Integer Set Library: Manual

Version: isl-0.06

Sven Verdoolaege

March 20, 2011

Contents

Chapter 1

User Manual

1.1 Introduction

isl is a thread-safe C library for manipulating sets and relations of integer points bounded by affine constraints. The descriptions of the sets and relations may involve both parameters and existentially quantified variables. All computations are performed in exact integer arithmetic using GMP. The isl library offers functionality that is similar to that offered by the Omega and Omega+ libraries, but the underlying algorithms are in most cases completely different.

The library is by no means complete and some fairly basic functionality is still missing. Still, even in its current form, the library has been successfully used as a backend polyhedral library for the polyhedral scanner CLooG and as part of an equivalence checker of static affine programs. For bug reports, feature requests and questions, visit the the discussion group at http://groups.google.com/group/isl-development.

1.1.1 Backward Incompatible Changes

Changes since isl-0.02

- The old printing functions have been deprecated and replaced by isl_printer functions, see Input and Output.
- Most functions related to dependence analysis have acquired an extra must argument. To obtain the old behavior, this argument should be given the value 1. See Dependence Analysis.

Changes since isl-0.03

- The function isl_pw_qpolynomial_fold_add has been renamed to isl_pw_qpolynomial_fold_fold. Similarly, isl_union_pw_qpolynomial_fold_add has been renamed to isl_union_pw_qpolynomial_fold_fol

Changes since isl-0.04

- All header files have been renamed from `isl_header.h` to `isl/header.h`.

Changes since isl-0.05

- The functions `isl_printer_print_basic_set` and `isl_printer_print_basic_map` no longer print a newline.
- The functions `isl_flow_get_no_source` and `isl_union_map_compute_flow` now return the accesses for which no source could be found instead of the iterations where those accesses occur.
- The functions `isl_basic_map_identity` and `isl_map_identity` now take the dimension specification of a **map** as input. An old call `isl_map_identity(dim)` can be rewritten to `isl_map_identity(isl_dim_map_from_set(dim))`.
- The function `isl_map_power` no longer takes a parameter position as input. Instead, the exponent is now expressed as the domain of the resulting relation.

1.2 Installation

The source of `isl` can be obtained either as a tarball or from the git repository. Both are available from http://freshmeat.net/projects/isl/. The installation process depends on how you obtained the source.

1.2.1 Installation from the git repository

1. Clone or update the repository

   The first time the source is obtained, you need to clone the repository.

   ```
   git clone git://repo.or.cz/isl.git
   ```

   To obtain updates, you need to pull in the latest changes

   ```
   git pull
   ```

2. Generate `configure`

   ```
   ./autogen.sh
   ```

After performing the above steps, continue with the Common installation instructions.

1.2.2 Common installation instructions

1. Obtain GMP

   Building isl requires GMP, including its headers files. Your distribution may not provide these header files by default and you may need to install a package called gmp-devel or something similar. Alternatively, GMP can be built from source, available from http://gmplib.org/.

2. Configure

   isl uses the standard autoconf configure script. To run it, just type

   ```
   ./configure
   ```

   optionally followed by some configure options. A complete list of options can be obtained by running

   ```
   ./configure --help
   ```

   Below we discuss some of the more common options.

   isl can optionally use piplib, but no piplib functionality is currently used by default. The --with-piplib option can be used to specify which piplib library to use, either an installed version (system), an externally built version (build) or no version (no). The option build is mostly useful in configure scripts of larger projects that bundle both isl and piplib.

   --prefix
   : Installation prefix for isl

   --with-gmp-prefix
   : Installation prefix for GMP (architecture-independent files).

   --with-gmp-exec-prefix
   : Installation prefix for GMP (architecture-dependent files).

   --with-piplib
   : Which copy of piplib to use, either no (default), system or build.

   --with-piplib-prefix
   : Installation prefix for system piplib (architecture-independent files).

   --with-piplib-exec-prefix
   : Installation prefix for system piplib (architecture-dependent files).

   --with-piplib-builddir
   : Location where build piplib was built.

3. Compile

   ```
   make
   ```

4. Install (optional)

   ```
   make install
   ```

1.3 Library

1.3.1 Initialization

All manipulations of integer sets and relations occur within the context of an `isl_ctx`. A given `isl_ctx` can only be used within a single thread. All arguments of a function are required to have been allocated within the same context. There are currently no functions available for moving an object from one `isl_ctx` to another `isl_ctx`. This means that there is currently no way of safely moving an object from one thread to another, unless the whole `isl_ctx` is moved.

An `isl_ctx` can be allocated using `isl_ctx_alloc` and freed using `isl_ctx_free`. All objects allocated within an `isl_ctx` should be freed before the `isl_ctx` itself is freed.

```
isl_ctx *isl_ctx_alloc();
void isl_ctx_free(isl_ctx *ctx);
```

1.3.2 Integers

All operations on integers, mainly the coefficients of the constraints describing the sets and relations, are performed in exact integer arithmetic using `GMP`. However, to allow future versions of `isl` to optionally support fixed integer arithmetic, all calls to `GMP` are wrapped inside `isl` specific macros. The basic type is `isl_int` and the operations below are available on this type. The meanings of these operations are essentially the same as their `GMP` `mpz_` counterparts. As always with `GMP` types, `isl_int`s need to be initialized with `isl_int_init` before they can be used and they need to be released with `isl_int_clear` after the last use. The user should not assume that an `isl_int` is represented as a `mpz_t`, but should instead explicitly convert between `mpz_t`s and `isl_int`s using `isl_int_set_gmp` and `isl_int_get_gmp` whenever a `mpz_t` is required.

isl_int_init(i)

isl_int_clear(i)

isl_int_set(r,i)

isl_int_set_si(r,i)

isl_int_set_gmp(r,g)

isl_int_get_gmp(i,g)

isl_int_abs(r,i)

isl_int_neg(r,i)

isl_int_swap(i,j)

isl_int_swap_or_set(i,j)

isl_int_add_ui(r,i,j)

isl_int_sub_ui(r,i,j)

isl_int_add(r,i,j)

isl_int_sub(r,i,j)

isl_int_mul(r,i,j)

isl_int_mul_ui(r,i,j)

isl_int_addmul(r,i,j)

isl_int_submul(r,i,j)

isl_int_gcd(r,i,j)

isl_int_lcm(r,i,j)

isl_int_divexact(r,i,j)

isl_int_cdiv_q(r,i,j)

isl_int_fdiv_q(r,i,j)

isl_int_fdiv_r(r,i,j)

isl_int_fdiv_q_ui(r,i,j)

isl_int_read(r,s)

isl_int_print(out,i,width)

isl_int_sgn(i)

isl_int_cmp(i,j)

isl_int_cmp_si(i,si)

isl_int_eq(i,j)

isl_int_ne(i,j)

isl_int_lt(i,j)

isl_int_le(i,j)

isl_int_gt(i,j)

isl_int_ge(i,j)

isl_int_abs_eq(i,j)

isl_int_abs_ne(i,j)

isl_int_abs_lt(i,j)

isl_int_abs_gt(i,j)

isl_int_abs_ge(i,j)

isl_int_is_zero(i)

isl_int_is_one(i)

isl_int_is_negone(i)

isl_int_is_pos(i)

isl_int_is_neg(i)

isl_int_is_nonpos(i)

isl_int_is_nonneg(i)

isl_int_is_divisible_by(i,j)

1.3.3 Sets and Relations

`isl` uses six types of objects for representing sets and relations, `isl_basic_set`, `isl_basic_map`, `isl_set`, `isl_map`, `isl_union_set` and `isl_union_map`. `isl_basic_set` and `isl_basic_map` represent sets and relations that can be described as a conjunction of affine constraints, while `isl_set` and `isl_map` represent unions of `isl_basic_set`s and `isl_basic_map`s, respectively. However, all `isl_basic_set`s or `isl_basic_map`s in the union need to have the same dimension. `isl_union_set`s and `isl_union_map`s represent unions of `isl_set`s or `isl_map`s of *different* dimensions, where dimensions with different space names (see Dimension Specifications) are considered different as well. The difference between sets and relations (maps) is that sets have one set of variables, while relations have two sets of variables, input variables and output variables.

1.3.4 Memory Management

Since a high-level operation on sets and/or relations usually involves several substeps and since the user is usually not interested in the intermediate results, most functions that return a new object will also release all the objects passed as arguments. If the user still wants to use one or more of these arguments after the function call, she should pass along a copy of the object rather than the object itself. The user is then responsible for making sure that the original object gets used somewhere else or is explicitly freed.

The arguments and return values of all documents functions are annotated to make clear which arguments are released and which arguments are preserved. In particular, the following annotations are used

__isl_give

__isl_give means that a new object is returned. The user should make sure that the returned pointer is used exactly once as a value for an __isl_take argument. In between, it can be used as a value for as many __isl_keep arguments as the user likes. There is one exception, and that is the case where the pointer returned is NULL. Is this case, the user is free to use it as an __isl_take argument or not.

__isl_take

__isl_take means that the object the argument points to is taken over by the function and may no longer be used by the user as an argument to any other function. The pointer value must be one returned by a function returning an __isl_give pointer. If the user passes in a NULL value, then this will be treated as an error in the sense that the function will not perform its usual operation. However, it will still make sure that all the the other __isl_take arguments are released.

__isl_keep

__isl_keep means that the function will only use the object temporarily. After the function has finished, the user can still use it as an argument to other functions. A NULL value will be treated in the same way as a NULL value for an __isl_take argument.

1.3.5 Dimension Specifications

Whenever a new set or relation is created from scratch, its dimension needs to be specified using an isl_dim.

```
#include <isl/dim.h>
__isl_give isl_dim *isl_dim_alloc(isl_ctx *ctx,
        unsigned nparam, unsigned n_in, unsigned n_out);
__isl_give isl_dim *isl_dim_set_alloc(isl_ctx *ctx,
        unsigned nparam, unsigned dim);
__isl_give isl_dim *isl_dim_copy(__isl_keep isl_dim *dim);
void isl_dim_free(__isl_take isl_dim *dim);
unsigned isl_dim_size(__isl_keep isl_dim *dim,
        enum isl_dim_type type);
```

The dimension specification used for creating a set needs to be created using isl_dim_set_alloc, while that for creating a relation needs to be created using isl_dim_alloc. isl_dim_size can be used to find out the number of dimensions of each type in a dimension specification, where type may be isl_dim_param, isl_dim_in (only for relations), isl_dim_out (only for relations), isl_dim_set (only for sets) or isl_dim_all.

It is often useful to create objects that live in the same space as some other object. This can be accomplished by creating the new objects (see Creating New Sets and Relations or Creating New (Piecewise) Quasipolynomials) based on the dimension specification of the original object.

```
#include <isl/set.h>
__isl_give isl_dim *isl_basic_set_get_dim(
        __isl_keep isl_basic_set *bset);
__isl_give isl_dim *isl_set_get_dim(__isl_keep isl_set *set);

#include <isl/union_set.h>
__isl_give isl_dim *isl_union_set_get_dim(
        __isl_keep isl_union_set *uset);

#include <isl/map.h>
__isl_give isl_dim *isl_basic_map_get_dim(
        __isl_keep isl_basic_map *bmap);
__isl_give isl_dim *isl_map_get_dim(__isl_keep isl_map *map);

#include <isl/union_map.h>
__isl_give isl_dim *isl_union_map_get_dim(
        __isl_keep isl_union_map *umap);

#include <isl/polynomial.h>
__isl_give isl_dim *isl_qpolynomial_get_dim(
        __isl_keep isl_qpolynomial *qp);
__isl_give isl_dim *isl_pw_qpolynomial_get_dim(
        __isl_keep isl_pw_qpolynomial *pwqp);
__isl_give isl_dim *isl_union_pw_qpolynomial_get_dim(
        __isl_keep isl_union_pw_qpolynomial *upwqp);
__isl_give isl_dim *isl_union_pw_qpolynomial_fold_get_dim(
        __isl_keep isl_union_pw_qpolynomial_fold *upwf);
```

The names of the individual dimensions may be set or read off using the following functions.

```
#include <isl/dim.h>
__isl_give isl_dim *isl_dim_set_name(__isl_take isl_dim *dim,
                        enum isl_dim_type type, unsigned pos,
                        __isl_keep const char *name);
__isl_keep const char *isl_dim_get_name(__isl_keep isl_dim *dim,
                        enum isl_dim_type type, unsigned pos);
```

Note that `isl_dim_get_name` returns a pointer to some internal data structure, so the result can only be used while the corresponding `isl_dim` is alive. Also note that every function that operates on two sets or relations requires that both arguments have the same parameters. This also means that if one of the arguments has named parameters, then the other needs to have named parameters too and the names need to match. Pairs of `isl_union_set` and/or `isl_union_map` arguments may have different parameters (as long as they are named), in which case the result will have as parameters the union of the parameters of the arguments.

The names of entire spaces may be set or read off using the following functions.

```
#include <isl/dim.h>
__isl_give isl_dim *isl_dim_set_tuple_name(
        __isl_take isl_dim *dim,
        enum isl_dim_type type, const char *s);
const char *isl_dim_get_tuple_name(__isl_keep isl_dim *dim,
        enum isl_dim_type type);
```

The dim argument needs to be one of isl_dim_in, isl_dim_out or isl_dim_set. As with isl_dim_get_name, the isl_dim_get_tuple_name function returns a pointer to some internal data structure. Binary operations require the corresponding spaces of their arguments to have the same name.

Spaces can be nested. In particular, the domain of a set or the domain or range of a relation can be a nested relation. The following functions can be used to construct and deconstruct such nested dimension specifications.

```
#include <isl/dim.h>
int isl_dim_is_wrapping(__isl_keep isl_dim *dim);
__isl_give isl_dim *isl_dim_wrap(__isl_take isl_dim *dim);
__isl_give isl_dim *isl_dim_unwrap(__isl_take isl_dim *dim);
```

The input to isl_dim_is_wrapping and isl_dim_unwrap should be the dimension specification of a set, while that of isl_dim_wrap should be the dimension specification of a relation. Conversely, the output of isl_dim_unwrap is the dimension specification of a relation, while that of isl_dim_wrap is the dimension specification of a set.

Dimension specifications can be created from other dimension specifications using the following functions.

```
__isl_give isl_dim *isl_dim_domain(__isl_take isl_dim *dim);
__isl_give isl_dim *isl_dim_from_domain(__isl_take isl_dim *dim);
__isl_give isl_dim *isl_dim_range(__isl_take isl_dim *dim);
__isl_give isl_dim *isl_dim_from_range(__isl_take isl_dim *dim);
__isl_give isl_dim *isl_dim_reverse(__isl_take isl_dim *dim);
__isl_give isl_dim *isl_dim_join(__isl_take isl_dim *left,
        __isl_take isl_dim *right);
__isl_give isl_dim *isl_dim_insert(__isl_take isl_dim *dim,
        enum isl_dim_type type, unsigned pos, unsigned n);
__isl_give isl_dim *isl_dim_add(__isl_take isl_dim *dim,
        enum isl_dim_type type, unsigned n);
__isl_give isl_dim *isl_dim_drop(__isl_take isl_dim *dim,
        enum isl_dim_type type, unsigned first, unsigned n);
__isl_give isl_dim *isl_dim_map_from_set(
        __isl_take isl_dim *dim);
__isl_give isl_dim *isl_dim_zip(__isl_take isl_dim *dim);
```

Note that if dimensions are added or removed from a space, then the name and the internal structure are lost.

1.3.6 Input and Output

isl supports its own input/output format, which is similar to the Omega format, but also supports the PolyLib format in some cases.

isl format

The isl format is similar to that of Omega, but has a different syntax for describing the parameters and allows for the definition of an existentially quantified variable as the integer division of an affine expression. For example, the set of integers i between 0 and n such that i % 10 <= 6 can be described as

```
[n] -> { [i] : exists (a = [i/10] : 0 <= i and i <= n and
                          i - 10 a <= 6) }
```

A set or relation can have several disjuncts, separated by the keyword or. Each disjunct is either a conjunction of constraints or a projection (exists) of a conjunction of constraints. The constraints are separated by the keyword and.

PolyLib format

If the represented set is a union, then the first line contains a single number representing the number of disjuncts. Otherwise, a line containing the number 1 is optional.

Each disjunct is represented by a matrix of constraints. The first line contains two numbers representing the number of rows and columns, where the number of rows is equal to the number of constraints and the number of columns is equal to two plus the number of variables. The following lines contain the actual rows of the constraint matrix. In each row, the first column indicates whether the constraint is an equality (0) or inequality (1). The final column corresponds to the constant term.

If the set is parametric, then the coefficients of the parameters appear in the last columns before the constant column. The coefficients of any existentially quantified variables appear between those of the set variables and those of the parameters.

Extended PolyLib format

The extended PolyLib format is nearly identical to the PolyLib format. The only difference is that the line containing the number of rows and columns of a constraint matrix also contains four additional numbers: the number of output dimensions, the number of input dimensions, the number of local dimensions (i.e., the number of existentially quantified variables) and the number of parameters. For sets, the number of "output" dimensions is equal to the number of set dimensions, while the number of "input" dimensions is zero.

Input

```
#include <isl/set.h>
__isl_give isl_basic_set *isl_basic_set_read_from_file(
```

```
        isl_ctx *ctx, FILE *input, int nparam);
__isl_give isl_basic_set *isl_basic_set_read_from_str(
        isl_ctx *ctx, const char *str, int nparam);
__isl_give isl_set *isl_set_read_from_file(isl_ctx *ctx,
        FILE *input, int nparam);
__isl_give isl_set *isl_set_read_from_str(isl_ctx *ctx,
        const char *str, int nparam);

#include <isl/map.h>
__isl_give isl_basic_map *isl_basic_map_read_from_file(
        isl_ctx *ctx, FILE *input, int nparam);
__isl_give isl_basic_map *isl_basic_map_read_from_str(
        isl_ctx *ctx, const char *str, int nparam);
__isl_give isl_map *isl_map_read_from_file(
        struct isl_ctx *ctx, FILE *input, int nparam);
__isl_give isl_map *isl_map_read_from_str(isl_ctx *ctx,
        const char *str, int nparam);

#include <isl/union_set.h>
__isl_give isl_union_set *isl_union_set_read_from_file(
        isl_ctx *ctx, FILE *input);
__isl_give isl_union_set *isl_union_set_read_from_str(
        struct isl_ctx *ctx, const char *str);

#include <isl/union_map.h>
__isl_give isl_union_map *isl_union_map_read_from_file(
        isl_ctx *ctx, FILE *input);
__isl_give isl_union_map *isl_union_map_read_from_str(
        struct isl_ctx *ctx, const char *str);
```

The input format is autodetected and may be either the `PolyLib` format or the `isl` format. `nparam` specifies how many of the final columns in the `PolyLib` format correspond to parameters. If input is given in the `isl` format, then the number of parameters needs to be equal to `nparam`. If `nparam` is negative, then any number of parameters is accepted in the `isl` format and zero parameters are assumed in the `PolyLib` format.

Output

Before anything can be printed, an `isl_printer` needs to be created.

```
__isl_give isl_printer *isl_printer_to_file(isl_ctx *ctx,
        FILE *file);
__isl_give isl_printer *isl_printer_to_str(isl_ctx *ctx);
void isl_printer_free(__isl_take isl_printer *printer);
__isl_give char *isl_printer_get_str(
        __isl_keep isl_printer *printer);
```

The behavior of the printer can be modified in various ways

```
__isl_give isl_printer *isl_printer_set_output_format(
        __isl_take isl_printer *p, int output_format);
__isl_give isl_printer *isl_printer_set_indent(
        __isl_take isl_printer *p, int indent);
__isl_give isl_printer *isl_printer_set_prefix(
        __isl_take isl_printer *p, const char *prefix);
__isl_give isl_printer *isl_printer_set_suffix(
        __isl_take isl_printer *p, const char *suffix);
```

The `output_format` may be either `ISL_FORMAT_ISL`, `ISL_FORMAT_OMEGA`, `ISL_FORMAT_POLYLIB`, `ISL_FORMAT_EXT_POLYLIB` or `ISL_FORMAT_LATEX` and defaults to `ISL_FORMAT_ISL`. Each line in the output is indented by `indent` spaces (default: 0), prefixed by `prefix` and suffixed by `suffix`. In the `PolyLib` format output, the coefficients of the existentially quantified variables appear between those of the set variables and those of the parameters.

To actually print something, use

```
#include <isl/set.h>
__isl_give isl_printer *isl_printer_print_basic_set(
        __isl_take isl_printer *printer,
        __isl_keep isl_basic_set *bset);
__isl_give isl_printer *isl_printer_print_set(
        __isl_take isl_printer *printer,
        __isl_keep isl_set *set);
```

```
#include <isl/map.h>
__isl_give isl_printer *isl_printer_print_basic_map(
        __isl_take isl_printer *printer,
        __isl_keep isl_basic_map *bmap);
__isl_give isl_printer *isl_printer_print_map(
        __isl_take isl_printer *printer,
        __isl_keep isl_map *map);
```

```
#include <isl/union_set.h>
__isl_give isl_printer *isl_printer_print_union_set(
        __isl_take isl_printer *p,
        __isl_keep isl_union_set *uset);
```

```
#include <isl/union_map.h>
__isl_give isl_printer *isl_printer_print_union_map(
        __isl_take isl_printer *p,
        __isl_keep isl_union_map *umap);
```

When called on a file printer, the following function flushes the file. When called on a string printer, the buffer is cleared.

```
__isl_give isl_printer *isl_printer_flush(
        __isl_take isl_printer *p);
```

1.3.7 Creating New Sets and Relations

isl has functions for creating some standard sets and relations.

