm for dependence analysis”. In: *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*. Albuquerque, New Mexico, United States: ACM Press, pp. 4–13. DOI: 10.1145/125826.125848. [67]

Pugh, William (June 1991b). “Uniform techniques for loop optimization”. In: *International Conference on Supercomputing*. Cologne, Germany, pp. 341–352. DOI: 10.1145/109025.109108. [40]

Pugh, William (Aug. 1992). “The Omega test: a fast and practical integer programming algorithm for dependence analysis”. In: *Communications of the ACM* 8, pp. 102–114. DOI: 10.1145/135226.135233. [9]

Pugh, William and David Wonnacott (1992). “Eliminating false data dependences using the Omega test”. In: *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*. San Francisco, California, United States: ACM Press, pp. 140–151. DOI: 10.1145/143095.143129. [9]

Pugh, William and David Wonnacott (1994). “An Exact Method for Analysis of Value-based Array Data Dependences”. In: *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, pp. 546–566. DOI: 10.1007/3-540-57659-2_31. [130, 151]

Pugh, William and David Wonnacott (1995). “Going beyond integer programming with the Omega test to eliminate false data dependences”. In: *Paral-*

lel and Distributed Systems, IEEE Transactions on 6.2, pp. 204–211. DOI: 10.1109/71.342135. [67]

Quilleré, Fabien, Sanjay Rajopadhye, and Doran K. Wilde (Oct. 2000). "Generation of Efficient Nested Loops from Polyhedra". In: vol. 28. 5. Springer, pp. 469–498. [131]

Quinton, Patrice (1984). "Automatic synthesis of systolic arrays from uniform recurrent equations". In: *The 11th annual international symposium on Computer architecture*, pp. 208–214. DOI: 10.1145/800015.808184. [85]

Schrijver, Alexander (1986). *Theory of Linear and Integer Programming.* John Wiley & Sons. [70]

Stansifer, Ryan (Sept. 1984). *Presburger's article on integer arithmetic: Remarks and translation.* Tech. rep. TR84-639. Cornell University. [157]

Stock, Kevin, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan (2014). "A Framework for Enhancing Data Reuse via Associative Reordering". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '14. Edinburgh, United Kingdom: ACM, pp. 65–76. DOI: 10.1145/2594291.2594342. [131]

Strout, Michelle Mills, Geri George, and Catherine Olschanowsky (Sept. 2012). "Set and Relation Manipulation for the Sparse Polyhedral Framework". In: *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing (LCPC).* DOI: 10.1007/978-3-642-37658-0_5. [69]

Trifunovic, Konrad, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta (2010). "GRAPHITE two years after: First lessons learned from real-world polyhedral compilation". In: *GCC Research Opportunities Workshop (GROW'10).* [129]

Triolet, Rémi, François Irigoin, and Paul Feautrier (July 1986). "Direct parallelization of call statements". In: *SIGPLAN Not.* 21 (7), pp. 176–185. DOI: 10.1145/13310.13329. [9]

Vandierendonck, Hans, Sean Rul, and Koen De Bosschere (2010). "The Paralax infrastructure: automatic parallelization with a helping hand". In: *Proceedings of the 19th international conference on Parallel architectures and compilation techniques.* PACT '10. Vienna, Austria: ACM, pp. 389–400. DOI: 10.1145/1854273.1854322. [151]

Vasilache, Nicolas, Cédric Bastoul, Albert Cohen, and Sylvain Girbal (2006). "Violated dependence analysis". In: *ICS '06: Proceedings of the 20th annual international conference on Supercomputing* (Cairns, Queensland, Australia). New York, NY, USA: ACM, pp. 335–344. DOI: 10.1145/1183401.1183448. [8]

Verdoolaege, Sven (2010). "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software - ICMS 2010.* Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49. [8]

Verdoolaege, Sven (Apr. 2011). “Counting Affine Calculator and Applications”. In: *First International Workshop on Polyhedral Compilation Techniques (IMPACT’11)*. Chamonix, France. [9, 40]

Verdoolaege, Sven (2013). “Polyhedral process networks”. In: *Handbook of Signal Processing Systems*. Ed. by Shuvra Bhattacharrya, Ed Deprettere, Rainer Leupers, and Jarmo Takala. 2nd ed. Springer, pp. 1335–1375. DOI: `10.1007/978-1-4614-6859-2_41`. [8]

Verdoolaege, Sven (Jan. 2015a). “Integer Set Coalescing”. In: *Proceedings of the 5th International Workshop on Polyhedral Compilation Techniques*. Amsterdam, The Netherlands. DOI: `10.13140/2.1.1313.6968`. [68]

Verdoolaege, Sven (May 2015b). *PENCIL support in pet and PPCG*. Tech. rep. RT-457, version 2. INRIA Paris-Rocquencourt. DOI: `10.13140/RG.2.1.4063.7926`. [130]

Verdoolaege, Sven and Tobias Grosser (Jan. 2012). “Polyhedral Extraction Tool”. In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT’12)*. Paris, France. DOI: `10.13140/RG.2.1.4213.4562`. [9, 131]

Verdoolaege, Sven, Serge Guelton, Tobias Grosser, and Albert Cohen (Jan. 2014). “Schedule Trees”. In: *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*. Vienna, Austria. DOI: `10.13140/RG.2.1.4475.6001`. [130]

Verdoolaege, Sven, Gerda Janssens, and Maurice Bruynooghe (June 2009). “Equivalence checking of static affine programs using widening to handle recurrences”. In: *Computer Aided Verification 21*. Springer, pp. 599–613. DOI: `10.1007/978-3-642-02658-4_44`. [8]

Verdoolaege, Sven, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor (2013). “Polyhedral parallel code generation for CUDA”. In: *ACM Trans. Archit. Code Optim.* 9.4, p. 54. DOI: `10.1145/2400682.2400713`. [8]

Verdoolaege, Sven, Hristo Nikolov, and Todor Stefanov (Jan. 2013). “On Demand Parametric Array Dataflow Analysis”. In: *Third International Workshop on Polyhedral Compilation Techniques (IMPACT’13)*. Berlin, Germany. DOI: `10.13140/RG.2.1.4737.7441`. [151]

Verdoolaege, Sven, R. Seghir, K. Beyls, Vincent Loechner, and Maurice Bruynooghe (June 2007). “Counting integer points in parametric polytopes using Barvinok’s rational functions”. In: *Algorithmica* 48.1, pp. 37–66. DOI: `10.1007/s00453-006-1231-0`.

Wilde, Doran K. (1993). *A Library for doing polyhedral operations*. Tech. rep. 785. IRISA, Rennes, France, 45 p. [9]

Yang, Yi-Qing, Corinne Ancourt, and François Irigoin (Aug. 1995). “Minimal Data Dependence Abstractions for Loop Transformations: Extended Version”. In: *Int. J. Parallel Program.* 23.4, pp. 359–388. DOI: `10.1007/BF02577771`. [130]

Yuki, Tomofumi, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat (2013). “Array dataflow analysis for polyhedral X10 programs”. In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of par-*

allel programming. PPoPP '13. Shenzhen, China: ACM, pp. 23–34. DOI: 10.1145/2442516.2442520. [151]

Yuki, Tomofumi, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye (2012). "AlphaZ: A System for Design Space Exploration in the Polyhedral Model". In: *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*. [130]

Index