- Empty sets and relations

```
__isl_give isl_basic_set *isl_basic_set_empty(
        __isl_take isl_dim *dim);
__isl_give isl_basic_map *isl_basic_map_empty(
        __isl_take isl_dim *dim);
__isl_give isl_set *isl_set_empty(
        __isl_take isl_dim *dim);
__isl_give isl_map *isl_map_empty(
        __isl_take isl_dim *dim);
__isl_give isl_union_set *isl_union_set_empty(
        __isl_take isl_dim *dim);
__isl_give isl_union_map *isl_union_map_empty(
        __isl_take isl_dim *dim);
```

  For isl_union_sets and isl_union_maps, the dimensions specification is only used to specify the parameters.

- Universe sets and relations

```
__isl_give isl_basic_set *isl_basic_set_universe(
        __isl_take isl_dim *dim);
__isl_give isl_basic_map *isl_basic_map_universe(
        __isl_take isl_dim *dim);
__isl_give isl_set *isl_set_universe(
        __isl_take isl_dim *dim);
__isl_give isl_map *isl_map_universe(
        __isl_take isl_dim *dim);
```

  The sets and relations constructed by the functions above contain all integer values, while those constructed by the functions below only contain non-negative values.

```
__isl_give isl_basic_set *isl_basic_set_nat_universe(
        __isl_take isl_dim *dim);
__isl_give isl_basic_map *isl_basic_map_nat_universe(
        __isl_take isl_dim *dim);
__isl_give isl_set *isl_set_nat_universe(
        __isl_take isl_dim *dim);
__isl_give isl_map *isl_map_nat_universe(
        __isl_take isl_dim *dim);
```

- Identity relations

```
__isl_give isl_basic_map *isl_basic_map_identity(
        __isl_take isl_dim *dim);
__isl_give isl_map *isl_map_identity(
        __isl_take isl_dim *dim);
```

The number of input and output dimensions in `dim` needs to be the same.

- Lexicographic order

```
__isl_give isl_map *isl_map_lex_lt(
        __isl_take isl_dim *set_dim);
__isl_give isl_map *isl_map_lex_le(
        __isl_take isl_dim *set_dim);
__isl_give isl_map *isl_map_lex_gt(
        __isl_take isl_dim *set_dim);
__isl_give isl_map *isl_map_lex_ge(
        __isl_take isl_dim *set_dim);
__isl_give isl_map *isl_map_lex_lt_first(
        __isl_take isl_dim *dim, unsigned n);
__isl_give isl_map *isl_map_lex_le_first(
        __isl_take isl_dim *dim, unsigned n);
__isl_give isl_map *isl_map_lex_gt_first(
        __isl_take isl_dim *dim, unsigned n);
__isl_give isl_map *isl_map_lex_ge_first(
        __isl_take isl_dim *dim, unsigned n);
```

The first four functions take a dimension specification for a **set** and return relations that express that the elements in the domain are lexicographically less (`isl_map_lex_lt`), less or equal (`isl_map_lex_le`), greater (`isl_map_lex_gt`) or greater or equal (`isl_map_lex_ge`) than the elements in the range. The last four functions take a dimension specification for a map and return relations that express that the first `n` dimensions in the domain are lexicographically less (`isl_map_lex_lt_first`), less or equal (`isl_map_lex_le_first`), greater (`isl_map_lex_gt_first`) or greater or equal (`isl_map_lex_ge_first`) than the first `n` dimensions in the range.

A basic set or relation can be converted to a set or relation using the following functions.

```
__isl_give isl_set *isl_set_from_basic_set(
        __isl_take isl_basic_set *bset);
__isl_give isl_map *isl_map_from_basic_map(
        __isl_take isl_basic_map *bmap);
```

Sets and relations can be converted to union sets and relations using the following functions.

```
__isl_give isl_union_map *isl_union_map_from_map(
        __isl_take isl_map *map);
__isl_give isl_union_set *isl_union_set_from_set(
        __isl_take isl_set *set);
```

Sets and relations can be copied and freed again using the following functions.

```
__isl_give isl_basic_set *isl_basic_set_copy(
        __isl_keep isl_basic_set *bset);
__isl_give isl_set *isl_set_copy(__isl_keep isl_set *set);
__isl_give isl_union_set *isl_union_set_copy(
        __isl_keep isl_union_set *uset);
__isl_give isl_basic_map *isl_basic_map_copy(
        __isl_keep isl_basic_map *bmap);
__isl_give isl_map *isl_map_copy(__isl_keep isl_map *map);
__isl_give isl_union_map *isl_union_map_copy(
        __isl_keep isl_union_map *umap);
void isl_basic_set_free(__isl_take isl_basic_set *bset);
void isl_set_free(__isl_take isl_set *set);
void isl_union_set_free(__isl_take isl_union_set *uset);
void isl_basic_map_free(__isl_take isl_basic_map *bmap);
void isl_map_free(__isl_take isl_map *map);
void isl_union_map_free(__isl_take isl_union_map *umap);
```

Other sets and relations can be constructed by starting from a universe set or relation, adding equality and/or inequality constraints and then projecting out the existentially quantified variables, if any. Constraints can be constructed, manipulated and added to basic sets and relations using the following functions.

```
#include <isl/constraint.h>
__isl_give isl_constraint *isl_equality_alloc(
        __isl_take isl_dim *dim);
__isl_give isl_constraint *isl_inequality_alloc(
        __isl_take isl_dim *dim);
void isl_constraint_set_constant(
        __isl_keep isl_constraint *constraint, isl_int v);
void isl_constraint_set_coefficient(
        __isl_keep isl_constraint *constraint,
        enum isl_dim_type type, int pos, isl_int v);
__isl_give isl_basic_map *isl_basic_map_add_constraint(
        __isl_take isl_basic_map *bmap,
        __isl_take isl_constraint *constraint);
__isl_give isl_basic_set *isl_basic_set_add_constraint(
        __isl_take isl_basic_set *bset,
        __isl_take isl_constraint *constraint);
```

For example, to create a set containing the even integers between 10 and 42, you would use the following code.

```
isl_int v;
struct isl_dim *dim;
struct isl_constraint *c;
struct isl_basic_set *bset;

isl_int_init(v);
dim = isl_dim_set_alloc(ctx, 0, 2);
bset = isl_basic_set_universe(isl_dim_copy(dim));

c = isl_equality_alloc(isl_dim_copy(dim));
isl_int_set_si(v, -1);
isl_constraint_set_coefficient(c, isl_dim_set, 0, v);
isl_int_set_si(v, 2);
isl_constraint_set_coefficient(c, isl_dim_set, 1, v);
bset = isl_basic_set_add_constraint(bset, c);

c = isl_inequality_alloc(isl_dim_copy(dim));
isl_int_set_si(v, -10);
isl_constraint_set_constant(c, v);
isl_int_set_si(v, 1);
isl_constraint_set_coefficient(c, isl_dim_set, 0, v);
bset = isl_basic_set_add_constraint(bset, c);

c = isl_inequality_alloc(dim);
isl_int_set_si(v, 42);
isl_constraint_set_constant(c, v);
isl_int_set_si(v, -1);
isl_constraint_set_coefficient(c, isl_dim_set, 0, v);
bset = isl_basic_set_add_constraint(bset, c);

bset = isl_basic_set_project_out(bset, isl_dim_set, 1, 1);

isl_int_clear(v);
```

Or, alternatively,

```
struct isl_basic_set *bset;
bset = isl_basic_set_read_from_str(ctx,
        "{[i] : exists (a : i = 2a and i >= 10 and i <= 42)}", -1);
```

A basic set or relation can also be constructed from two matrices describing the equalities and the inequalities.

```
__isl_give isl_basic_set *isl_basic_set_from_constraint_matrices(
        __isl_take isl_dim *dim,
        __isl_take isl_mat *eq, __isl_take isl_mat *ineq,
        enum isl_dim_type c1,
        enum isl_dim_type c2, enum isl_dim_type c3,
```

```
        enum isl_dim_type c4);
__isl_give isl_basic_map *isl_basic_map_from_constraint_matrices(
        __isl_take isl_dim *dim,
        __isl_take isl_mat *eq, __isl_take isl_mat *ineq,
        enum isl_dim_type c1,
        enum isl_dim_type c2, enum isl_dim_type c3,
        enum isl_dim_type c4, enum isl_dim_type c5);
```

The `isl_dim_type` arguments indicate the order in which different kinds of variables appear in the input matrices and should be a permutation of `isl_dim_cst`, `isl_dim_param`, `isl_dim_set` and `isl_dim_div` for sets and of `isl_dim_cst`, `isl_dim_param`, `isl_dim_in`, `isl_dim_out` and `isl_dim_div` for relations.

1.3.8 Inspecting Sets and Relations

Usually, the user should not have to care about the actual constraints of the sets and maps, but should instead apply the abstract operations explained in the following sections. Occasionally, however, it may be required to inspect the individual coefficients of the constraints. This section explains how to do so. In these cases, it may also be useful to have `isl` compute an explicit representation of the existentially quantified variables.

```
__isl_give isl_set *isl_set_compute_divs(
        __isl_take isl_set *set);
__isl_give isl_map *isl_map_compute_divs(
        __isl_take isl_map *map);
__isl_give isl_union_set *isl_union_set_compute_divs(
        __isl_take isl_union_set *uset);
__isl_give isl_union_map *isl_union_map_compute_divs(
        __isl_take isl_union_map *umap);
```

This explicit representation defines the existentially quantified variables as integer divisions of the other variables, possibly including earlier existentially quantified variables. An explicitly represented existentially quantified variable therefore has a unique value when the values of the other variables are known. If, furthermore, the same existentials, i.e., existentials with the same explicit representations, should appear in the same order in each of the disjuncts of a set or map, then the user should call either of the following functions.

```
__isl_give isl_set *isl_set_align_divs(
        __isl_take isl_set *set);
__isl_give isl_map *isl_map_align_divs(
        __isl_take isl_map *map);
```

Alternatively, the existentially quantified variables can be removed using the following functions, which compute an overapproximation.

```
__isl_give isl_basic_set *isl_basic_set_remove_divs(
        __isl_take isl_basic_set *bset);
__isl_give isl_basic_map *isl_basic_map_remove_divs(
        __isl_take isl_basic_map *bmap);
__isl_give isl_set *isl_set_remove_divs(
        __isl_take isl_set *set);
```

To iterate over all the sets or maps in a union set or map, use

```
int isl_union_set_foreach_set(__isl_keep isl_union_set *uset,
        int (*fn)(__isl_take isl_set *set, void *user),
        void *user);
int isl_union_map_foreach_map(__isl_keep isl_union_map *umap,
        int (*fn)(__isl_take isl_map *map, void *user),
        void *user);
```

The number of sets or maps in a union set or map can be obtained from

```
int isl_union_set_n_set(__isl_keep isl_union_set *uset);
int isl_union_map_n_map(__isl_keep isl_union_map *umap);
```

To extract the set or map from a union with a given dimension specification, use

```
__isl_give isl_set *isl_union_set_extract_set(
        __isl_keep isl_union_set *uset,
        __isl_take isl_dim *dim);
__isl_give isl_map *isl_union_map_extract_map(
        __isl_keep isl_union_map *umap,
        __isl_take isl_dim *dim);
```

To iterate over all the basic sets or maps in a set or map, use

```
int isl_set_foreach_basic_set(__isl_keep isl_set *set,
        int (*fn)(__isl_take isl_basic_set *bset, void *user),
        void *user);
int isl_map_foreach_basic_map(__isl_keep isl_map *map,
        int (*fn)(__isl_take isl_basic_map *bmap, void *user),
        void *user);
```

The callback function `fn` should return 0 if successful and -1 if an error occurs. In the latter case, or if any other error occurs, the above functions will return -1.

It should be noted that `isl` does not guarantee that the basic sets or maps passed to `fn` are disjoint. If this is required, then the user should call one of the following functions first.

```
__isl_give isl_set *isl_set_make_disjoint(
        __isl_take isl_set *set);
__isl_give isl_map *isl_map_make_disjoint(
        __isl_take isl_map *map);
```

The number of basic sets in a set can be obtained from

```
int isl_set_n_basic_set(__isl_keep isl_set *set);
```

To iterate over the constraints of a basic set or map, use

```
#include <isl/constraint.h>

int isl_basic_map_foreach_constraint(
        __isl_keep isl_basic_map *bmap,
        int (*fn)(__isl_take isl_constraint *c, void *user),
        void *user);
void isl_constraint_free(struct isl_constraint *c);
```

Again, the callback function `fn` should return 0 if successful and -1 if an error occurs. In the latter case, or if any other error occurs, the above functions will return -1. The constraint `c` represents either an equality or an inequality. Use the following function to find out whether a constraint represents an equality. If not, it represents an inequality.

```
int isl_constraint_is_equality(
        __isl_keep isl_constraint *constraint);
```

The coefficients of the constraints can be inspected using the following functions.

```
void isl_constraint_get_constant(
        __isl_keep isl_constraint *constraint, isl_int *v);
void isl_constraint_get_coefficient(
        __isl_keep isl_constraint *constraint,
        enum isl_dim_type type, int pos, isl_int *v);
```

The explicit representations of the existentially quantified variables can be inspected using the following functions. Note that the user is only allowed to use these functions if the inspected set or map is the result of a call to `isl_set_compute_divs` or `isl_map_compute_divs`.

```
__isl_give isl_div *isl_constraint_div(
        __isl_keep isl_constraint *constraint, int pos);
void isl_div_get_constant(__isl_keep isl_div *div,
        isl_int *v);
void isl_div_get_denominator(__isl_keep isl_div *div,
        isl_int *v);
void isl_div_get_coefficient(__isl_keep isl_div *div,
        enum isl_dim_type type, int pos, isl_int *v);
```

To obtain the constraints of a basic set or map in matrix form, use the following functions.

```
__isl_give isl_mat *isl_basic_set_equalities_matrix(
        __isl_keep isl_basic_set *bset,
        enum isl_dim_type c1, enum isl_dim_type c2,
        enum isl_dim_type c3, enum isl_dim_type c4);
__isl_give isl_mat *isl_basic_set_inequalities_matrix(
        __isl_keep isl_basic_set *bset,
        enum isl_dim_type c1, enum isl_dim_type c2,
        enum isl_dim_type c3, enum isl_dim_type c4);
__isl_give isl_mat *isl_basic_map_equalities_matrix(
        __isl_keep isl_basic_map *bmap,
        enum isl_dim_type c1,
        enum isl_dim_type c2, enum isl_dim_type c3,
        enum isl_dim_type c4, enum isl_dim_type c5);
__isl_give isl_mat *isl_basic_map_inequalities_matrix(
        __isl_keep isl_basic_map *bmap,
        enum isl_dim_type c1,
        enum isl_dim_type c2, enum isl_dim_type c3,
        enum isl_dim_type c4, enum isl_dim_type c5);
```

The isl_dim_type arguments dictate the order in which different kinds of variables appear in the resulting matrix and should be a permutation of isl_dim_cst, isl_dim_param, isl_dim_in, isl_dim_out and isl_dim_div.

The names of the domain and range spaces of a set or relation can be read off using the following functions.

```
const char *isl_basic_set_get_tuple_name(
        __isl_keep isl_basic_set *bset);
const char *isl_set_get_tuple_name(
        __isl_keep isl_set *set);
const char *isl_basic_map_get_tuple_name(
        __isl_keep isl_basic_map *bmap,
        enum isl_dim_type type);
const char *isl_map_get_tuple_name(
        __isl_keep isl_map *map,
        enum isl_dim_type type);
```

As with isl_dim_get_tuple_name, the value returned points to an internal data structure. The names of individual dimensions can be read off using the following functions.

```
const char *isl_constraint_get_dim_name(
        __isl_keep isl_constraint *constraint,
        enum isl_dim_type type, unsigned pos);
const char *isl_basic_set_get_dim_name(
        __isl_keep isl_basic_set *bset,
        enum isl_dim_type type, unsigned pos);
const char *isl_set_get_dim_name(
```

```
        __isl_keep isl_set *set,
        enum isl_dim_type type, unsigned pos);
const char *isl_basic_map_get_dim_name(
        __isl_keep isl_basic_map *bmap,
        enum isl_dim_type type, unsigned pos);
const char *isl_map_get_dim_name(
        __isl_keep isl_map *map,
        enum isl_dim_type type, unsigned pos);
```

These functions are mostly useful to obtain the names of the parameters.

1.3.9 Properties

Unary Properties

- Emptiness

  The following functions test whether the given set or relation contains any integer points. The "fast" variants do not perform any computations, but simply check if the given set or relation is already known to be empty.

```
int isl_basic_set_fast_is_empty(__isl_keep isl_basic_set *bset);
int isl_basic_set_is_empty(__isl_keep isl_basic_set *bset);
int isl_set_is_empty(__isl_keep isl_set *set);
int isl_union_set_is_empty(__isl_keep isl_union_set *uset);
int isl_basic_map_fast_is_empty(__isl_keep isl_basic_map *bmap);
int isl_basic_map_is_empty(__isl_keep isl_basic_map *bmap);
int isl_map_fast_is_empty(__isl_keep isl_map *map);
int isl_map_is_empty(__isl_keep isl_map *map);
int isl_union_map_is_empty(__isl_keep isl_union_map *umap);
```

- Universality

```
int isl_basic_set_is_universe(__isl_keep isl_basic_set *bset);
int isl_basic_map_is_universe(__isl_keep isl_basic_map *bmap);
int isl_set_fast_is_universe(__isl_keep isl_set *set);
```

- Single-valuedness

```
int isl_map_is_single_valued(__isl_keep isl_map *map);
```

- Bijectivity

```
int isl_map_is_bijective(__isl_keep isl_map *map);
```

- Wrapping

  The followning functions check whether the domain of the given (basic) set is a wrapped relation.

```
int isl_basic_set_is_wrapping(
        __isl_keep isl_basic_set *bset);
int isl_set_is_wrapping(__isl_keep isl_set *set);
```

- Internal Product

```
int isl_basic_map_can_zip(
        __isl_keep isl_basic_map *bmap);
int isl_map_can_zip(__isl_keep isl_map *map);
```

Check whether the product of domain and range of the given relation can be computed, i.e., whether both domain and range are nested relations.

Binary Properties

- Equality

```
int isl_set_fast_is_equal(__isl_keep isl_set *set1,
        __isl_keep isl_set *set2);
int isl_set_is_equal(__isl_keep isl_set *set1,
        __isl_keep isl_set *set2);
int isl_union_set_is_equal(
        __isl_keep isl_union_set *uset1,
        __isl_keep isl_union_set *uset2);
int isl_basic_map_is_equal(
        __isl_keep isl_basic_map *bmap1,
        __isl_keep isl_basic_map *bmap2);
int isl_map_is_equal(__isl_keep isl_map *map1,
        __isl_keep isl_map *map2);
int isl_map_fast_is_equal(__isl_keep isl_map *map1,
        __isl_keep isl_map *map2);
int isl_union_map_is_equal(
        __isl_keep isl_union_map *umap1,
        __isl_keep isl_union_map *umap2);
```

- Disjointness

```
int isl_set_fast_is_disjoint(__isl_keep isl_set *set1,
        __isl_keep isl_set *set2);
```

- Subset

```
int isl_set_is_subset(__isl_keep isl_set *set1,
        __isl_keep isl_set *set2);
int isl_set_is_strict_subset(
        __isl_keep isl_set *set1,
```

```
        __isl_keep isl_set *set2);
int isl_union_set_is_subset(
        __isl_keep isl_union_set *uset1,
        __isl_keep isl_union_set *uset2);
int isl_union_set_is_strict_subset(
        __isl_keep isl_union_set *uset1,
        __isl_keep isl_union_set *uset2);
int isl_basic_map_is_subset(
        __isl_keep isl_basic_map *bmap1,
        __isl_keep isl_basic_map *bmap2);
int isl_basic_map_is_strict_subset(
        __isl_keep isl_basic_map *bmap1,
        __isl_keep isl_basic_map *bmap2);
int isl_map_is_subset(
        __isl_keep isl_map *map1,
        __isl_keep isl_map *map2);
int isl_map_is_strict_subset(
        __isl_keep isl_map *map1,
        __isl_keep isl_map *map2);
int isl_union_map_is_subset(
        __isl_keep isl_union_map *umap1,
        __isl_keep isl_union_map *umap2);
int isl_union_map_is_strict_subset(
        __isl_keep isl_union_map *umap1,
        __isl_keep isl_union_map *umap2);
```

1.3.10 Unary Operations

- Complement

```
__isl_give isl_set *isl_set_complement(
        __isl_take isl_set *set);
```

- Inverse map

```
__isl_give isl_basic_map *isl_basic_map_reverse(
        __isl_take isl_basic_map *bmap);
__isl_give isl_map *isl_map_reverse(
        __isl_take isl_map *map);
__isl_give isl_union_map *isl_union_map_reverse(
        __isl_take isl_union_map *umap);
```

- Projection

```
__isl_give isl_basic_set *isl_basic_set_project_out(
        __isl_take isl_basic_set *bset,
```

```
        enum isl_dim_type type, unsigned first, unsigned n);
__isl_give isl_basic_map *isl_basic_map_project_out(
        __isl_take isl_basic_map *bmap,
        enum isl_dim_type type, unsigned first, unsigned n);
__isl_give isl_set *isl_set_project_out(__isl_take isl_set *set,
        enum isl_dim_type type, unsigned first, unsigned n);
__isl_give isl_map *isl_map_project_out(__isl_take isl_map *map,
        enum isl_dim_type type, unsigned first, unsigned n);
__isl_give isl_basic_set *isl_basic_map_domain(
        __isl_take isl_basic_map *bmap);
__isl_give isl_basic_set *isl_basic_map_range(
        __isl_take isl_basic_map *bmap);
__isl_give isl_set *isl_map_domain(
        __isl_take isl_map *bmap);
__isl_give isl_set *isl_map_range(
        __isl_take isl_map *map);
__isl_give isl_union_set *isl_union_map_domain(
        __isl_take isl_union_map *umap);
__isl_give isl_union_set *isl_union_map_range(
        __isl_take isl_union_map *umap);

__isl_give isl_basic_map *isl_basic_map_domain_map(
        __isl_take isl_basic_map *bmap);
__isl_give isl_basic_map *isl_basic_map_range_map(
        __isl_take isl_basic_map *bmap);
__isl_give isl_map *isl_map_domain_map(__isl_take isl_map *map);
__isl_give isl_map *isl_map_range_map(__isl_take isl_map *map);
__isl_give isl_union_map *isl_union_map_domain_map(
        __isl_take isl_union_map *umap);
__isl_give isl_union_map *isl_union_map_range_map(
        __isl_take isl_union_map *umap);
```

The functions above construct a (basic, regular or union) relation that maps (a wrapped version of) the input relation to its domain or range.

- Identity

```
__isl_give isl_map *isl_set_identity(
        __isl_take isl_set *set);
__isl_give isl_union_map *isl_union_set_identity(
        __isl_take isl_union_set *uset);
```

Construct an identity relation on the given (union) set.

- Deltas

```
__isl_give isl_basic_set *isl_basic_map_deltas(
        __isl_take isl_basic_map *bmap);
__isl_give isl_set *isl_map_deltas(__isl_take isl_map *map);
__isl_give isl_union_set *isl_union_map_deltas(
        __isl_take isl_union_map *umap);
```

These functions return a (basic) set containing the differences between image elements and corresponding domain elements in the input.

```
__isl_give isl_basic_map *isl_basic_map_deltas_map(
        __isl_take isl_basic_map *bmap);
__isl_give isl_map *isl_map_deltas_map(
        __isl_take isl_map *map);
__isl_give isl_union_map *isl_union_map_deltas_map(
        __isl_take isl_union_map *umap);
```

The functions above construct a (basic, regular or union) relation that maps (a wrapped version of) the input relation to its delta set.

- Coalescing

  Simplify the representation of a set or relation by trying to combine pairs of basic sets or relations into a single basic set or relation.

```
__isl_give isl_set *isl_set_coalesce(__isl_take isl_set *set);
__isl_give isl_map *isl_map_coalesce(__isl_take isl_map *map);
__isl_give isl_union_set *isl_union_set_coalesce(
        __isl_take isl_union_set *uset);
__isl_give isl_union_map *isl_union_map_coalesce(
        __isl_take isl_union_map *umap);
```

- Detecting equalities

```
__isl_give isl_basic_set *isl_basic_set_detect_equalities(
        __isl_take isl_basic_set *bset);
__isl_give isl_basic_map *isl_basic_map_detect_equalities(
        __isl_take isl_basic_map *bmap);
__isl_give isl_set *isl_set_detect_equalities(
        __isl_take isl_set *set);
__isl_give isl_map *isl_map_detect_equalities(
        __isl_take isl_map *map);
__isl_give isl_union_set *isl_union_set_detect_equalities(
        __isl_take isl_union_set *uset);
__isl_give isl_union_map *isl_union_map_detect_equalities(
        __isl_take isl_union_map *umap);
```

Simplify the representation of a set or relation by detecting implicit equalities.

- Convex hull

```
__isl_give isl_basic_set *isl_set_convex_hull(
        __isl_take isl_set *set);
__isl_give isl_basic_map *isl_map_convex_hull(
        __isl_take isl_map *map);
```

  If the input set or relation has any existentially quantified variables, then the result of these operations is currently undefined.

- Simple hull

```
__isl_give isl_basic_set *isl_set_simple_hull(
        __isl_take isl_set *set);
__isl_give isl_basic_map *isl_map_simple_hull(
        __isl_take isl_map *map);
__isl_give isl_union_map *isl_union_map_simple_hull(
        __isl_take isl_union_map *umap);
```

  These functions compute a single basic set or relation that contains the whole input set or relation. In particular, the output is described by translates of the constraints describing the basic sets or relations in the input.

  (See Section 2.2.)

- Affine hull

```
__isl_give isl_basic_set *isl_basic_set_affine_hull(
        __isl_take isl_basic_set *bset);
__isl_give isl_basic_set *isl_set_affine_hull(
        __isl_take isl_set *set);
__isl_give isl_union_set *isl_union_set_affine_hull(
        __isl_take isl_union_set *uset);
__isl_give isl_basic_map *isl_basic_map_affine_hull(
        __isl_take isl_basic_map *bmap);
__isl_give isl_basic_map *isl_map_affine_hull(
        __isl_take isl_map *map);
__isl_give isl_union_map *isl_union_map_affine_hull(
        __isl_take isl_union_map *umap);
```

  In case of union sets and relations, the affine hull is computed per space.

- Polyhedral hull

```
__isl_give isl_basic_set *isl_set_polyhedral_hull(
        __isl_take isl_set *set);
__isl_give isl_basic_map *isl_map_polyhedral_hull(
        __isl_take isl_map *map);
```

```
__isl_give isl_union_set *isl_union_set_polyhedral_hull(
        __isl_take isl_union_set *uset);
__isl_give isl_union_map *isl_union_map_polyhedral_hull(
        __isl_take isl_union_map *umap);
```

These functions compute a single basic set or relation not involving any existentially quantified variables that contains the whole input set or relation. In case of union sets and relations, the polyhedral hull is computed per space.

- Power

```
__isl_give isl_map *isl_map_power(__isl_take isl_map *map,
        int *exact);
__isl_give isl_union_map *isl_union_map_power(
        __isl_take isl_union_map *umap, int *exact);
```

Compute a parametric representation for all positive powers k of `map`. The result maps k to a nested relation corresponding to the kth power of `map`. The result may be an overapproximation. If the result is known to be exact, then `*exact` is set to 1.

- Transitive closure

```
__isl_give isl_map *isl_map_transitive_closure(
        __isl_take isl_map *map, int *exact);
__isl_give isl_union_map *isl_union_map_transitive_closure(
        __isl_take isl_union_map *umap, int *exact);
```

Compute the transitive closure of `map`. The result may be an overapproximation. If the result is known to be exact, then `*exact` is set to 1.

- Reaching path lengths

```
__isl_give isl_map *isl_map_reaching_path_lengths(
        __isl_take isl_map *map, int *exact);
```

Compute a relation that maps each element in the range of `map` to the lengths of all paths composed of edges in `map` that end up in the given element. The result may be an overapproximation. If the result is known to be exact, then `*exact` is set to 1. To compute the *maximal* path length, the resulting relation should be postprocessed by `isl_map_lexmax`. In particular, if the input relation is a dependence relation (mapping sources to sinks), then the maximal path length corresponds to the free schedule. Note, however, that `isl_map_lexmax` expects the maximum to be finite, so if the path lengths are unbounded (possibly due to the overapproximation), then you will get an error message.

- Wrapping

```
__isl_give isl_basic_set *isl_basic_map_wrap(
        __isl_take isl_basic_map *bmap);
__isl_give isl_set *isl_map_wrap(
        __isl_take isl_map *map);
__isl_give isl_union_set *isl_union_map_wrap(
        __isl_take isl_union_map *umap);
__isl_give isl_basic_map *isl_basic_set_unwrap(
        __isl_take isl_basic_set *bset);
__isl_give isl_map *isl_set_unwrap(
        __isl_take isl_set *set);
__isl_give isl_union_map *isl_union_set_unwrap(
        __isl_take isl_union_set *uset);
```

- Flattening

  Remove any internal structure of domain (and range) of the given set or relation. If there is any such internal structure in the input, then the name of the space is also removed.

```
__isl_give isl_basic_set *isl_basic_set_flatten(
        __isl_take isl_basic_set *bset);
__isl_give isl_set *isl_set_flatten(
        __isl_take isl_set *set);
__isl_give isl_basic_map *isl_basic_map_flatten(
        __isl_take isl_basic_map *bmap);
__isl_give isl_map *isl_map_flatten(
        __isl_take isl_map *map);

__isl_give isl_map *isl_set_flatten_map(
        __isl_take isl_set *set);
```

  The function above constructs a relation that maps the input set to a flattened version of the set.

- Internal Product

```
__isl_give isl_basic_map *isl_basic_map_zip(
        __isl_take isl_basic_map *bmap);
__isl_give isl_map *isl_map_zip(
        __isl_take isl_map *map);
__isl_give isl_union_map *isl_union_map_zip(
        __isl_take isl_union_map *umap);
```

  Given a relation with nested relations for domain and range, interchange the range of the domain with the domain of the range.

- Dimension manipulation

```
__isl_give isl_set *isl_set_add_dims(
        __isl_take isl_set *set,
        enum isl_dim_type type, unsigned n);
__isl_give isl_map *isl_map_add_dims(
        __isl_take isl_map *map,
        enum isl_dim_type type, unsigned n);
```

It is usually not advisable to directly change the (input or output) space of a set or a relation as this removes the name and the internal structure of the space. However, the above functions can be useful to add new parameters.

1.3.11 Binary Operations

The two arguments of a binary operation not only need to live in the same `isl_ctx`, they currently also need to have the same (number of) parameters.

Basic Operations

- Intersection

```
__isl_give isl_basic_set *isl_basic_set_intersect(
        __isl_take isl_basic_set *bset1,
        __isl_take isl_basic_set *bset2);
__isl_give isl_set *isl_set_intersect(
        __isl_take isl_set *set1,
        __isl_take isl_set *set2);
__isl_give isl_union_set *isl_union_set_intersect(
        __isl_take isl_union_set *uset1,
        __isl_take isl_union_set *uset2);
__isl_give isl_basic_map *isl_basic_map_intersect_domain(
        __isl_take isl_basic_map *bmap,
        __isl_take isl_basic_set *bset);
__isl_give isl_basic_map *isl_basic_map_intersect_range(
        __isl_take isl_basic_map *bmap,
        __isl_take isl_basic_set *bset);
__isl_give isl_basic_map *isl_basic_map_intersect(
        __isl_take isl_basic_map *bmap1,
        __isl_take isl_basic_map *bmap2);
__isl_give isl_map *isl_map_intersect_domain(
        __isl_take isl_map *map,
        __isl_take isl_set *set);
__isl_give isl_map *isl_map_intersect_range(
        __isl_take isl_map *map,
        __isl_take isl_set *set);
__isl_give isl_map *isl_map_intersect(
        __isl_take isl_map *map1,
```

```
        __isl_take isl_map *map2);
__isl_give isl_union_map *isl_union_map_intersect_domain(
        __isl_take isl_union_map *umap,
        __isl_take isl_union_set *uset);
__isl_give isl_union_map *isl_union_map_intersect_range(
        __isl_take isl_union_map *umap,
        __isl_take isl_union_set *uset);
__isl_give isl_union_map *isl_union_map_intersect(
        __isl_take isl_union_map *umap1,
        __isl_take isl_union_map *umap2);
```

- Union

```
__isl_give isl_set *isl_basic_set_union(
        __isl_take isl_basic_set *bset1,
        __isl_take isl_basic_set *bset2);
__isl_give isl_map *isl_basic_map_union(
        __isl_take isl_basic_map *bmap1,
        __isl_take isl_basic_map *bmap2);
__isl_give isl_set *isl_set_union(
        __isl_take isl_set *set1,
        __isl_take isl_set *set2);
__isl_give isl_map *isl_map_union(
        __isl_take isl_map *map1,
        __isl_take isl_map *map2);
__isl_give isl_union_set *isl_union_set_union(
        __isl_take isl_union_set *uset1,
        __isl_take isl_union_set *uset2);
__isl_give isl_union_map *isl_union_map_union(
        __isl_take isl_union_map *umap1,
        __isl_take isl_union_map *umap2);
```

- Set difference

```
__isl_give isl_set *isl_set_subtract(
        __isl_take isl_set *set1,
        __isl_take isl_set *set2);
__isl_give isl_map *isl_map_subtract(
        __isl_take isl_map *map1,
        __isl_take isl_map *map2);
__isl_give isl_union_set *isl_union_set_subtract(
        __isl_take isl_union_set *uset1,
        __isl_take isl_union_set *uset2);
__isl_give isl_union_map *isl_union_map_subtract(
        __isl_take isl_union_map *umap1,
        __isl_take isl_union_map *umap2);
```

- Application

```
__isl_give isl_basic_set *isl_basic_set_apply(
        __isl_take isl_basic_set *bset,
        __isl_take isl_basic_map *bmap);
__isl_give isl_set *isl_set_apply(
        __isl_take isl_set *set,
        __isl_take isl_map *map);
__isl_give isl_union_set *isl_union_set_apply(
        __isl_take isl_union_set *uset,
        __isl_take isl_union_map *umap);
__isl_give isl_basic_map *isl_basic_map_apply_domain(
        __isl_take isl_basic_map *bmap1,
        __isl_take isl_basic_map *bmap2);
__isl_give isl_basic_map *isl_basic_map_apply_range(
        __isl_take isl_basic_map *bmap1,
        __isl_take isl_basic_map *bmap2);
__isl_give isl_map *isl_map_apply_domain(
        __isl_take isl_map *map1,
        __isl_take isl_map *map2);
__isl_give isl_union_map *isl_union_map_apply_domain(
        __isl_take isl_union_map *umap1,
        __isl_take isl_union_map *umap2);
__isl_give isl_map *isl_map_apply_range(
        __isl_take isl_map *map1,
        __isl_take isl_map *map2);
__isl_give isl_union_map *isl_union_map_apply_range(
        __isl_take isl_union_map *umap1,
        __isl_take isl_union_map *umap2);
```

- Cartesian Product

```
__isl_give isl_set *isl_set_product(
        __isl_take isl_set *set1,
        __isl_take isl_set *set2);
__isl_give isl_union_set *isl_union_set_product(
        __isl_take isl_union_set *uset1,
        __isl_take isl_union_set *uset2);
__isl_give isl_basic_map *isl_basic_map_range_product(
        __isl_take isl_basic_map *bmap1,
        __isl_take isl_basic_map *bmap2);
__isl_give isl_map *isl_map_range_product(
        __isl_take isl_map *map1,
        __isl_take isl_map *map2);
__isl_give isl_union_map *isl_union_map_range_product(
        __isl_take isl_union_map *umap1,
```

```
        __isl_take isl_union_map *umap2);
__isl_give isl_map *isl_map_product(
        __isl_take isl_map *map1,
        __isl_take isl_map *map2);
__isl_give isl_union_map *isl_union_map_product(
        __isl_take isl_union_map *umap1,
        __isl_take isl_union_map *umap2);
```

The above functions compute the cross product of the given sets or relations. The domains and ranges of the results are wrapped maps between domains and ranges of the inputs. To obtain a "flat" product, use the following functions instead.

```
__isl_give isl_basic_set *isl_basic_set_flat_product(
        __isl_take isl_basic_set *bset1,
        __isl_take isl_basic_set *bset2);
__isl_give isl_set *isl_set_flat_product(
        __isl_take isl_set *set1,
        __isl_take isl_set *set2);
__isl_give isl_basic_map *isl_basic_map_flat_product(
        __isl_take isl_basic_map *bmap1,
        __isl_take isl_basic_map *bmap2);
__isl_give isl_map *isl_map_flat_product(
        __isl_take isl_map *map1,
        __isl_take isl_map *map2);
```

- Simplification

```
__isl_give isl_basic_set *isl_basic_set_gist(
        __isl_take isl_basic_set *bset,
        __isl_take isl_basic_set *context);
__isl_give isl_set *isl_set_gist(__isl_take isl_set *set,
        __isl_take isl_set *context);
__isl_give isl_union_set *isl_union_set_gist(
        __isl_take isl_union_set *uset,
        __isl_take isl_union_set *context);
__isl_give isl_basic_map *isl_basic_map_gist(
        __isl_take isl_basic_map *bmap,
        __isl_take isl_basic_map *context);
__isl_give isl_map *isl_map_gist(__isl_take isl_map *map,
        __isl_take isl_map *context);
__isl_give isl_union_map *isl_union_map_gist(
        __isl_take isl_union_map *umap,
        __isl_take isl_union_map *context);
```

The gist operation returns a set or relation that has the same intersection with the context as the input set or relation. Any implicit equality in the intersection is

made explicit in the result, while all inequalities that are redundant with respect to the intersection are removed. In case of union sets and relations, the gist operation is performed per space.

Lexicographic Optimization

Given a (basic) set `set` (or `bset`) and a zero-dimensional domain `dom`, the following functions compute a set that contains the lexicographic minimum or maximum of the elements in `set` (or `bset`) for those values of the parameters that satisfy `dom`. If `empty` is not `NULL`, then `*empty` is assigned a set that contains the parameter values in `dom` for which `set` (or `bset`) has no elements. In other words, the union of the parameter values for which the result is non-empty and of `*empty` is equal to `dom`.

```
__isl_give isl_set *isl_basic_set_partial_lexmin(
        __isl_take isl_basic_set *bset,
        __isl_take isl_basic_set *dom,
        __isl_give isl_set **empty);
__isl_give isl_set *isl_basic_set_partial_lexmax(
        __isl_take isl_basic_set *bset,
        __isl_take isl_basic_set *dom,
        __isl_give isl_set **empty);
__isl_give isl_set *isl_set_partial_lexmin(
        __isl_take isl_set *set, __isl_take isl_set *dom,
        __isl_give isl_set **empty);
__isl_give isl_set *isl_set_partial_lexmax(
        __isl_take isl_set *set, __isl_take isl_set *dom,
        __isl_give isl_set **empty);
```

Given a (basic) set `set` (or `bset`), the following functions simply return a set containing the lexicographic minimum or maximum of the elements in `set` (or `bset`). In case of union sets, the optimum is computed per space.

```
__isl_give isl_set *isl_basic_set_lexmin(
        __isl_take isl_basic_set *bset);
__isl_give isl_set *isl_basic_set_lexmax(
        __isl_take isl_basic_set *bset);
__isl_give isl_set *isl_set_lexmin(
        __isl_take isl_set *set);
__isl_give isl_set *isl_set_lexmax(
        __isl_take isl_set *set);
__isl_give isl_union_set *isl_union_set_lexmin(
        __isl_take isl_union_set *uset);
__isl_give isl_union_set *isl_union_set_lexmax(
        __isl_take isl_union_set *uset);
```

Given a (basic) relation `map` (or `bmap`) and a domain `dom`, the following functions compute a relation that maps each element of `dom` to the single lexicographic minimum

or maximum of the elements that are associated to that same element in map (or bmap). If empty is not NULL, then *empty is assigned a set that contains the elements in dom that do not map to any elements in map (or bmap). In other words, the union of the domain of the result and of *empty is equal to dom.

```
__isl_give isl_map *isl_basic_map_partial_lexmax(
        __isl_take isl_basic_map *bmap,
        __isl_take isl_basic_set *dom,
        __isl_give isl_set **empty);
__isl_give isl_map *isl_basic_map_partial_lexmin(
        __isl_take isl_basic_map *bmap,
        __isl_take isl_basic_set *dom,
        __isl_give isl_set **empty);
__isl_give isl_map *isl_map_partial_lexmax(
        __isl_take isl_map *map, __isl_take isl_set *dom,
        __isl_give isl_set **empty);
__isl_give isl_map *isl_map_partial_lexmin(
        __isl_take isl_map *map, __isl_take isl_set *dom,
        __isl_give isl_set **empty);
```

Given a (basic) map map (or bmap), the following functions simply return a map mapping each element in the domain of map (or bmap) to the lexicographic minimum or maximum of all elements associated to that element. In case of union relations, the optimum is computed per space.

```
__isl_give isl_map *isl_basic_map_lexmin(
        __isl_take isl_basic_map *bmap);
__isl_give isl_map *isl_basic_map_lexmax(
        __isl_take isl_basic_map *bmap);
__isl_give isl_map *isl_map_lexmin(
        __isl_take isl_map *map);
__isl_give isl_map *isl_map_lexmax(
        __isl_take isl_map *map);
__isl_give isl_union_map *isl_union_map_lexmin(
        __isl_take isl_union_map *umap);
__isl_give isl_union_map *isl_union_map_lexmax(
        __isl_take isl_union_map *umap);
```

1.3.12 Matrices

Matrices can be created, copied and freed using the following functions.

```
#include <isl/mat.h>
__isl_give isl_mat *isl_mat_alloc(struct isl_ctx *ctx,
        unsigned n_row, unsigned n_col);
__isl_give isl_mat *isl_mat_copy(__isl_keep isl_mat *mat);
void isl_mat_free(__isl_take isl_mat *mat);
```

Note that the elements of a newly created matrix may have arbitrary values. The elements can be changed and inspected using the following functions.

```
int isl_mat_rows(__isl_keep isl_mat *mat);
int isl_mat_cols(__isl_keep isl_mat *mat);
int isl_mat_get_element(__isl_keep isl_mat *mat,
        int row, int col, isl_int *v);
__isl_give isl_mat *isl_mat_set_element(__isl_take isl_mat *mat,
        int row, int col, isl_int v);
```

`isl_mat_get_element` will return a negative value if anything went wrong. In that case, the value of `*v` is undefined.

The following function can be used to compute the (right) inverse of a matrix, i.e., a matrix such that the product of the original and the inverse (in that order) is a multiple of the identity matrix. The input matrix is assumed to be of full row-rank.

```
__isl_give isl_mat *isl_mat_right_inverse(__isl_take isl_mat *mat);
```

The following function can be used to compute the (right) kernel (or null space) of a matrix, i.e., a matrix such that the product of the original and the kernel (in that order) is the zero matrix.

```
__isl_give isl_mat *isl_mat_right_kernel(__isl_take isl_mat *mat);
```

1.3.13 Points

Points are elements of a set. They can be used to construct simple sets (boxes) or they can be used to represent the individual elements of a set. The zero point (the origin) can be created using

```
__isl_give isl_point *isl_point_zero(__isl_take isl_dim *dim);
```

The coordinates of a point can be inspected, set and changed using

```
void isl_point_get_coordinate(__isl_keep isl_point *pnt,
        enum isl_dim_type type, int pos, isl_int *v);
__isl_give isl_point *isl_point_set_coordinate(
        __isl_take isl_point *pnt,
        enum isl_dim_type type, int pos, isl_int v);

__isl_give isl_point *isl_point_add_ui(
        __isl_take isl_point *pnt,
        enum isl_dim_type type, int pos, unsigned val);
__isl_give isl_point *isl_point_sub_ui(
        __isl_take isl_point *pnt,
        enum isl_dim_type type, int pos, unsigned val);
```

Points can be copied or freed using

```
__isl_give isl_point *isl_point_copy(
        __isl_keep isl_point *pnt);
void isl_point_free(__isl_take isl_point *pnt);
```

A singleton set can be created from a point using

```
__isl_give isl_basic_set *isl_basic_set_from_point(
        __isl_take isl_point *pnt);
__isl_give isl_set *isl_set_from_point(
        __isl_take isl_point *pnt);
```

and a box can be created from two opposite extremal points using

```
__isl_give isl_basic_set *isl_basic_set_box_from_points(
        __isl_take isl_point *pnt1,
        __isl_take isl_point *pnt2);
__isl_give isl_set *isl_set_box_from_points(
        __isl_take isl_point *pnt1,
        __isl_take isl_point *pnt2);
```

All elements of a **bounded** (union) set can be enumerated using the following functions.

```
int isl_set_foreach_point(__isl_keep isl_set *set,
        int (*fn)(__isl_take isl_point *pnt, void *user),
        void *user);
int isl_union_set_foreach_point(__isl_keep isl_union_set *uset,
        int (*fn)(__isl_take isl_point *pnt, void *user),
        void *user);
```

The function `fn` is called for each integer point in `set` with as second argument the last argument of the `isl_set_foreach_point` call. The function `fn` should return `0` on success and `-1` on failure. In the latter case, `isl_set_foreach_point` will stop enumerating and return `-1` as well. If the enumeration is performed successfully and to completion, then `isl_set_foreach_point` returns `0`.

To obtain a single point of a (basic) set, use

```
__isl_give isl_point *isl_basic_set_sample_point(
        __isl_take isl_basic_set *bset);
__isl_give isl_point *isl_set_sample_point(
        __isl_take isl_set *set);
```

If `set` does not contain any (integer) points, then the resulting point will be "void", a property that can be tested using

```
int isl_point_is_void(__isl_keep isl_point *pnt);
```

1.3.14 Piecewise Quasipolynomials

A piecewise quasipolynomial is a particular kind of function that maps a parametric point to a rational value. More specifically, a quasipolynomial is a polynomial expression in greatest integer parts of affine expressions of parameters and variables. A piecewise quasipolynomial is a subdivision of a given parametric domain into disjoint cells with a quasipolynomial associated to each cell. The value of the piecewise quasipolynomial at a given point is the value of the quasipolynomial associated to the cell that contains the point. Outside of the union of cells, the value is assumed to be zero. For example, the piecewise quasipolynomial

```
[n] -> { [x] -> ((1 + n) - x) : x <= n and x >= 0 }
```

maps `x` to `1 + n - x` for values of `x` between `0` and `n`. A given piecewise quasipolynomial has a fixed domain dimension. Union piecewise quasipolynomials are used to contain piecewise quasipolynomials defined over different domains. Piecewise quasipolynomials are mainly used by the `barvinok` library for representing the number of elements in a parametric set or map. For example, the piecewise quasipolynomial above represents the number of points in the map

```
[n] -> { [x] -> [y] : x,y >= 0 and 0 <= x + y <= n }
```

Printing (Piecewise) Quasipolynomials

Quasipolynomials and piecewise quasipolynomials can be printed using the following functions.

```
__isl_give isl_printer *isl_printer_print_qpolynomial(
        __isl_take isl_printer *p,
        __isl_keep isl_qpolynomial *qp);

__isl_give isl_printer *isl_printer_print_pw_qpolynomial(
        __isl_take isl_printer *p,
        __isl_keep isl_pw_qpolynomial *pwqp);

__isl_give isl_printer *isl_printer_print_union_pw_qpolynomial(
        __isl_take isl_printer *p,
        __isl_keep isl_union_pw_qpolynomial *upwqp);
```

The output format of the printer needs to be set to either `ISL_FORMAT_ISL` or `ISL_FORMAT_C`. For `isl_printer_print_union_pw_qpolynomial`, only `ISL_FORMAT_ISL` is supported. In case of printing in `ISL_FORMAT_C`, the user may want to set the names of all dimensions

```
__isl_give isl_qpolynomial *isl_qpolynomial_set_dim_name(
        __isl_take isl_qpolynomial *qp,
        enum isl_dim_type type, unsigned pos,
        const char *s);
```

```
__isl_give isl_pw_qpolynomial *
isl_pw_qpolynomial_set_dim_name(
        __isl_take isl_pw_qpolynomial *pwqp,
        enum isl_dim_type type, unsigned pos,
        const char *s);
```

Creating New (Piecewise) Quasipolynomials

Some simple quasipolynomials can be created using the following functions. More complicated quasipolynomials can be created by applying operations such as addition and multiplication on the resulting quasipolynomials

```
__isl_give isl_qpolynomial *isl_qpolynomial_zero(
        __isl_take isl_dim *dim);
__isl_give isl_qpolynomial *isl_qpolynomial_one(
        __isl_take isl_dim *dim);
__isl_give isl_qpolynomial *isl_qpolynomial_infty(
        __isl_take isl_dim *dim);
__isl_give isl_qpolynomial *isl_qpolynomial_neginfty(
        __isl_take isl_dim *dim);
__isl_give isl_qpolynomial *isl_qpolynomial_nan(
        __isl_take isl_dim *dim);
__isl_give isl_qpolynomial *isl_qpolynomial_rat_cst(
        __isl_take isl_dim *dim,
        const isl_int n, const isl_int d);
__isl_give isl_qpolynomial *isl_qpolynomial_div(
        __isl_take isl_div *div);
__isl_give isl_qpolynomial *isl_qpolynomial_var(
        __isl_take isl_dim *dim,
        enum isl_dim_type type, unsigned pos);
```

The zero piecewise quasipolynomial or a piecewise quasipolynomial with a single cell can be created using the following functions. Multiple of these single cell piecewise quasipolynomials can be combined to create more complicated piecewise quasipolynomials.

```
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_zero(
        __isl_take isl_dim *dim);
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_alloc(
        __isl_take isl_set *set,
        __isl_take isl_qpolynomial *qp);

__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_zero(
        __isl_take isl_dim *dim);
__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_from_pw_qpolynomial(
        __isl_take isl_pw_qpolynomial *pwqp);
__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_add_pw_qpolynomial(
```

```
        __isl_take isl_union_pw_qpolynomial *upwqp,
        __isl_take isl_pw_qpolynomial *pwqp);
```

Quasipolynomials can be copied and freed again using the following functions.

```
__isl_give isl_qpolynomial *isl_qpolynomial_copy(
        __isl_keep isl_qpolynomial *qp);
void isl_qpolynomial_free(__isl_take isl_qpolynomial *qp);

__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_copy(
        __isl_keep isl_pw_qpolynomial *pwqp);
void isl_pw_qpolynomial_free(
        __isl_take isl_pw_qpolynomial *pwqp);

__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_copy(
        __isl_keep isl_union_pw_qpolynomial *upwqp);
void isl_union_pw_qpolynomial_free(
        __isl_take isl_union_pw_qpolynomial *upwqp);
```

Inspecting (Piecewise) Quasipolynomials

To iterate over all piecewise quasipolynomials in a union piecewise quasipolynomial, use the following function

```
int isl_union_pw_qpolynomial_foreach_pw_qpolynomial(
        __isl_keep isl_union_pw_qpolynomial *upwqp,
        int (*fn)(__isl_take isl_pw_qpolynomial *pwqp, void *user),
        void *user);
```

To extract the piecewise quasipolynomial from a union with a given dimension specification, use

```
__isl_give isl_pw_qpolynomial *
isl_union_pw_qpolynomial_extract_pw_qpolynomial(
        __isl_keep isl_union_pw_qpolynomial *upwqp,
        __isl_take isl_dim *dim);
```

To iterate over the cells in a piecewise quasipolynomial, use either of the following two functions

```
int isl_pw_qpolynomial_foreach_piece(
        __isl_keep isl_pw_qpolynomial *pwqp,
        int (*fn)(__isl_take isl_set *set,
                  __isl_take isl_qpolynomial *qp,
                  void *user), void *user);
int isl_pw_qpolynomial_foreach_lifted_piece(
        __isl_keep isl_pw_qpolynomial *pwqp,
        int (*fn)(__isl_take isl_set *set,
                  __isl_take isl_qpolynomial *qp,
                  void *user), void *user);
```

As usual, the function fn should return 0 on success and -1 on failure. The difference between isl_pw_qpolynomial_foreach_piece and isl_pw_qpolynomial_foreach_lifted_piece is that isl_pw_qpolynomial_foreach_lifted_piece will first compute unique representations for all existentially quantified variables and then turn these existentially quantified variables into extra set variables, adapting the associated quasipolynomial accordingly. This means that the set passed to fn will not have any existentially quantified variables, but that the dimensions of the sets may be different for different invocations of fn.

To iterate over all terms in a quasipolynomial, use

```
int isl_qpolynomial_foreach_term(
        __isl_keep isl_qpolynomial *qp,
        int (*fn)(__isl_take isl_term *term,
                  void *user), void *user);
```

The terms themselves can be inspected and freed using these functions

```
unsigned isl_term_dim(__isl_keep isl_term *term,
        enum isl_dim_type type);
void isl_term_get_num(__isl_keep isl_term *term,
        isl_int *n);
void isl_term_get_den(__isl_keep isl_term *term,
        isl_int *d);
int isl_term_get_exp(__isl_keep isl_term *term,
        enum isl_dim_type type, unsigned pos);
__isl_give isl_div *isl_term_get_div(
        __isl_keep isl_term *term, unsigned pos);
void isl_term_free(__isl_take isl_term *term);
```

Each term is a product of parameters, set variables and integer divisions. The function isl_term_get_exp returns the exponent of a given dimensions in the given term. The isl_ints in the arguments of isl_term_get_num and isl_term_get_den need to have been initialized using isl_int_init before calling these functions.

Properties of (Piecewise) Quasipolynomials

To check whether a quasipolynomial is actually a constant, use the following function.

```
int isl_qpolynomial_is_cst(__isl_keep isl_qpolynomial *qp,
        isl_int *n, isl_int *d);
```

If qp is a constant and if n and d are not NULL then the numerator and denominator of the constant are returned in *n and *d, respectively.

Operations on (Piecewise) Quasipolynomials

```
__isl_give isl_qpolynomial *isl_qpolynomial_neg(
        __isl_take isl_qpolynomial *qp);
__isl_give isl_qpolynomial *isl_qpolynomial_add(
        __isl_take isl_qpolynomial *qp1,
        __isl_take isl_qpolynomial *qp2);
__isl_give isl_qpolynomial *isl_qpolynomial_sub(
        __isl_take isl_qpolynomial *qp1,
        __isl_take isl_qpolynomial *qp2);
__isl_give isl_qpolynomial *isl_qpolynomial_mul(
        __isl_take isl_qpolynomial *qp1,
        __isl_take isl_qpolynomial *qp2);
__isl_give isl_qpolynomial *isl_qpolynomial_pow(
        __isl_take isl_qpolynomial *qp, unsigned exponent);

__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_add(
        __isl_take isl_pw_qpolynomial *pwqp1,
        __isl_take isl_pw_qpolynomial *pwqp2);
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_sub(
        __isl_take isl_pw_qpolynomial *pwqp1,
        __isl_take isl_pw_qpolynomial *pwqp2);
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_add_disjoint(
        __isl_take isl_pw_qpolynomial *pwqp1,
        __isl_take isl_pw_qpolynomial *pwqp2);
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_neg(
        __isl_take isl_pw_qpolynomial *pwqp);
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_mul(
        __isl_take isl_pw_qpolynomial *pwqp1,
        __isl_take isl_pw_qpolynomial *pwqp2);

__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_add(
        __isl_take isl_union_pw_qpolynomial *upwqp1,
        __isl_take isl_union_pw_qpolynomial *upwqp2);
__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_sub(
        __isl_take isl_union_pw_qpolynomial *upwqp1,
        __isl_take isl_union_pw_qpolynomial *upwqp2);
__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_mul(
        __isl_take isl_union_pw_qpolynomial *upwqp1,
        __isl_take isl_union_pw_qpolynomial *upwqp2);

__isl_give isl_qpolynomial *isl_pw_qpolynomial_eval(
        __isl_take isl_pw_qpolynomial *pwqp,
        __isl_take isl_point *pnt);

__isl_give isl_qpolynomial *isl_union_pw_qpolynomial_eval(
        __isl_take isl_union_pw_qpolynomial *upwqp,
        __isl_take isl_point *pnt);
```

```
__isl_give isl_set *isl_pw_qpolynomial_domain(
        __isl_take isl_pw_qpolynomial *pwqp);
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_intersect_domain(
        __isl_take isl_pw_qpolynomial *pwpq,
        __isl_take isl_set *set);

__isl_give isl_union_set *isl_union_pw_qpolynomial_domain(
        __isl_take isl_union_pw_qpolynomial *upwqp);
__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_intersect_domain(
        __isl_take isl_union_pw_qpolynomial *upwpq,
        __isl_take isl_union_set *uset);

__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_coalesce(
        __isl_take isl_union_pw_qpolynomial *upwqp);

__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_gist(
        __isl_take isl_pw_qpolynomial *pwqp,
        __isl_take isl_set *context);

__isl_give isl_union_pw_qpolynomial *isl_union_pw_qpolynomial_gist(
        __isl_take isl_union_pw_qpolynomial *upwqp,
        __isl_take isl_union_set *context);
```

The gist operation applies the gist operation to each of the cells in the domain of the input piecewise quasipolynomial. The context is also exploited to simplify the quasipolynomials associated to each cell.

```
__isl_give isl_pw_qpolynomial *isl_pw_qpolynomial_to_polynomial(
        __isl_take isl_pw_qpolynomial *pwqp, int sign);
__isl_give isl_union_pw_qpolynomial *
isl_union_pw_qpolynomial_to_polynomial(
        __isl_take isl_union_pw_qpolynomial *upwqp, int sign);
```

Approximate each quasipolynomial by a polynomial. If `sign` is positive, the polynomial will be an overapproximation. If `sign` is negative, it will be an underapproximation. If `sign` is zero, the approximation will lie somewhere in between.

1.3.15 Bounds on Piecewise Quasipolynomials and Piecewise Quasipolynomial Reductions

A piecewise quasipolynomial reduction is a piecewise reduction (or fold) of quasipolynomials. In particular, the reduction can be maximum or a minimum. The objects are mainly used to represent the result of an upper or lower bound on a quasipolynomial over its domain, i.e., as the result of the following function.

```
__isl_give isl_pw_qpolynomial_fold *isl_pw_qpolynomial_bound(
        __isl_take isl_pw_qpolynomial *pwqp,
        enum isl_fold type, int *tight);
```

```
__isl_give isl_union_pw_qpolynomial_fold *isl_union_pw_qpolynomial_bound(
        __isl_take isl_union_pw_qpolynomial *upwqp,
        enum isl_fold type, int *tight);
```

The type argument may be either isl_fold_min or isl_fold_max. If tight is not NULL, then *tight is set to 1 is the returned bound is known be tight, i.e., for each value of the parameters there is at least one element in the domain that reaches the bound. If the domain of pwqp is not wrapping, then the bound is computed over all elements in that domain and the result has a purely parametric domain. If the domain of pwqp is wrapping, then the bound is computed over the range of the wrapped relation. The domain of the wrapped relation becomes the domain of the result.

A (piecewise) quasipolynomial reduction can be copied or freed using the following functions.

```
__isl_give isl_qpolynomial_fold *isl_qpolynomial_fold_copy(
        __isl_keep isl_qpolynomial_fold *fold);
__isl_give isl_pw_qpolynomial_fold *isl_pw_qpolynomial_fold_copy(
        __isl_keep isl_pw_qpolynomial_fold *pwf);
__isl_give isl_union_pw_qpolynomial_fold *isl_union_pw_qpolynomial_fold_copy(
        __isl_keep isl_union_pw_qpolynomial_fold *upwf);
void isl_qpolynomial_fold_free(
        __isl_take isl_qpolynomial_fold *fold);
void isl_pw_qpolynomial_fold_free(
        __isl_take isl_pw_qpolynomial_fold *pwf);
void isl_union_pw_qpolynomial_fold_free(
        __isl_take isl_union_pw_qpolynomial_fold *upwf);
```

Printing Piecewise Quasipolynomial Reductions

Piecewise quasipolynomial reductions can be printed using the following function.

```
__isl_give isl_printer *isl_printer_print_pw_qpolynomial_fold(
        __isl_take isl_printer *p,
        __isl_keep isl_pw_qpolynomial_fold *pwf);
__isl_give isl_printer *isl_printer_print_union_pw_qpolynomial_fold(
        __isl_take isl_printer *p,
        __isl_keep isl_union_pw_qpolynomial_fold *upwf);
```

For isl_printer_print_pw_qpolynomial_fold, output format of the printer needs to be set to either ISL_FORMAT_ISL or ISL_FORMAT_C. For isl_printer_print_union_pw_qpolynomial_fold, output format of the printer needs to be set to ISL_FORMAT_ISL. In case of printing in ISL_FORMAT_C, the user may want to set the names of all dimensions

```
__isl_give isl_pw_qpolynomial_fold *
isl_pw_qpolynomial_fold_set_dim_name(
        __isl_take isl_pw_qpolynomial_fold *pwf,
        enum isl_dim_type type, unsigned pos,
        const char *s);
```

Inspecting (Piecewise) Quasipolynomial Reductions

To iterate over all piecewise quasipolynomial reductions in a union piecewise quasipolynomial reduction, use the following function

```
int isl_union_pw_qpolynomial_fold_foreach_pw_qpolynomial_fold(
        __isl_keep isl_union_pw_qpolynomial_fold *upwf,
        int (*fn)(__isl_take isl_pw_qpolynomial_fold *pwf,
                  void *user), void *user);
```

To iterate over the cells in a piecewise quasipolynomial reduction, use either of the following two functions

```
int isl_pw_qpolynomial_fold_foreach_piece(
        __isl_keep isl_pw_qpolynomial_fold *pwf,
        int (*fn)(__isl_take isl_set *set,
                  __isl_take isl_qpolynomial_fold *fold,
                  void *user), void *user);
int isl_pw_qpolynomial_fold_foreach_lifted_piece(
        __isl_keep isl_pw_qpolynomial_fold *pwf,
        int (*fn)(__isl_take isl_set *set,
                  __isl_take isl_qpolynomial_fold *fold,
                  void *user), void *user);
```

See Inspecting (Piecewise) Quasipolynomials for an explanation of the difference between these two functions.

To iterate over all quasipolynomials in a reduction, use

```
int isl_qpolynomial_fold_foreach_qpolynomial(
        __isl_keep isl_qpolynomial_fold *fold,
        int (*fn)(__isl_take isl_qpolynomial *qp,
                  void *user), void *user);
```

Operations on Piecewise Quasipolynomial Reductions

```
__isl_give isl_pw_qpolynomial_fold *isl_pw_qpolynomial_fold_add(
        __isl_take isl_pw_qpolynomial_fold *pwf1,
        __isl_take isl_pw_qpolynomial_fold *pwf2);

__isl_give isl_pw_qpolynomial_fold *isl_pw_qpolynomial_fold_fold(
        __isl_take isl_pw_qpolynomial_fold *pwf1,
        __isl_take isl_pw_qpolynomial_fold *pwf2);

__isl_give isl_union_pw_qpolynomial_fold *isl_union_pw_qpolynomial_fold_fold(
        __isl_take isl_union_pw_qpolynomial_fold *upwf1,
        __isl_take isl_union_pw_qpolynomial_fold *upwf2);
```

```
__isl_give isl_qpolynomial *isl_pw_qpolynomial_fold_eval(
        __isl_take isl_pw_qpolynomial_fold *pwf,
        __isl_take isl_point *pnt);

__isl_give isl_qpolynomial *isl_union_pw_qpolynomial_fold_eval(
        __isl_take isl_union_pw_qpolynomial_fold *upwf,
        __isl_take isl_point *pnt);

__isl_give isl_union_set *isl_union_pw_qpolynomial_fold_domain(
        __isl_take isl_union_pw_qpolynomial_fold *upwf);
__isl_give isl_union_pw_qpolynomial_fold *isl_union_pw_qpolynomial_fold_intersect_do
        __isl_take isl_union_pw_qpolynomial_fold *upwf,
        __isl_take isl_union_set *uset);

__isl_give isl_pw_qpolynomial_fold *isl_pw_qpolynomial_fold_coalesce(
        __isl_take isl_pw_qpolynomial_fold *pwf);

__isl_give isl_union_pw_qpolynomial_fold *isl_union_pw_qpolynomial_fold_coalesce(
        __isl_take isl_union_pw_qpolynomial_fold *upwf);

__isl_give isl_pw_qpolynomial_fold *isl_pw_qpolynomial_fold_gist(
        __isl_take isl_pw_qpolynomial_fold *pwf,
        __isl_take isl_set *context);

__isl_give isl_union_pw_qpolynomial_fold *isl_union_pw_qpolynomial_fold_gist(
        __isl_take isl_union_pw_qpolynomial_fold *upwf,
        __isl_take isl_union_set *context);
```

The gist operation applies the gist operation to each of the cells in the domain of the input piecewise quasipolynomial reduction. In future, the operation will also exploit the context to simplify the quasipolynomial reductions associated to each cell.

```
__isl_give isl_pw_qpolynomial_fold *
isl_set_apply_pw_qpolynomial_fold(
        __isl_take isl_set *set,
        __isl_take isl_pw_qpolynomial_fold *pwf,
        int *tight);
__isl_give isl_pw_qpolynomial_fold *
isl_map_apply_pw_qpolynomial_fold(
        __isl_take isl_map *map,
        __isl_take isl_pw_qpolynomial_fold *pwf,
        int *tight);
__isl_give isl_union_pw_qpolynomial_fold *
isl_union_set_apply_union_pw_qpolynomial_fold(
        __isl_take isl_union_set *uset,
        __isl_take isl_union_pw_qpolynomial_fold *upwf,
        int *tight);
```

```
__isl_give isl_union_pw_qpolynomial_fold *
isl_union_map_apply_union_pw_qpolynomial_fold(
        __isl_take isl_union_map *umap,
        __isl_take isl_union_pw_qpolynomial_fold *upwf,
        int *tight);
```

The functions taking a map compose the given map with the given piecewise quasipolynomial reduction. That is, compute a bound (of the same type as `pwf` or `upwf` itself) over all elements in the intersection of the range of the map and the domain of the piecewise quasipolynomial reduction as a function of an element in the domain of the map. The functions taking a set compute a bound over all elements in the intersection of the set and the domain of the piecewise quasipolynomial reduction.

1.3.16 Dependence Analysis

`isl` contains specialized functionality for performing array dataflow analysis. That is, given a *sink* access relation and a collection of possible *source* access relations, `isl` can compute relations that describe for each iteration of the sink access, which iteration of which of the source access relations was the last to access the same data element before the given iteration of the sink access. To compute standard flow dependences, the sink should be a read, while the sources should be writes. If any of the source accesses are marked as being *may* accesses, then there will be a dependence to the last *must* access **and** to any *may* access that follows this last *must* access. In particular, if *all* sources are *may* accesses, then memory based dependence analysis is performed. If, on the other hand, all sources are *must* accesses, then value based dependence analysis is performed.

```
#include <isl/flow.h>

typedef int (*isl_access_level_before)(void *first, void *second);

__isl_give isl_access_info *isl_access_info_alloc(
        __isl_take isl_map *sink,
        void *sink_user, isl_access_level_before fn,
        int max_source);
__isl_give isl_access_info *isl_access_info_add_source(
        __isl_take isl_access_info *acc,
        __isl_take isl_map *source, int must,
        void *source_user);
void isl_access_info_free(__isl_take isl_access_info *acc);

__isl_give isl_flow *isl_access_info_compute_flow(
        __isl_take isl_access_info *acc);

int isl_flow_foreach(__isl_keep isl_flow *deps,
        int (*fn)(__isl_take isl_map *dep, int must,
                  void *dep_user, void *user),
```

```
        void *user);
__isl_give isl_map *isl_flow_get_no_source(
        __isl_keep isl_flow *deps, int must);
void isl_flow_free(__isl_take isl_flow *deps);
```

The function `isl_access_info_compute_flow` performs the actual dependence analysis. The other functions are used to construct the input for this function or to read off the output.

The input is collected in an `isl_access_info`, which can be created through a call to `isl_access_info_alloc`. The arguments to this functions are the sink access relation `sink`, a token `sink_user` used to identify the sink access to the user, a callback function for specifying the relative order of source and sink accesses, and the number of source access relations that will be added. The callback function has type `int (*)(void *first, void *second)`. The function is called with two user supplied tokens identifying either a source or the sink and it should return the shared nesting level and the relative order of the two accesses. In particular, let *n* be the number of loops shared by the two accesses. If `first` precedes `second` textually, then the function should return *2 * n + 1*; otherwise, it should return *2 * n*. The sources can be added to the `isl_access_info` by performing (at most) `max_source` calls to `isl_access_info_add_source`. `must` indicates whether the source is a *must* access or a *may* access. Note that a multi-valued access relation should only be marked *must* if every iteration in the domain of the relation accesses *all* elements in its image. The `source_user` token is again used to identify the source access. The range of the source access relation `source` should have the same dimension as the range of the sink access relation. The `isl_access_info_free` function should usually not be called explicitly, because it is called implicitly by `isl_access_info_compute_flow`.

The result of the dependence analysis is collected in an `isl_flow`. There may be elements of the sink access for which no preceding source access could be found or for which all preceding sources are *may* accesses. The relations containing these elements can be obtained through calls to `isl_flow_get_no_source`, the first with `must` set and the second with `must` unset. In the case of standard flow dependence analysis, with the sink a read and the sources *must* writes, the first relation corresponds to the reads from uninitialized array elements and the second relation is empty. The actual flow dependences can be extracted using `isl_flow_foreach`. This function will call the user-specified callback function `fn` for each **non-empty** dependence between a source and the sink. The callback function is called with four arguments, the actual flow dependence relation mapping source iterations to sink iterations, a boolean that indicates whether it is a *must* or *may* dependence, a token identifying the source and an additional `void *` with value equal to the third argument of the `isl_flow_foreach` call. A dependence is marked *must* if it originates from a *must* source and if it is not followed by any *may* sources.

After finishing with an `isl_flow`, the user should call `isl_flow_free` to free all associated memory.

A higher-level interface to dependence analysis is provided by the following function.

```
#include <isl/flow.h>
```

```
int isl_union_map_compute_flow(__isl_take isl_union_map *sink,
        __isl_take isl_union_map *must_source,
        __isl_take isl_union_map *may_source,
        __isl_take isl_union_map *schedule,
        __isl_give isl_union_map **must_dep,
        __isl_give isl_union_map **may_dep,
        __isl_give isl_union_map **must_no_source,
        __isl_give isl_union_map **may_no_source);
```

The arrays are identified by the tuple names of the ranges of the accesses. The iteration domains by the tuple names of the domains of the accesses and of the schedule. The relative order of the iteration domains is given by the schedule. The relations returned through `must_no_source` and `may_no_source` are subsets of `sink`. Any of `must_dep`, `may_dep`, `must_no_source` or `may_no_source` may be `NULL`, but a `NULL` value for any of the other arguments is treated as an error.

1.3.17 Parametric Vertex Enumeration

The parametric vertex enumeration described in this section is mainly intended to be used internally and by the `barvinok` library.

```
#include <isl/vertices.h>
__isl_give isl_vertices *isl_basic_set_compute_vertices(
        __isl_keep isl_basic_set *bset);
```

The function `isl_basic_set_compute_vertices` performs the actual computation of the parametric vertices and the chamber decomposition and store the result in an `isl_vertices` object. This information can be queried by either iterating over all the vertices or iterating over all the chambers or cells and then iterating over all vertices that are active on the chamber.

```
int isl_vertices_foreach_vertex(
        __isl_keep isl_vertices *vertices,
        int (*fn)(__isl_take isl_vertex *vertex, void *user),
        void *user);

int isl_vertices_foreach_cell(
        __isl_keep isl_vertices *vertices,
        int (*fn)(__isl_take isl_cell *cell, void *user),
        void *user);
int isl_cell_foreach_vertex(__isl_keep isl_cell *cell,
        int (*fn)(__isl_take isl_vertex *vertex, void *user),
        void *user);
```

Other operations that can be performed on an `isl_vertices` object are the following.

```
isl_ctx *isl_vertices_get_ctx(
        __isl_keep isl_vertices *vertices);
int isl_vertices_get_n_vertices(
        __isl_keep isl_vertices *vertices);
void isl_vertices_free(__isl_take isl_vertices *vertices);
```

Vertices can be inspected and destroyed using the following functions.

```
isl_ctx *isl_vertex_get_ctx(__isl_keep isl_vertex *vertex);
int isl_vertex_get_id(__isl_keep isl_vertex *vertex);
__isl_give isl_basic_set *isl_vertex_get_domain(
        __isl_keep isl_vertex *vertex);
__isl_give isl_basic_set *isl_vertex_get_expr(
        __isl_keep isl_vertex *vertex);
void isl_vertex_free(__isl_take isl_vertex *vertex);
```

`isl_vertex_get_expr` returns a singleton parametric set describing the vertex, while `isl_vertex_get_domain` returns the activity domain of the vertex. Note that `isl_vertex_get_domain` and `isl_vertex_get_expr` return **rational** basic sets, so they should mainly be used for inspection and should not be mixed with integer sets.

Chambers can be inspected and destroyed using the following functions.

```
isl_ctx *isl_cell_get_ctx(__isl_keep isl_cell *cell);
__isl_give isl_basic_set *isl_cell_get_domain(
        __isl_keep isl_cell *cell);
void isl_cell_free(__isl_take isl_cell *cell);
```

1.4 Applications

Although `isl` is mainly meant to be used as a library, it also contains some basic applications that use some of the functionality of `isl`. The input may be specified in either the isl format or the PolyLib format.

1.4.1 `isl_polyhedron_sample`

`isl_polyhedron_sample` takes a polyhedron as input and prints an integer element of the polyhedron, if there is any. The first column in the output is the denominator and is always equal to 1. If the polyhedron contains no integer points, then a vector of length zero is printed.

1.4.2 `isl_pip`

`isl_pip` takes the same input as the `example` program from the `piplib` distribution, i.e., a set of constraints on the parameters, a line containing only -1 and finally a set of constraints on a parametric polyhedron. The coefficients of the parameters appear in the last columns (but before the final constant column). The output is the lexicographic

minimum of the parametric polyhedron. As isl currently does not have its own output format, the output is just a dump of the internal state.

1.4.3 isl_polyhedron_minimize

isl_polyhedron_minimize computes the minimum of some linear or affine objective function over the integer points in a polyhedron. If an affine objective function is given, then the constant should appear in the last column.

1.4.4 isl_polytope_scan

Given a polytope, isl_polytope_scan prints all integer points in the polytope.

1.5 isl-polylib

The isl-polylib library provides the following functions for converting between isl objects and PolyLib objects. The library is distributed separately for licensing reasons.

```
#include <isl_set_polylib.h>
__isl_give isl_basic_set *isl_basic_set_new_from_polylib(
        Polyhedron *P, __isl_take isl_dim *dim);
Polyhedron *isl_basic_set_to_polylib(
        __isl_keep isl_basic_set *bset);
__isl_give isl_set *isl_set_new_from_polylib(Polyhedron *D,
        __isl_take isl_dim *dim);
Polyhedron *isl_set_to_polylib(__isl_keep isl_set *set);

#include <isl_map_polylib.h>
__isl_give isl_basic_map *isl_basic_map_new_from_polylib(
        Polyhedron *P, __isl_take isl_dim *dim);
__isl_give isl_map *isl_map_new_from_polylib(Polyhedron *D,
        __isl_take isl_dim *dim);
Polyhedron *isl_basic_map_to_polylib(
        __isl_keep isl_basic_map *bmap);
Polyhedron *isl_map_to_polylib(__isl_keep isl_map *map);
```

Chapter 2

Implementation Details

2.1 Sets and Relations

Definition 2.1.1 (Polyhedral Set) *A* polyhedral set *S is a finite union of basic sets $S = \bigcup_i S_i$, each of which can be represented using affine constraints*

$$S_i : \mathbb{Z}^n \to 2^{\mathbb{Z}^d} : \mathbf{s} \mapsto S_i(\mathbf{s}) = \{\, \mathbf{x} \in \mathbb{Z}^d \mid \exists \mathbf{z} \in \mathbb{Z}^e : A\mathbf{x} + B\mathbf{s} + D\mathbf{z} + \mathbf{c} \geq \mathbf{0} \,\},$$

with $A \in \mathbb{Z}^{m \times d}$, $B \in \mathbb{Z}^{m \times n}$, $D \in \mathbb{Z}^{m \times e}$ and $\mathbf{c} \in \mathbb{Z}^m$.

Definition 2.1.2 (Parameter Domain of a Set) *Let $S \in \mathbb{Z}^n \to 2^{\mathbb{Z}^d}$ be a set. The* parameter domain *of S is the set*

$$\operatorname{pdom} S \coloneqq \{\, \mathbf{s} \in \mathbb{Z}^n \mid S(\mathbf{s}) \neq \emptyset \,\}.$$

Definition 2.1.3 (Polyhedral Relation) *A* polyhedral relation *R is a finite union of basic relations $R = \bigcup_i R_i$ of type $\mathbb{Z}^n \to 2^{\mathbb{Z}^{d_1 + d_2}}$, each of which can be represented using affine constraints*

$$R_i = \mathbf{s} \mapsto R_i(\mathbf{s}) = \{\, \mathbf{x}_1 \to \mathbf{x}_2 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \exists \mathbf{z} \in \mathbb{Z}^e : A_1\mathbf{x}_1 + A_2\mathbf{x}_2 + B\mathbf{s} + D\mathbf{z} + \mathbf{c} \geq \mathbf{0} \,\},$$

with $A_i \in \mathbb{Z}^{m \times d_i}$, $B \in \mathbb{Z}^{m \times n}$, $D \in \mathbb{Z}^{m \times e}$ and $\mathbf{c} \in \mathbb{Z}^m$.

Definition 2.1.4 (Parameter Domain of a Relation) *Let $R \in \mathbb{Z}^n \to 2^{\mathbb{Z}^{d+d}}$ be a relation. The* parameter domain *of R is the set*

$$\operatorname{pdom} R \coloneqq \{\, \mathbf{s} \in \mathbb{Z}^n \mid R(\mathbf{s}) \neq \emptyset \,\}.$$

Definition 2.1.5 (Domain of a Relation) *Let $R \in \mathbb{Z}^n \to 2^{\mathbb{Z}^{d+d}}$ be a relation. The* domain *of R is the polyhedral set*

$$\operatorname{dom} R \coloneqq \mathbf{s} \mapsto \{\, \mathbf{x}_1 \in \mathbb{Z}^{d_1} \mid \exists \mathbf{x}_2 \in \mathbb{Z}^{d_2} : (\mathbf{x}_1, \mathbf{x}_2) \in R(\mathbf{s}) \,\}.$$

Definition 2.1.6 (Range of a Relation) *Let $R \in \mathbb{Z}^n \to 2^{\mathbb{Z}^{d+d}}$ be a relation. The* range *of R is the polyhedral set*

$$\operatorname{ran} R \coloneqq \mathbf{s} \mapsto \{\, \mathbf{x}_2 \in \mathbb{Z}^{d_2} \mid \exists \mathbf{x}_1 \in \mathbb{Z}^{d_1} : (\mathbf{x}_1, \mathbf{x}_2) \in R(\mathbf{s}) \,\}.$$

Definition 2.1.7 (Composition of Relations) *Let $R \in \mathbb{Z}^n \to 2^{\mathbb{Z}^{d_1+d_2}}$ and $S \in \mathbb{Z}^n \to 2^{\mathbb{Z}^{d_2+d_3}}$ be two relations, then the composition of R and S is defined as*

$$S \circ R \coloneqq \mathbf{s} \mapsto \{\, \mathbf{x}_1 \to \mathbf{x}_3 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_3} \mid \exists \mathbf{x}_2 \in \mathbb{Z}^{d_2} : \mathbf{x}_1 \to \mathbf{x}_2 \in R(\mathbf{s}) \wedge \mathbf{x}_2 \to \mathbf{x}_3 \in S(\mathbf{s}) \,\}.$$

Definition 2.1.8 (Difference Set of a Relation) *Let $R \in \mathbb{Z}^n \to 2^{\mathbb{Z}^{d+d}}$ be a relation. The difference set (ΔR) of R is the set of differences between image elements and the corresponding domain elements,*

$$\Delta R \coloneqq \mathbf{s} \mapsto \{\, \boldsymbol{\delta} \in \mathbb{Z}^d \mid \exists \mathbf{x} \to \mathbf{y} \in R : \boldsymbol{\delta} = \mathbf{y} - \mathbf{x} \,\}$$

2.2 Simple Hull

It is sometimes useful to have a single basic set or basic relation that contains a given set or relation. For rational sets, the obvious choice would be to compute the (rational) convex hull. For integer sets, the obvious choice would be the integer hull. However, `isl` currently does not support an integer hull operation and even if it did, it would be fairly expensive to compute. The convex hull operation is supported, but it is also fairly expensive to compute given only an implicit representation.

Usually, it is not required to compute the exact integer hull, and an overapproximation of this hull is sufficient. The "simple hull" of a set is such an overapproximation and it is defined as the (inclusion-wise) smallest basic set that is described by constraints that are translates of the constraints in the input set. This means that the simple hull is relatively cheap to compute and that the number of constraints in the simple hull is no larger than the number of constraints in the input.

Definition 2.2.1 (Simple Hull of a Set) *The* simple hull *of a set $S = \bigcup_{1 \le i \le v} S_i$, with*

$$S : \mathbb{Z}^n \to 2^{\mathbb{Z}^d} : \mathbf{s} \mapsto S(\mathbf{s}) = \left\{ \mathbf{x} \in \mathbb{Z}^d \mid \exists \mathbf{z} \in \mathbb{Z}^e : \bigvee_{1 \le i \le v} A_i \mathbf{x} + B_i \mathbf{s} + D_i \mathbf{z} + \mathbf{c}_i \ge \mathbf{0} \right\}$$

is the set

$$H : \mathbb{Z}^n \to 2^{\mathbb{Z}^d} : \mathbf{s} \mapsto S(\mathbf{s}) = \left\{ \mathbf{x} \in \mathbb{Z}^d \mid \exists \mathbf{z} \in \mathbb{Z}^e : \bigwedge_{1 \le i \le v} A_i \mathbf{x} + B_i \mathbf{s} + D_i \mathbf{z} + \mathbf{c}_i + \mathbf{K}_i \ge \mathbf{0} \right\},$$

with $\mathbf{K}_i$ the (component-wise) smallest non-negative integer vectors such that $S \subseteq H$.

The $\mathbf{K}_i$ can be obtained by solving a number of LP problems, one for each element of each $\mathbf{K}_i$. If any LP problem is unbounded, then the corresponding constraint is dropped.

2.3 Parametric Integer Programming

2.3.1 Introduction

Parametric integer programming (Feautrier 1988) is used to solve many problems within the context of the polyhedral model. Here, we are mainly interested in dependence analysis (Feautrier 1991) and in computing a unique representation for existentially quantified variables. The latter operation has been used for counting elements in sets involving such variables (Boulet and Redon 1998; Verdoolaege et al. 2005) and lies at the core of the internal representation of `isl`.

Parametric integer programming was first implemented in `PipLib`. An alternative method for parametric integer programming was later implemented in `barvinok` (Verdoolaege 2006). This method is not based on Feautrier's algorithm, but on rational generating functions (Barvinok and Woods 2003) and was inspired by the "digging" technique of De Loera et al. (2004) for solving non-parametric integer programming problems.

In the following sections, we briefly recall the dual simplex method combined with Gomory cuts and describe some extensions and optimizations. The main algorithm is applied to a matrix data structure known as a tableau. In case of parametric problems, there are two tableaus, one for the main problem and one for the constraints on the parameters, known as the context tableau. The handling of the context tableau is described in Section 2.3.7.

2.3.2 The Dual Simplex Method

Tableaus can be represented in several slightly different ways. In `isl`, the dual simplex method uses the same representation as that used by its incremental LP solver based on the *primal* simplex method. The implementation of this LP solver is based on that of `Simplify` (Detlefs et al. 2005), which, in turn, was derived from the work of Nelson (1980). In the original (Nelson 1980), the tableau was implemented as a sparse matrix, but neither `Simplify` nor the current implementation of `isl` does so.

Given some affine constraints on the variables, $A\mathbf{x} + \mathbf{b} \geq \mathbf{0}$, the tableau represents the relationship between the variables $\mathbf{x}$ and non-negative variables $\mathbf{y} = A\mathbf{x} + \mathbf{b}$ corresponding to the constraints. The initial tableau contains $\begin{pmatrix}\mathbf{b} & A\end{pmatrix}$ and expresses the constraints $\mathbf{y}$ in the rows in terms of the variables $\mathbf{x}$ in the columns. The main operation defined on a tableau exchanges a column and a row variable and is called a pivot. During this process, some coefficients may become rational. As in the `PipLib` implementation, `isl` maintains a shared denominator per row. The sample value of a tableau is one where each column variable is assigned zero and each row variable is assigned the constant term of the row. This sample value represents a valid solution if each constraint variable is assigned a non-negative value, i.e., if the constant terms of rows corresponding to constraints are all non-negative.

The dual simplex method starts from an initial sample value that may be invalid, but that is known to be (lexicographically) no greater than any solution, and gradually increments this sample value through pivoting until a valid solution is obtained. In particular, each pivot exchanges a row variable $r = -n + \sum_i a_i c_i$ with negative sample

value $-n$ with a column variable c_j such that $a_j > 0$. Since $c_j = (n + r - \sum_{i \neq j} a_i c_i)/a_j$, the new row variable will have a positive sample value n. If no such column can be found, then the problem is infeasible. By always choosing the column that leads to the (lexicographically) smallest increment in the variables $\mathbf{x}$, the first solution found is guaranteed to be the (lexicographically) minimal solution (Feautrier 1988). In order to be able to determine the smallest increment, the tableau is (implicitly) extended with extra rows defining the original variables in terms of the column variables. If we assume that all variables are non-negative, then we know that the zero vector is no greater than the minimal solution and then the initial extended tableau looks as follows.

$$\begin{array}{c} \\ \mathbf{x} \\ \mathbf{r} \end{array} \begin{array}{c} \begin{array}{cc} 1 & \mathbf{c} \end{array} \\ \begin{pmatrix} \mathbf{0} & I \\ \mathbf{b} & A \end{pmatrix} \end{array}$$

Each column in this extended tableau is lexicographically positive and will remain so because of the column choice explained above. It is then clear that the value of $\mathbf{x}$ will increase in each step. Note that there is no need to store the extra rows explicitly. If a given x_i is a column variable, then the corresponding row is the unit vector e_i. If, on the other hand, it is a row variable, then the row already appears somewhere else in the tableau.

In case of parametric problems, the sign of the constant term may depend on the parameters. Each time the constant term of a constraint row changes, we therefore need to check whether the new term can attain negative and/or positive values over the current set of possible parameter values, i.e., the context. If all these terms can only attain non-negative values, the current state of the tableau represents a solution. If one of the terms can only attain non-positive values and is not identically zero, the corresponding row can be pivoted. Otherwise, we pick one of the terms that can attain both positive and negative values and split the context into a part where it only attains non-negative values and a part where it only attains negative values.

2.3.3 Gomory Cuts

The solution found by the dual simplex method may have non-integral coordinates. If so, some rational solutions (including the current sample value), can be cut off by applying a (parametric) Gomory cut. Let $r = b(\mathbf{p}) + \langle \mathbf{a}, \mathbf{c} \rangle$ be the row corresponding to the first non-integral coordinate of $\mathbf{x}$, with $b(\mathbf{p})$ the constant term, an affine expression in the parameters $\mathbf{p}$, i.e., $b(\mathbf{p}) = \langle \mathbf{f}, \mathbf{p} \rangle + g$. Note that only row variables can attain non-integral values as the sample value of the column variables is zero. Consider the expression $b(\mathbf{p}) - \lceil b(\mathbf{p}) \rceil + \langle \{\mathbf{a}\}, \mathbf{c} \rangle$, with $\lceil \cdot \rceil$ the ceiling function and $\{\cdot\}$ the fractional part. This expression is negative at the sample value since $\mathbf{c} = \mathbf{0}$ and $r = b(\mathbf{p})$ is fractional, i.e., $\lceil b(\mathbf{p}) \rceil > b(\mathbf{p})$. On the other hand, for each integral value of r and $\mathbf{c} \geq 0$, the expression is non-negative because $b(\mathbf{p}) - \lceil b(\mathbf{p}) \rceil > -1$. Imposing this expression to be non-negative therefore does not invalidate any integral solutions, while it does cut away the current fractional sample value. To be able to formulate this constraint, a new variable $q = \lfloor -b(\mathbf{p}) \rfloor = -\lceil b(\mathbf{p}) \rceil$ is added to the context. This integral variable

is uniquely defined by the constraints $0 \leq -d\,b(\mathbf{p}) - d\,q \leq d - 1$, with d the common denominator of $\mathbf{f}$ and g. In practice, the variable $q' = \lfloor \langle \{-f\}, \mathbf{p} \rangle + \{-g\} \rfloor$ is used instead and the coefficients of the new constraint are adjusted accordingly. The sign of the constant term of this new constraint need not be determined as it is non-positive by construction. When several of these extra context variables are added, it is important to avoid adding duplicates. Recent versions of `PipLib` also check for such duplicates.

2.3.4 Negative Unknowns and Maximization

There are two places in the above algorithm where the unknowns $\mathbf{x}$ are assumed to be non-negative: the initial tableau starts from sample value $\mathbf{x} = \mathbf{0}$ and $\mathbf{c}$ is assumed to be non-negative during the construction of Gomory cuts. To deal with negative unknowns, Feautrier (1991, Appendix A.2) proposed to use a "big parameter", say M, that is taken to be an arbitrarily large positive number. Instead of looking for the lexicographically minimal value of $\mathbf{x}$, we search instead for the lexicographically minimal value of $\mathbf{x}' = \mathbf{M} + \mathbf{x}$. The sample value $\mathbf{x}' = \mathbf{0}$ of the initial tableau then corresponds to $\mathbf{x} = -\mathbf{M}$, which is clearly not greater than any potential solution. The sign of the constant term of a row is determined lexicographically, with the coefficient of M considered first. That is, if the coefficient of M is not zero, then its sign is the sign of the entire term. Otherwise, the sign is determined by the remaining affine expression in the parameters. If the original problem has a bounded optimum, then the final sample value will be of the form $\mathbf{M} + \mathbf{v}$ and the optimal value of the original problem is then $\mathbf{v}$. Maximization problems can be handled in a similar way by computing the minimum of $\mathbf{M} - \mathbf{x}$.

When the optimum is unbounded, the optimal value computed for the original problem will involve the big parameter. In the original implementation of `PipLib`, the big parameter could even appear in some of the extra variables $\mathbf{q}$ created during the application of a Gomory cut. The final result could then contain implicit conditions on the big parameter through conditions on such $\mathbf{q}$ variables. This problem was resolved in later versions of `PipLib` by taking M to be divisible by any positive number. The big parameter can then never appear in any $\mathbf{q}$ because $\{\alpha M\} = 0$. It should be noted, though, that an unbounded problem usually (but not always) indicates an incorrect formulation of the problem.

The original version of `PipLib` required the user to "manually" add a big parameter, perform the reformulation and interpret the result (Feautrier et al. 2002). Recent versions allow the user to simply specify that the unknowns may be negative or that the maximum should be computed and then these transformations are performed internally. Although there are some application, e.g., that of Feautrier (1992), where it is useful to have explicit control over the big parameter, negative unknowns and maximization are by far the most common applications of the big parameter and we believe that the user should not be bothered with such implementation issues. The current version of `isl` therefore does not provide any interface for specifying big parameters. Instead, the user can specify whether a maximum needs to be computed and no assumptions are made on the sign of the unknowns. Instead, the sign of the unknowns is checked internally and a big parameter is automatically introduced when needed. For compatibility with `PipLib`, the `isl_pip` tool does explicitly add non-negativity constraints on the unknowns unless the `Urs_unknowns` option is specified. Currently, there is also no

way in isl of expressing a big parameter in the output. Even though isl makes the same divisibility assumption on the big parameter as recent versions of PipLib, it will therefore eventually produce an error if the problem turns out to be unbounded.

2.3.5 Preprocessing

In this section, we describe some transformations that are or can be applied in advance to reduce the running time of the actual dual simplex method with Gomory cuts.

Feasibility Check and Detection of Equalities

Experience with the original PipLib has shown that Gomory cuts do not perform very well on problems that are (non-obviously) empty, i.e., problems with rational solutions, but no integer solutions. In isl, we therefore first perform a feasibility check on the original problem considered as a non-parametric problem over the combined space of unknowns and parameters. In fact, we do not simply check the feasibility, but we also check for implicit equalities among the integer points by computing the integer affine hull. The algorithm used is the same as that described in Section 2.3.7 below. Computing the affine hull is fairly expensive, but it can bring huge benefits if any equalities can be found or if the problem turns out to be empty.

Constraint Simplification

If the coefficients of the unknown and parameters in a constraint have a common factor, then this factor should be removed, possibly rounding down the constant term. For example, the constraint $2x - 5 \geq 0$ should be simplified to $x - 3 \geq 0$. isl performs such simplifications on all sets and relations. Recent versions of PipLib also perform this simplification on the input.

Exploiting Equalities

If there are any (explicit) equalities in the input description, PipLib converts each into a pair of inequalities. It is also possible to write r equalities as $r + 1$ inequalities (Feautrier et al. 2002), but it is even better to *exploit* the equalities to reduce the dimensionality of the problem. Given an equality involving at least one unknown, we pivot the row corresponding to the equality with the column corresponding to the last unknown with non-zero coefficient. The new column variable can then be removed completely because it is identically zero, thereby reducing the dimensionality of the problem by one. The last unknown is chosen to ensure that the columns of the initial tableau remain lexicographically positive. In particular, if the equality is of the form $b + \sum_{i \leq j} a_i x_i = 0$ with $a_j \neq 0$, then the (implicit) top rows of the initial tableau are

changed as follows

$$\begin{array}{c} \\ j \end{array}\!\!\begin{array}{c} \begin{array}{cccc} & & \mathrm{j} & \end{array} \\ \begin{pmatrix} 0 & I_1 & & \\ 0 & & 1 & \\ 0 & & & I_2 \end{pmatrix} \end{array} \rightsquigarrow \begin{array}{c} \\ j \end{array}\!\!\begin{pmatrix} 0 & I_1 & \\ -b/a_j & -a_i/a_j & \\ 0 & & I_2 \end{pmatrix}$$

Currently, `isl` also eliminates equalities involving only parameters in a similar way, provided at least one of the coefficients is equal to one. The application of parameter compression (see below) would obviate the need for removing parametric equalities.

Offline Symmetry Detection

Some problems, notably those of Bygde (2010), have a collection of constraints, say $b_i(\mathbf{p}) + \langle \mathbf{a}, \mathbf{x} \rangle \geq 0$, that only differ in their (parametric) constant terms. These constant terms will be non-negative on different parts of the context and this context may have to be split for each of the constraints. In the worst case, the basic algorithm may have to consider all possible orderings of the constant terms. Instead, `isl` introduces a new parameter, say u, and replaces the collection of constraints by the single constraint $u + \langle \mathbf{a}, \mathbf{x} \rangle \geq 0$ along with context constraints $u \leq b_i(\mathbf{p})$. Any solution to the new system is also a solution to the original system since $\langle \mathbf{a}, \mathbf{x} \rangle \geq -u \geq -b_i(\mathbf{p})$. Conversely, $m = \min_i b_i(\mathbf{p})$ satisfies the constraints on u and therefore extends a solution to the new system. It can also be plugged into a new solution. See Section 2.3.6 for how this substitution is currently performed in `isl`. The method described in this section can only detect symmetries that are explicitly available in the input. See Section 2.3.9 for the detection and exploitation of symmetries that appear during the course of the dual simplex method.

Parameter Compression

It may in some cases be apparent from the equalities in the problem description that there can only be a solution for a sublattice of the parameters. In such cases "parameter compression" (Meister 2004; Meister and Verdoolaege 2008) can be used to replace the parameters by alternative "dense" parameters. For example, if there is a constraint $2x = n$, then the system will only have solutions for even values of n and n can be replaced by $2n'$. Similarly, the parameters n and m in a system with the constraint $2n = 3m$ can be replaced by a single parameter n' with $n = 3n'$ and $m = 2n'$. It is also possible to perform a similar compression on the unknowns, but it would be more complicated as the compression would have to preserve the lexicographical order. Moreover, due to our handling of equalities described above there should be no need for such variable compression. Although parameter compression has been implemented in `isl`, it is currently not yet used during parametric integer programming.

2.3.6 Postprocessing

The output of `PipLib` is a quast (quasi-affine selection tree). Each internal node in this tree corresponds to a split of the context based on a parametric constant term in the main

tableau with indeterminate sign. Each of these nodes may introduce extra variables in the context corresponding to integer divisions. Each leaf of the tree prescribes the solution in that part of the context that satisfies all the conditions on the path leading to the leaf. Such a quast is a very economical way of representing the solution, but it would not be suitable as the (only) internal representation of sets and relations in `isl`. Instead, `isl` represents the constraints of a set or relation in disjunctive normal form. The result of a parametric integer programming problem is then also converted to this internal representation. Unfortunately, the conversion to disjunctive normal form can lead to an explosion of the size of the representation. In some cases, this overhead would have to be paid anyway in subsequent operations, but in other cases, especially for outside users that just want to solve parametric integer programming problems, we would like to avoid this overhead in future. That is, we are planning on introducing quasts or a related representation as one of several possible internal representations and on allowing the output of `isl_pip` to optionally be printed as a quast.

Currently, `isl` also does not have an internal representation for expressions such as $\min_i b_i(\mathbf{p})$ from the offline symmetry detection of Section 2.3.5. Assume that one of these expressions has n bounds $b_i(\mathbf{p})$. If the expression does not appear in the affine expression describing the solution, but only in the constraints, and if moreover, the expression only appears with a positive coefficient, i.e., $\min_i b_i(\mathbf{p}) \geq f_j(\mathbf{p})$, then each of these constraints can simply be reduplicated n times, once for each of the bounds. Otherwise, a conversion to disjunctive normal form leads to n cases, each described as $u = b_i(\mathbf{p})$ with constraints $b_i(\mathbf{p}) \leq b_j(\mathbf{p})$ for $j > i$ and $b_i(\mathbf{p}) < b_j(\mathbf{p})$ for $j < i$. Note that even though this conversion leads to a size increase by a factor of n, not detecting the symmetry could lead to an increase by a factor of $n!$ if all possible orderings end up being considered.

2.3.7 Context Tableau

The main operation that a context tableau needs to provide is a test on the sign of an affine expression over the elements of the context. This sign can be determined by solving two integer linear feasibility problems, one with a constraint added to the context that enforces the expression to be non-negative and one where the expression is negative. As already mentioned by Feautrier (1988), any integer linear feasibility solver could be used, but the `PipLib` implementation uses a recursive call to the dual simplex with Gomory cuts algorithm to determine the feasibility of a context. In `isl`, two ways of handling the context have been implemented, one that performs the recursive call and one, used by default, that uses generalized basis reduction. We start with some optimizations that are shared between the two implementations and then discuss additional details of each of them.

Maintaining Witnesses

A common feature of both integer linear feasibility solvers is that they will not only say whether a set is empty or not, but if the set is non-empty, they will also provide a *witness* for this result, i.e., a point that belongs to the set. By maintaining a list of such witnesses, we can avoid many feasibility tests during the determination of the signs

of affine expressions. In particular, if the expression evaluates to a positive number on some of these points and to a negative number on some others, then no feasibility test needs to be performed. If all the evaluations are non-negative, we only need to check for the possibility of a negative value and similarly in case of all non-positive evaluations. Finally, in the rare case that all points evaluate to zero or at the start, when no points have been collected yet, one or two feasibility tests need to be performed depending on the result of the first test.

When a new constraint is added to the context, the points that violate the constraint are temporarily removed. They are reconsidered when we backtrack over the addition of the constraint, as they will satisfy the negation of the constraint. It is only when we backtrack over the addition of the points that they are finally removed completely. When an extra integer division is added to the context, the new coordinates of the witnesses can easily be computed by evaluating the integer division. The idea of keeping track of witnesses was first used in `barvinok`.

Choice of Constant Term on which to Split

Recall that if there are no rows with a non-positive constant term, but there are rows with an indeterminate sign, then the context needs to be split along the constant term of one of these rows. If there is more than one such row, then we need to choose which row to split on first. `PipLib` uses a heuristic based on the (absolute) sizes of the coefficients. In particular, it takes the largest coefficient of each row and then selects the row where this largest coefficient is smaller than those of the other rows.

In `isl`, we take that row for which non-negativity of its constant term implies non-negativity of as many of the constant terms of the other rows as possible. The intuition behind this heuristic is that on the positive side, we will have fewer negative and indeterminate signs, while on the negative side, we need to perform a pivot, which may affect any number of rows meaning that the effect on the signs is difficult to predict. This heuristic is of course much more expensive to evaluate than the heuristic used by `PipLib`. More extensive tests are needed to evaluate whether the heuristic is worthwhile.

Dual Simplex + Gomory Cuts

When a new constraint is added to the context, the first steps of the dual simplex method applied to this new context will be the same or at least very similar to those taken on the original context, i.e., before the constraint was added. In `isl`, we therefore apply the dual simplex method incrementally on the context and backtrack to a previous state when a constraint is removed again. An initial implementation that was never made public would also keep the Gomory cuts, but the current implementation backtracks to before the point where Gomory cuts are added before adding an extra constraint to the context. Keeping the Gomory cuts has the advantage that the sample value is always an integer point and that this point may also satisfy the new constraint. However, due to the technique of maintaining witnesses explained above, we would not perform a feasibility test in such cases and then the previously added cuts may be redundant, possibly resulting in an accumulation of a large number of cuts.

If the parameters may be negative, then the same big parameter trick used in the main tableau is applied to the context. This big parameter is of course unrelated to the big parameter from the main tableau. Note that it is not a requirement for this parameter to be "big", but it does allow for some code reuse in `isl`. In `PipLib`, the extra parameter is not "big", but this may be because the big parameter of the main tableau also appears in the context tableau.

Finally, it was reported by Galea (2009), who worked on a parametric integer programming implementation in `PPL` (Bagnara et al.), that it is beneficial to add cuts for *all* rational coordinates in the context tableau. Based on this report, the initial `isl` implementation was adapted accordingly.

Generalized Basis Reduction

The default algorithm used in `isl` for feasibility checking is generalized basis reduction (Cook et al. 1991). This algorithm is also used in the `barvinok` implementation. The algorithm is fairly robust, but it has some overhead. We therefore try to avoid calling the algorithm in easy cases. In particular, we incrementally keep track of points for which the entire unit hypercube positioned at that point lies in the context. This set is described by translates of the constraints of the context and if (rationally) non-empty, any rational point in the set can be rounded up to yield an integer point in the context.

A restriction of the algorithm is that it only works on bounded sets. The affine hull of the recession cone therefore needs to be projected out first. As soon as the algorithm is invoked, we then also incrementally keep track of this recession cone. The reduced basis found by one call of the algorithm is also reused as initial basis for the next call.

Some problems lead to the introduction of many integer divisions. Within a given context, some of these integer divisions may be equal to each other, even if the expressions are not identical, or they may be equal to some affine combination of other variables. To detect such cases, we compute the affine hull of the context each time a new integer division is added. The algorithm used for computing this affine hull is that of Karr (1976), while the points used in this algorithm are obtained by performing integer feasibility checks on that part of the context outside the current approximation of the affine hull. The list of witnesses is used to construct an initial approximation of the hull, while any extra points found during the construction of the hull is added to this list. Any equality found in this way that expresses an integer division as an *integer* affine combination of other variables is propagated to the main tableau, where it is used to eliminate that integer division.

2.3.8 Experiments

Table 2.1 compares the execution times of `isl` (with both types of context tableau) on some more difficult instances to those of other tools, run on an Intel Xeon W3520 @ 2.66GHz. Easier problems such as the test cases distributed with `PipLib` can be solved so quickly that we would only be measuring overhead such as input/output and conversions and not the running time of the actual algorithm. We compare the following versions: `piplib-1.4.0-5-g0132fd9`, `barvinok-0.32.1-73-gc5d7751`, `isl-0.05.1-82-g3a37260` and `PPL` version 0.11.2.

	PipLib	barvinok	isl cut	isl gbr	PPL
Phideo	TC	793m	>999m	2.7s	372m
e1	0.33s	3.5s	0.08s	0.11s	0.18s
e3	0.14s	0.13s	0.10s	0.10s	0.17s
e4	0.24s	9.1s	0.09s	0.11s	0.70s
e5	0.12s	6.0s	0.06s	0.14s	0.17s
e6	0.10s	6.8s	0.17s	0.08s	0.21s
e7	0.03s	0.27s	0.04s	0.04s	0.03s
e8	0.03s	0.18s	0.03s	0.04s	0.01s
e9	OOM	70m	2.6s	0.94s	22s
vd	0.04s	0.10s	0.03s	0.03s	0.03s
bouleti	0.25s	line	0.06s	0.06s	0.15s
difficult	OOM	1.3s	1.7s	0.33s	1.4s
cnt/sum	TC	max	2.2s	2.2s	OOM
jcomplex	TC	max	3.7s	3.9s	OOM

Table 2.1: Comparison of Execution Times

The first test case is the following dependence analysis problem originating from the Phideo project (Verhaegh 1995) that was communicated to us by Bart Kienhuis:

```
lexmax { [j1,j2] -> [i1,i2,i3,i4,i5,i6,i7,i8,i9,i10] : 1 <= i1,j1
     <= 8 and 1 <= i2,i3,i4,i5,i6,i7,i8,i9,i10 <= 2 and 1 <= j2
    <= 128 and i1-1 = j1-1 and i2-1+2*i3-2+4*i4-4+8*i5-8+16*i6
    -16+32*i7-32+64*i8-64+128*i9-128+256*i10-256=3*j2-3+66 };
```

This problem was the main inspiration for some of the optimizations in Section 2.3.7. The second group of test cases are projections used during counting. The first nine of these come from Seghir and Loechner (2006). The remaining two come from Verdoolaege et al. (2005) and were used to drive the first, Gomory cuts based, implementation in `isl`. The third and final group of test cases are borrowed from Bygde (2010) and inspired the offline symmetry detection of Section 2.3.5. Without symmetry detection, the running times are 11s and 5.9s. All running times of `barvinok` and `isl` include a conversion to disjunctive normal form. Without this conversion, the final two cases can be solved in 0.07s and 0.21s. The `PipLib` implementation has some fixed limits and will sometimes report the problem to be too complex (TC), while on some other problems it will run out of memory (OOM). The `barvinok` implementation does not support problems with a non-trivial lineality space (line) nor maximization problems (max). The Gomory cuts based `isl` implementation was terminated after 1000 minutes on the first problem. The gbr version introduces some overhead on some of the easier problems, but is overall the clear winner.

2.3.9 Online Symmetry Detection

Manual experiments on small instances of the problems of Bygde (2010) and an analysis of the results by the approximate MPA method developed by Bygde (2010) have

revealed that these problems contain many more symmetries than can be detected using the offline method of Section 2.3.5. In this section, we present an online detection mechanism that has not been implemented yet, but that has shown promising results in manual applications.

Let us first consider what happens when we do not perform offline symmetry detection. At some point, one of the $b_i(\mathbf{p}) + \langle \mathbf{a}, \mathbf{x} \rangle \geq 0$ constraints, say the jth constraint, appears as a column variable, say c_1, while the other constraints are represented as rows of the form $b_i(\mathbf{p}) - b_j(\mathbf{p}) + c$. The context is then split according to the relative order of $b_j(\mathbf{p})$ and one of the remaining $b_i(\mathbf{p})$. The offline method avoids this split by replacing all $b_i(\mathbf{p})$ by a single newly introduced parameter that represents the minimum of these $b_i(\mathbf{p})$. In the online method the split is similarly avoided by the introduction of a new parameter. In particular, a new parameter is introduced that represents $\left|b_j(\mathbf{p}) - b_i(\mathbf{p})\right|_+ = \max(b_j(\mathbf{p}) - b_i(\mathbf{p}), 0)$.

In general, let $r = b(\mathbf{p}) + \langle \mathbf{a}, \mathbf{c} \rangle$ be a row of the tableau such that the sign of $b(\mathbf{p})$ is indeterminate and such that exactly one of the elements of $\mathbf{a}$ is a 1, while all remaining elements are non-positive. That is, $r = b(\mathbf{p}) + c_j - f$ with $f = -\sum_{i \neq j} a_i c_i \geq 0$. We introduce a new parameter t with context constraints $t \geq -b(\mathbf{p})$ and $t \geq 0$ and replace the column variable c_j by $c' + t$. The row r is now equal to $b(\mathbf{p}) + t + c' - f$. The constant term of this row is always non-negative because any negative value of $b(\mathbf{p})$ is compensated by $t \geq -b(\mathbf{p})$ while and non-negative value remains non-negative because $t \geq 0$.

We need to show that this transformation does not eliminate any valid solutions and that it does not introduce any spurious solutions. Given a valid solution for the original problem, we need to find a non-negative value of c' satisfying the constraints. If $b(\mathbf{p}) \geq 0$, we can take $t = 0$ so that $c' = c_j - t = c_j \geq 0$. If $b(\mathbf{p}) < 0$, we can take $t = -b(\mathbf{p})$. Since $r = b(\mathbf{p}) + c_j - f \geq 0$ and $f \geq 0$, we have $c' = c_j + b(\mathbf{p}) \geq 0$. Note that these choices amount to plugging in $t = |-b(\mathbf{p})|_+ = \max(-b(\mathbf{p}), 0)$. Conversely, given a solution to the new problem, we need to find a non-negative value of c_j, but this is easy since $c_j = c' + t$ and both of these are non-negative.

Plugging in $t = \max(-b(\mathbf{p}), 0)$ can be performed as in Section 2.3.6, but, as in the case of offline symmetry detection, it may be better to provide a direct representation for such expressions in the internal representation of sets and relations or at least in a quast-like output format.

2.4 Coalescing

See Verdoolaege (2009), for now. More details will be added later.

2.5 Transitive Closure

2.5.1 Introduction

Definition 2.5.1 (Power of a Relation) *Let $R \in \mathbb{Z}^n \to 2^{\mathbb{Z}^{d+d}}$ be a relation and $k \in \mathbb{Z}_{\geq 1}$ a positive number, then power k of relation R is defined as*

$$R^k := \begin{cases} R & \text{if } k = 1 \\ R \circ R^{k-1} & \text{if } k \geq 2. \end{cases} \tag{2.1}$$

Definition 2.5.2 (Transitive Closure of a Relation) *Let $R \in \mathbb{Z}^n \to 2^{\mathbb{Z}^{d+d}}$ be a relation, then the transitive closure R^+ of R is the union of all positive powers of R,*

$$R^+ := \bigcup_{k \geq 1} R^k.$$

Alternatively, the transitive closure may be defined inductively as

$$R^+ := R \cup (R \circ R^+). \tag{2.2}$$

Since the transitive closure of a polyhedral relation may no longer be a polyhedral relation (Kelly et al. 1996c), we can, in the general case, only compute an approximation of the transitive closure. Whereas Kelly et al. (1996c) compute underapproximations, we, like Beletska et al. (2009), compute overapproximations. That is, given a relation R, we will compute a relation T such that $R^+ \subseteq T$. Of course, we want this approximation to be as close as possible to the actual transitive closure R^+ and we want to detect the cases where the approximation is exact, i.e., where $T = R^+$.

For computing an approximation of the transitive closure of R, we follow the same general strategy as Beletska et al. (2009) and first compute an approximation of R^k for $k \geq 1$ and then project out the parameter k from the resulting relation.

Example 2.5.3 *As a trivial example, consider the relation $R = \{ x \to x + 1 \}$. The kth power of this map for arbitrary k is*

$$R^k = k \mapsto \{ x \to x + k \mid k \geq 1 \}.$$

The transitive closure is then

$$\begin{aligned} R^+ &= \{ x \to y \mid \exists k \in \mathbb{Z}_{\geq 1} : y = x + k \} \\ &= \{ x \to y \mid y \geq x + 1 \}. \end{aligned}$$

2.5.2 Computing an Approximation of R^k

There are some special cases where the computation of R^k is very easy. One such case is that where R does not compose with itself, i.e., $R \circ R = \emptyset$ or $\operatorname{dom} R \cap \operatorname{ran} R = \emptyset$. In this case, R^k is only non-empty for $k = 1$ where it is equal to R itself.

In general, it is impossible to construct a closed form of R^k as a polyhedral relation. We will therefore need to make some approximations. As a first approximations, we

will consider each of the basic relations in R as simply adding one or more offsets to a domain element to arrive at an image element and ignore the fact that some of these offsets may only be applied to some of the domain elements. That is, we will only consider the difference set ΔR of the relation. In particular, we will first construct a collection P of paths that move through a total of k offsets and then intersect domain and range of this collection with those of R. That is,

$$K = P \cap (\operatorname{dom} R \to \operatorname{ran} R), \tag{2.3}$$

with

$$P = \mathbf{s} \mapsto \{\, \mathbf{x} \to \mathbf{y} \mid \exists k_i \in \mathbb{Z}_{\geq 0}, \boldsymbol{\delta}_i \in k_i\, \Delta_i(\mathbf{s}) : \mathbf{y} = \mathbf{x} + \sum_i \boldsymbol{\delta}_i \wedge \sum_i k_i = k > 0 \,\} \tag{2.4}$$

and with Δ_i the basic sets that compose the difference set ΔR. Note that the number of basic sets Δ_i need not be the same as the number of basic relations in R. Also note that since addition is commutative, it does not matter in which order we add the offsets and so we are allowed to group them as we did in (2.4).

If all the Δ_is are singleton sets $\Delta_i = \{\, \boldsymbol{\delta}_i \,\}$ with $\boldsymbol{\delta}_i \in \mathbb{Z}^d$, then (2.4) simplifies to

$$P = \{\, \mathbf{x} \to \mathbf{y} \mid \exists k_i \in \mathbb{Z}_{\geq 0} : \mathbf{y} = \mathbf{x} + \sum_i k_i\, \boldsymbol{\delta}_i \wedge \sum_i k_i = k > 0 \,\} \tag{2.5}$$

and then the approximation computed in (2.3) is essentially the same as that of Beletska et al. (2009). If some of the Δ_is are not singleton sets or if some of $\boldsymbol{\delta}_i$s are parametric, then we need to resort to further approximations.

To ease both the exposition and the implementation, we will for the remainder of this section work with extended offsets $\Delta'_i = \Delta_i \times \{\, 1 \,\}$. That is, each offset is extended with an extra coordinate that is set equal to one. The paths constructed by summing such extended offsets have the length encoded as the difference of their final coordinates. The path P' can then be decomposed into paths P'_i, one for each Δ_i,

$$P' = ((P'_m \cup \mathrm{Id}) \circ \cdots \circ (P'_2 \cup \mathrm{Id}) \circ (P'_1 \cup \mathrm{Id})) \cap \{\, \mathbf{x}' \to \mathbf{y}' \mid y_{d+1} - x_{d+1} = k > 0 \,\}, \tag{2.6}$$

with

$$P'_i = \mathbf{s} \mapsto \{\, \mathbf{x}' \to \mathbf{y}' \mid \exists k \in \mathbb{Z}_{\geq 1}, \boldsymbol{\delta} \in k\, \Delta'_i(\mathbf{s}) : \mathbf{y}' = \mathbf{x}' + \boldsymbol{\delta} \,\}.$$

Note that each P'_i contains paths of length at least one. We therefore need to take the union with the identity relation when composing the P'_is to allow for paths that do not contain any offsets from one or more Δ'_i. The path that consists of only identity relations is removed by imposing the constraint $y_{d+1} - x_{d+1} > 0$. Taking the union with the identity relation means that that the relations we compose in (2.6) each consist of two basic relations. If there are m disjuncts in the input relation, then a direct application of the composition operation may therefore result in a relation with 2^m disjuncts, which is prohibitively expensive. It is therefore crucial to apply coalescing (Section 2.4) after each composition.

Let us now consider how to compute an overapproximation of P'_i. Those that correspond to singleton Δ_is are grouped together and handled as in (2.5). Note that this is

just an optimization. The procedure described below would produce results that are at least as accurate. For simplicity, we first assume that no constraint in Δ'_i involves any existentially quantified variables. We will return to existentially quantified variables at the end of this section. Without existentially quantified variables, we can classify the constraints of Δ'_i as follows

1. non-parametric constraints
$$A_1\mathbf{x} + \mathbf{c}_1 \geq \mathbf{0} \tag{2.7}$$
2. purely parametric constraints
$$B_2\mathbf{s} + \mathbf{c}_2 \geq \mathbf{0} \tag{2.8}$$
3. negative mixed constraints
$$A_3\mathbf{x} + B_3\mathbf{s} + \mathbf{c}_3 \geq \mathbf{0} \tag{2.9}$$
such that for each row j and for all $\mathbf{s}$,
$$\Delta'_i(\mathbf{s}) \cap \{\, \boldsymbol{\delta}' \mid B_{3,j}\mathbf{s} + c_{3,j} > 0 \,\} = \emptyset$$
4. positive mixed constraints
$$A_4\mathbf{x} + B_4\mathbf{s} + \mathbf{c}_4 \geq \mathbf{0}$$
such that for each row j, there is at least one $\mathbf{s}$ such that
$$\Delta'_i(\mathbf{s}) \cap \{\, \boldsymbol{\delta}' \mid B_{4,j}\mathbf{s} + c_{4,j} > 0 \,\} \neq \emptyset$$

We will use the following approximation Q_i for P'_i:

$$\begin{aligned} Q_i = \mathbf{s} \mapsto \{\, \mathbf{x}' \to \mathbf{y}' \mid \exists k \in \mathbb{Z}_{\geq 1}, \mathbf{f} \in \mathbb{Z}^d : \mathbf{y}' = \mathbf{x}' + (\mathbf{f}, k) \wedge \\ A_1\mathbf{f} + k\mathbf{c}_1 \geq \mathbf{0} \wedge B_2\mathbf{s} + \mathbf{c}_2 \geq \mathbf{0} \wedge A_3\mathbf{f} + B_3\mathbf{s} + \mathbf{c}_3 \geq \mathbf{0} \,\}. \end{aligned} \tag{2.10}$$

To prove that Q_i is indeed an overapproximation of P'_i, we need to show that for every $\mathbf{s} \in \mathbb{Z}^n$, for every $k \in \mathbb{Z}_{\geq 1}$ and for every $\mathbf{f} \in k\,\Delta_i(\mathbf{s})$ we have that $(\mathbf{f}, k)$ satisfies the constraints in (2.10). If $\Delta_i(\mathbf{s})$ is non-empty, then $\mathbf{s}$ must satisfy the constraints in (2.8). Each element $(\mathbf{f}, k) \in k\,\Delta'_i(\mathbf{s})$ is a sum of k elements $(\mathbf{f}_j, 1)$ in $\Delta'_i(\mathbf{s})$. Each of these elements satisfies the constraints in (2.7), i.e.,

$$\begin{bmatrix} A_1 & \mathbf{c}_1 \end{bmatrix} \begin{bmatrix} \mathbf{f}_j \\ 1 \end{bmatrix} \geq \mathbf{0}.$$

The sum of these elements therefore satisfies the same set of inequalities, i.e., $A_1\mathbf{f} + k\mathbf{c}_1 \geq \mathbf{0}$. Finally, the constraints in (2.9) are such that for any $\mathbf{s}$ in the parameter domain of Δ, we have $-\mathbf{r}(\mathbf{s}) \coloneqq B_3\mathbf{s} + \mathbf{c}_3 \leq \mathbf{0}$, i.e., $A_3\mathbf{f}_j \geq \mathbf{r}(\mathbf{s}) \geq \mathbf{0}$ and therefore also $A_3\mathbf{f} \geq \mathbf{r}(\mathbf{s})$. Note that if there are no mixed constraints and if the rational relaxation of $\Delta_i(\mathbf{s})$, i.e., $\{\, \mathbf{x} \in \mathbb{Q}^d \mid A_1\mathbf{x} + \mathbf{c}_1 \geq \mathbf{0} \,\}$, has integer vertices, then the approximation

is exact, i.e., $Q_i = P'_i$. In this case, the vertices of $\Delta'_i(\mathbf{s})$ generate the rational cone $\{\, \mathbf{x}' \in \mathbb{Q}^{d+1} \mid \begin{bmatrix} A_1 & \mathbf{c}_1 \end{bmatrix} \mathbf{x}' \,\}$ and therefore $\Delta'_i(\mathbf{s})$ is a Hilbert basis of this cone (Schrijver 1986, Theorem 16.4).

Existentially quantified variables can be handled by classifying them into variables that are uniquely determined by the parameters, variables that are independent of the parameters and others. The first set can be treated as parameters and the second as variables. Constraints involving the other existentially quantified variables are removed.

Example 2.5.4 *Consider the relation*

$$R = n \to \{\, x \to y \mid \exists\, \alpha_0, \alpha_1 : 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -1 - x + y \wedge y \geq 6 + x \,\}.$$

The difference set of this relation is

$$\Delta = \Delta R = n \to \{\, x \mid \exists\, \alpha_0, \alpha_1 : 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -1 + x \wedge x \geq 6 \,\}.$$

The existentially quantified variables can be defined in terms of the parameters and variables as

$$\alpha_0 = \left\lfloor \frac{-2+n}{7} \right\rfloor \qquad \textit{and} \qquad \alpha_1 = \left\lfloor \frac{-1+x}{5} \right\rfloor.$$

α_0 can therefore be treated as a parameter, while α_1 can be treated as a variable. This in turn means that $7\alpha_0 = -2+n$ can be treated as a purely parametric constraint, while the other two constraints are non-parametric. The corresponding Q (2.10) *is therefore*

$$\begin{aligned} n \to \{\, (x,z) \to (y,w) \mid \exists\, \alpha_0, \alpha_1, k, f : k \geq 1 \wedge y = x + f \wedge w = z + k \wedge \\ 7\alpha_0 = -2 + n \wedge 5\alpha_1 = -k + x \wedge x \geq 6k \,\}. \end{aligned}$$

Projecting out the final coordinates encoding the length of the paths, results in the exact transitive closure

$$R^+ = n \to \{\, x \to y \mid \exists\, \alpha_0, \alpha_1 : 7\alpha_1 = -2 + n \wedge 6\alpha_0 \geq -x + y \wedge 5\alpha_0 \leq -1 - x + y \,\}.$$

The fact that we ignore some impure constraints clearly leads to a loss of accuracy. In some cases, some of this loss can be recovered by not considering the parameters in a special way. That is, instead of considering the set

$$\Delta = \Delta R = \mathbf{s} \mapsto \{\, \boldsymbol{\delta} \in \mathbb{Z}^d \mid \exists \mathbf{x} \to \mathbf{y} \in R : \boldsymbol{\delta} = \mathbf{y} - \mathbf{x} \,\}$$

we consider the set

$$\Delta' = \Delta R' = \{\, \boldsymbol{\delta} \in \mathbb{Z}^{n+d} \mid \exists (\mathbf{s}, \mathbf{x}) \to (\mathbf{s}, \mathbf{y}) \in R' : \boldsymbol{\delta} = (\mathbf{s} - \mathbf{s}, \mathbf{y} - \mathbf{x}) \,\}.$$

The first n coordinates of every element in Δ' are zero. Projecting out these zero coordinates from Δ' is equivalent to projecting out the parameters in Δ. The result is obviously a superset of Δ, but all its constraints are of type (2.7) and they can therefore all be used in the construction of Q_i.

Example 2.5.5 *Consider the relation*

$$R = n \to \{\, (x, y) \to (1 + x, 1 - n + y) \mid n \geq 2 \,\}.$$

We have

$$\Delta R = n \to \{\, (1, 1 - n) \mid n \geq 2 \,\}$$

and so, by treating the parameters in a special way, we obtain the following approximation for R^+*:*

$$n \to \{\, (x, y) \to (x', y') \mid n \geq 2 \wedge y' \leq 1 - n + y \wedge x' \geq 1 + x \,\}.$$

If we consider instead

$$R' = \{\, (n, x, y) \to (n, 1 + x, 1 - n + y) \mid n \geq 2 \,\}$$

then

$$\Delta R' = \{\, (0, 1, y) \mid y \leq -1 \,\}$$

and we obtain the approximation

$$n \to \{\, (x, y) \to (x', y') \mid n \geq 2 \wedge x' \geq 1 + x \wedge y' \leq x + y - x' \,\}.$$

If we consider both ΔR *and* $\Delta R'$*, then we obtain*

$$n \to \{\, (x, y) \to (x', y') \mid n \geq 2 \wedge y' \leq 1 - n + y \wedge x' \geq 1 + x \wedge y' \leq x + y - x' \,\}.$$

Note, however, that this is not the most accurate affine approximation that can be obtained. That would be

$$n \to \{\, (x, y) \to (x', y') \mid y' \leq 2 - n + x + y - x' \wedge n \geq 2 \wedge x' \geq 1 + x \,\}.$$

2.5.3 Checking Exactness

The approximation T for the transitive closure R^+ can be obtained by projecting out the parameter k from the approximation K (2.3) of the power R^k. Since K is an overapproximation of R^k, T will also be an overapproximation of R^+. To check whether the results are exact, we need to consider two cases depending on whether R is *cyclic*, where R is defined to be cyclic if R^+ maps any element to itself, i.e., $R^+ \cap \mathrm{Id} \neq \emptyset$. If R is acyclic, then the inductive definition of (2.2) is equivalent to its completion, i.e.,

$$R^+ = R \cup (R \circ R^+)$$

is a defining property. Since T is known to be an overapproximation, we only need to check whether

$$T \subseteq R \cup (R \circ T)\,.$$

This is essentially Theorem 5 of Kelly et al. (1996c). The only difference is that they only consider lexicographically forward relations, a special case of acyclic relations.

If, on the other hand, R is cyclic, then we have to resort to checking whether the approximation K of the power is exact. Note that T may be exact even if K is not exact, so the check is sound, but incomplete. To check exactness of the power, we simply need to check (2.1). Since again K is known to be an overapproximation, we only need to check whether

$$K'|_{y_{d+1}-x_{d+1}=1} \subseteq R'$$
$$K'|_{y_{d+1}-x_{d+1}\geq 2} \subseteq R' \circ K'|_{y_{d+1}-x_{d+1}\geq 1},$$

where $R' = \{ \mathbf{x}' \to \mathbf{y}' \mid \mathbf{x} \to \mathbf{y} \in R \wedge y_{d+1} - x_{d+1} = 1 \}$, i.e., R extended with path lengths equal to 1.

All that remains is to explain how to check the cyclicity of R. Note that the exactness on the power is always sound, even in the acyclic case, so we only need to be careful that we find all cyclic cases. Now, if R is cyclic, i.e., $R^+ \cap \mathrm{Id} \neq \emptyset$, then, since T is an overapproximation of R^+, also $T \cap \mathrm{Id} \neq \emptyset$. This in turn means that $\Delta K'$ contains a point whose first d coordinates are zero and whose final coordinate is positive. In the implementation we currently perform this test on P' instead of K'. Note that if R^+ is acyclic and T is not, then the approximation is clearly not exact and the approximation of the power K will not be exact either.

2.5.4 Decomposing R into strongly connected components

If the input relation R is a union of several basic relations that can be partially ordered then the accuracy of the approximation may be improved by computing an approximation of each strongly connected components separately. For example, if $R = R_1 \cup R_2$ and $R_1 \circ R_2 = \emptyset$, then we know that any path that passes through R_2 cannot later pass through R_1, i.e.,

$$R^+ = R_1^+ \cup R_2^+ \cup (R_2^+ \circ R_1^+). \tag{2.11}$$

We can therefore compute (approximations of) transitive closures of R_1 and R_2 separately. Note, however, that the condition $R_1 \circ R_2 = \emptyset$ is actually too strong. If $R_1 \circ R_2$ is a subset of $R_2 \circ R_1$ then we can reorder the segments in any path that moves through both R_1 and R_2 to first move through R_1 and then through R_2.

This idea can be generalized to relations that are unions of more than two basic relations by constructing the strongly connected components in the graph with as vertices the basic relations and an edge between two basic relations R_i and R_j if R_i needs to follow R_j in some paths. That is, there is an edge from R_i to R_j iff

$$R_i \circ R_j \not\subseteq R_j \circ R_i. \tag{2.12}$$

The components can be obtained from the graph by applying Tarjan's algorithm (Tarjan 1972).

In practice, we compute the (extended) powers K_i' of each component separately and then compose them as in (2.6). Note, however, that in this case the order in which we apply them is important and should correspond to a topological ordering of the strongly connected components. Simply applying Tarjan's algorithm will produce topologically sorted strongly connected components. The graph on which Tarjan's algorithm is applied is constructed on-the-fly. That is, whenever the algorithm checks

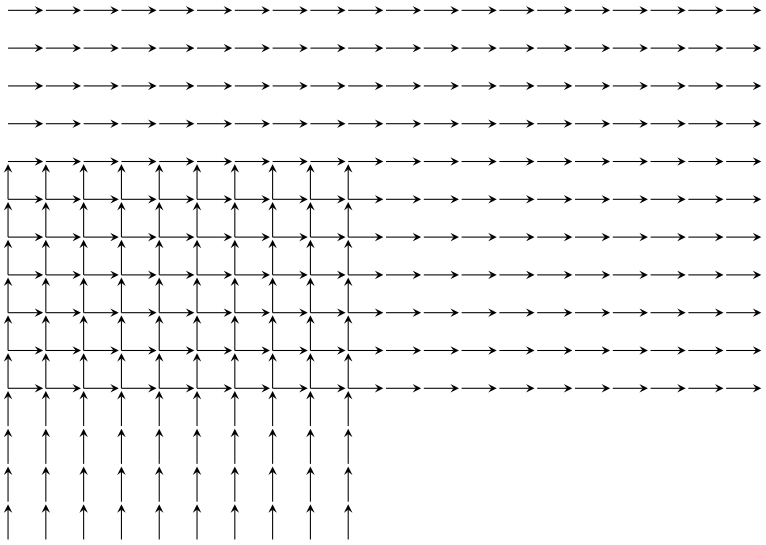

Figure 2.1: The relation from Example 2.5.6

if there is an edge between two vertices, we evaluate (2.12). The exactness check is performed on each component separately. If the approximation turns out to be inexact for any of the components, then the entire result is marked inexact and the exactness check is skipped on the components that still need to be handled.

It should be noted that (2.11) is only valid for exact transitive closures. If overapproximations are computed in the right hand side, then the result will still be an overapproximation of the left hand side, but this result may not be transitively closed. If we only separate components based on the condition $R_i \circ R_j = \emptyset$, then there is no problem, as this condition will still hold on the computed approximations of the transitive closures. If, however, we have exploited (2.12) during the decomposition and if the result turns out not to be exact, then we check whether the result is transitively closed. If not, we recompute the transitive closure, skipping the decomposition. Note that testing for transitive closedness on the result may be fairly expensive, so we may want to make this check configurable.

Example 2.5.6 *Consider the relation in example* `closure4` *that comes with the Omega calculator (Kelly et al. 1996a),* $R = R_1 \cup R_2$*, with*

$$R_1 = \{ (x, y) \to (x, y + 1) \mid 1 \leq x, y \leq 10 \}$$
$$R_2 = \{ (x, y) \to (x + 1, y) \mid 1 \leq x \leq 20 \wedge 5 \leq y \leq 15 \}.$$

This relation is shown graphically in Figure 2.1. We have

$$R_1 \circ R_2 = \{ (x, y) \to (x + 1, y + 1) \mid 1 \leq x \leq 9 \wedge 5 \leq y \leq 10 \}$$
$$R_2 \circ R_1 = \{ (x, y) \to (x + 1, y + 1) \mid 1 \leq x \leq 10 \wedge 4 \leq y \leq 10 \}.$$

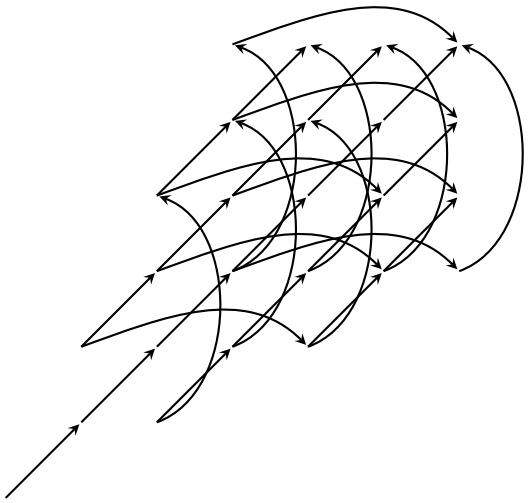

Figure 2.2: The relation from Example 2.5.7

Clearly, $R_1 \circ R_2 \subseteq R_2 \circ R_1$ *and so*

$$(R_1 \cup R_2)^+ = \left(R_2^+ \circ R_1^+\right) \cup R_1^+ \cup R_2^+.$$

Example 2.5.7 *Consider the relation on the right of Beletska et al. (2009, Figure 2), reproduced in Figure 2.2. The relation can be described as* $R = R_1 \cup R_2 \cup R_3$*, with*

$$R_1 = n \mapsto \{\,(i,j) \to (i+3,j) \mid i \leq 2j-4 \wedge i \leq n-3 \wedge j \leq 2i-1 \wedge j \leq n\,\}$$
$$R_2 = n \mapsto \{\,(i,j) \to (i,j+3) \mid i \leq 2j-1 \wedge i \leq n \wedge j \leq 2i-4 \wedge j \leq n-3\,\}$$
$$R_3 = n \mapsto \{\,(i,j) \to (i+1,j+1) \mid i \leq 2j-1 \wedge i \leq n-1 \wedge j \leq 2i-1 \wedge j \leq n-1\,\}.$$

The figure shows this relation for $n = 7$*. Both* $R_3 \circ R_1 \subseteq R_1 \circ R_3$ *and* $R_3 \circ R_2 \subseteq R_2 \circ R_3$*, which the reader can verify using the* `iscc` *calculator:*

```
R1 := [n] -> { [i,j] -> [i+3,j] : i <= 2 j - 4 and i <= n - 3 and
                                  j <= 2 i - 1 and j <= n };
R2 := [n] -> { [i,j] -> [i,j+3] : i <= 2 j - 1 and i <= n and
                                  j <= 2 i - 4 and j <= n - 3 };
R3 := [n] -> { [i,j] -> [i+1,j+1] : i <= 2 j - 1 and i <= n - 1 and
                                    j <= 2 i - 1 and j <= n - 1 };
(R1 . R3) - (R3 . R1);
(R2 . R3) - (R3 . R2);
```

R_3 *can therefore be moved forward in any path. For the other two basic relations, we have both* $R_2 \circ R_1 \not\subseteq R_1 \circ R_2$ *and* $R_1 \circ R_2 \not\subseteq R_2 \circ R_1$ *and so* R_1 *and* R_2 *form a strongly connected component. By computing the power of* R_3 *and* $R_1 \cup R_2$ *separately*

and composing the results, the power of R can be computed exactly using (2.5)*. As explained by Beletska et al. (2009), applying the same formula to R directly, without a decomposition, would result in an overapproximation of the power.*

2.5.5 Partitioning the domains and ranges of R

The algorithm of Section 2.5.2 assumes that the input relation R can be treated as a union of translations. This is a reasonable assumption if R maps elements of a given abstract domain to the same domain. However, if R is a union of relations that map between different domains, then this assumption no longer holds. In particular, when an entire dependence graph is encoded in a single relation, as is done by, e.g., Barthou et al. (2000, Section 6.1), then it does not make sense to look at differences between iterations of different domains. Now, arguably, a modified Floyd-Warshall algorithm should be applied to the dependence graph, as advocated by Kelly et al. (1996c), with the transitive closure operation only being applied to relations from a given domain to itself. However, it is also possible to detect disjoint domains and ranges and to apply Floyd-Warshall internally.

Algorithm 1: The modified Floyd-Warshall algorithm of Kelly et al. (1996c)

Input: Relations R_{pq}, $0 \leq p, q < n$
Output: Updated relations R_{pq} such that each relation R_{pq} contains all indirect paths from p to q in the input graph

```
1 for r ∈ [0, n − 1] do
2     R_rr := R_rr^+
3     for p ∈ [0, n − 1] do
4         for q ∈ [0, n − 1] do
5             if p ≠ r or q ≠ r then
6                 R_pq := R_pq ∪ (R_rq ∘ R_pr) ∪ (R_rq ∘ R_rr ∘ R_pr)
```

Let the input relation R be a union of m basic relations R_i. Let D_{2i} be the domains of R_i and D_{2i+1} the ranges of R_i. The first step is to group overlapping D_j until a partition is obtained. If the resulting partition consists of a single part, then we continue with the algorithm of Section 2.5.2. Otherwise, we apply Floyd-Warshall on the graph with as vertices the parts of the partition and as edges the R_i attached to the appropriate pairs of vertices. In particular, let there be n parts P_k in the partition. We construct n^2 relations

$$R_{pq} := \bigcup_{i \text{ s.t. } \operatorname{dom} R_i \subseteq P_p \wedge \operatorname{ran} R_i \subseteq P_q} R_i,$$

apply Algorithm 1 and return the union of all resulting R_{pq} as the transitive closure of R. Each iteration of the r-loop in Algorithm 1 updates all relations R_{pq} to include paths that go from p to r, possibly stay there for a while, and then go from r to q. Note that paths that "stay in r" include all paths that pass through earlier vertices since R_{rr} itself has been updated accordingly in previous iterations of the outer loop. In principle, it would

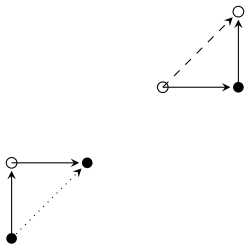

Figure 2.3: The relation (solid arrows) on the right of Figure 1 of Beletska et al. (2009) and its transitive closure

be sufficient to use the R_{pr} and R_{rq} computed in the previous iteration of the r-loop in Line 6. However, from an implementation perspective, it is easier to allow either or both of these to have been updated in the same iteration of the r-loop. This may result in duplicate paths, but these can usually be removed by coalescing (Section 2.4) the result of the union in Line 6, which should be done in any case. The transitive closure in Line 2 is performed using a recursive call. This recursive call includes the partitioning step, but the resulting partition will usually be a singleton. The result of the recursive call will either be exact or an overapproximation. The final result of Floyd-Warshall is therefore also exact or an overapproximation.

Example 2.5.8 *Consider the relation on the right of Figure 1 of Beletska et al. (2009), reproduced in Figure 2.3. This relation can be described as*

$$\{(x,y) \to (x_2,y_2) \mid (3y = 2x \wedge x_2 = x \wedge 3y_2 = 3 + 2x \wedge x \geq 0 \wedge x \leq 3) \vee \\ (x_2 = 1 + x \wedge y_2 = y \wedge x \geq 0 \wedge 3y \geq 2 + 2x \wedge x \leq 2 \wedge 3y \leq 3 + 2x)\}.$$

Note that the domain of the upward relation overlaps with the range of the rightward relation and vice versa, but that the domain of neither relation overlaps with its own range or the domain of the other relation. The domains and ranges can therefore be partitioned into two parts, P_0 and P_1, shown as the white and black dots in Figure 2.3, respectively. Initially, we have

$$R_{00} = \emptyset$$
$$R_{01} = \{(x,y) \to (x+1,y) \mid (x \geq 0 \wedge 3y \geq 2 + 2x \wedge x \leq 2 \wedge 3y \leq 3 + 2x)\}$$
$$R_{10} = \{(x,y) \to (x_2,y_2) \mid (3y = 2x \wedge x_2 = x \wedge 3y_2 = 3 + 2x \wedge x \geq 0 \wedge x \leq 3)\}$$
$$R_{11} = \emptyset.$$

In the first iteration, R_{00} remains the same ($\emptyset^+ = \emptyset$). R_{01} and R_{10} are therefore also unaffected, but R_{11} is updated to include $R_{01} \circ R_{10}$, i.e., the dashed arrow in the figure. This new R_{11} is obviously transitively closed, so it is not changed in the second iteration and it does not have an effect on R_{01} and R_{10}. However, R_{00} is updated to include $R_{10} \circ R_{01}$, i.e., the dotted arrow in the figure. The transitive closure of the original relation is then equal to $R_{00} \cup R_{01} \cup R_{10} \cup R_{11}$.

2.5.6 Incremental Computation

In some cases it is possible and useful to compute the transitive closure of union of basic relations incrementally. In particular, if R is a union of m basic maps,

$$R = \bigcup_j R_j,$$

then we can pick some R_i and compute the transitive closure of R as

$$R^+ = R_i^+ \cup \left(\bigcup_{j \neq i} R_i^* \circ R_j \circ R_i^* \right)^+ . \tag{2.13}$$

For this approach to be successful, it is crucial that each of the disjuncts in the argument of the second transitive closure in (2.13) be representable as a single basic relation, i.e., without a union. If this condition holds, then by using (2.13), the number of disjuncts in the argument of the transitive closure can be reduced by one. Now, $R_i^* = R_i^+ \cup \mathrm{Id}$, but in some cases it is possible to relax the constraints of R_i^+ to include part of the identity relation, say on domain D. We will use the notation $C(R_i, D) = R_i^+ \cup \mathrm{Id}_D$ to represent this relaxed version of R^+. Kelly et al. (1996c) use the notation $R_i^?$. $C(R_i, D)$ can be computed by allowing k to attain the value 0 in (2.10) and by using

$$P \cap (D \to D)$$

instead of (2.3). Typically, D will be a strict superset of both $\operatorname{dom} R_i$ and $\operatorname{ran} R_i$. We therefore need to check that domain and range of the transitive closure are part of $C(R_i, D)$, i.e., the part that results from the paths of positive length ($k \geq 1$), are equal to the domain and range of R_i. If not, then the incremental approach cannot be applied for the given choice of R_i and D.

In order to be able to replace R^* by $C(R_i, D)$ in (2.13), D should be chosen to include both $\operatorname{dom} R$ and $\operatorname{ran} R$, i.e., such that $\mathrm{Id}_D \circ R_j \circ \mathrm{Id}_D = R_j$ for all $j \neq i$. Kelly et al. (1996c) say that they use $D = \operatorname{dom} R_i \cup \operatorname{ran} R_i$, but presumably they mean that they use $D = \operatorname{dom} R \cup \operatorname{ran} R$. Now, this expression of D contains a union, so it not directly usable. Kelly et al. (1996c) do not explain how they avoid this union. Apparently, in their implementation, they are using the convex hull of $\operatorname{dom} R \cup \operatorname{ran} R$ or at least an approximation of this convex hull. We use the simple hull (Section 2.2) of $\operatorname{dom} R \cup \operatorname{ran} R$.

It is also possible to use a domain D that does *not* include $\operatorname{dom} R \cup \operatorname{ran} R$, but then we have to compose with $C(R_i, D)$ more selectively. In particular, if we have

$$\text{for each } j \neq i \text{ either } \operatorname{dom} R_j \subseteq D \text{ or } \operatorname{dom} R_j \cap \operatorname{ran} R_i = \emptyset \tag{2.14}$$

and, similarly,

$$\text{for each } j \neq i \text{ either } \operatorname{ran} R_j \subseteq D \text{ or } \operatorname{ran} R_j \cap \operatorname{dom} R_i = \emptyset \tag{2.15}$$

then we can refine (2.13) to

$$R_i^+ \cup \left(\left(\bigcup_{\substack{\operatorname{dom} R_j \subseteq D \\ \operatorname{ran} R_j \subseteq D}} C \circ R_j \circ C \right) \cup \left(\bigcup_{\substack{\operatorname{dom} R_j \cap \operatorname{ran} R_i = \emptyset \\ \operatorname{ran} R_j \subseteq D}} C \circ R_j \right) \cup \left(\bigcup_{\substack{\operatorname{dom} R_j \subseteq D \\ \operatorname{ran} R_j \cap \operatorname{dom} R_i = \emptyset}} R_j \circ C \right) \cup \left(\bigcup_{\substack{\operatorname{dom} R_j \cap \operatorname{ran} R_i = \emptyset \\ \operatorname{ran} R_j \cap \operatorname{dom} R_i = \emptyset}} R_j \right) \right)^+ .$$

If only property (2.14) holds, we can use

$$R_i^+ \cup \left((R_i^+ \cup \mathrm{Id}) \circ \left(\left(\bigcup_{\mathrm{dom}\, R_j \subseteq D} R_j \circ C \right) \cup \left(\bigcup_{\mathrm{dom}\, R_j \cap \mathrm{ran}\, R_i = \emptyset} R_j \right) \right)^+ \right),$$

while if only property (2.15) holds, we can use

$$R_i^+ \cup \left(\left(\left(\bigcup_{\mathrm{ran}\, R_j \subseteq D} C \circ R_j \right) \cup \left(\bigcup_{\mathrm{ran}\, R_j \cap \mathrm{dom}\, R_i = \emptyset} R_j \right) \right)^+ \circ (R_i^+ \cup \mathrm{Id}) \right).$$

It should be noted that if we want the result of the incremental approach to be transitively closed, then we can only apply it if all of the transitive closure operations involved are exact. If, say, the second transitive closure in (2.13) contains extra elements, then the result does not necessarily contain the composition of these extra elements with powers of R_i.

2.5.7 An `Omega`-like implementation

While the main algorithm of Kelly et al. (1996c) is designed to compute and underapproximation of the transitive closure, the authors mention that they could also compute overapproximations. In this section, we describe our implementation of an algorithm that is based on their ideas. Note that the `Omega` library computes underapproximations (Kelly et al. 1996b, Section 6.4).

The main tool is Equation (2) of Kelly et al. (1996c). The input relation R is first overapproximated by a "d-form" relation

$$\{ \mathbf{i} \to \mathbf{j} \mid \exists \boldsymbol{\alpha} : \mathbf{L} \leq \mathbf{j} - \mathbf{i} \leq \mathbf{U} \wedge (\forall p : j_p - i_p = M_p \alpha_p) \},$$

where p ranges over the dimensions and $\mathbf{L}$, $\mathbf{U}$ and $\mathbf{M}$ are constant integer vectors. The elements of $\mathbf{U}$ may be ∞, meaning that there is no upper bound corresponding to that element, and similarly for $\mathbf{L}$. Such an overapproximation can be obtained by computing strides, lower and upper bounds on the difference set ΔR. The transitive closure of such a "d-form" relation is

$$\{ \mathbf{i} \to \mathbf{j} \mid \exists \boldsymbol{\alpha}, k : k \geq 1 \wedge k\mathbf{L} \leq \mathbf{j} - \mathbf{i} \leq k\mathbf{U} \wedge (\forall p : j_p - i_p = M_p \alpha_p) \}. \tag{2.16}$$

The domain and range of this transitive closure are then intersected with those of the input relation. This is a special case of the algorithm in Section 2.5.2.

In their algorithm for computing lower bounds, the authors use the above algorithm as a substep on the disjuncts in the relation. At the end, they say

> If an upper bound is required, it can be calculated in a manner similar to that of a single conjunct [sic] relation.

Presumably, the authors mean that a "d-form" approximation of the whole input relation should be used. However, the accuracy can be improved by also trying to apply the incremental technique from the same paper, which is explained in more detail in

Section 2.5.6. In this case, $C(R_i, D)$ can be obtained by allowing the value zero for k in (2.16), i.e., by computing

$$\{\,\mathbf{i} \to \mathbf{j} \mid \exists \boldsymbol{\alpha}, k : k \geq 0 \wedge k\,\mathbf{L} \leq \mathbf{j} - \mathbf{i} \leq k\,\mathbf{U} \wedge (\forall p : j_p - i_p = M_p \alpha_p)\,\}.$$

In our implementation we take as D the simple hull (Section 2.2) of $\operatorname{dom} R \cup \operatorname{ran} R$. To determine whether it is safe to use $C(R_i, D)$, we check the following conditions, as proposed by Kelly et al. (1996c): $C(R_i, D) - R_i^+$ is not a union and for each $j \neq i$ the condition

$$\big(C(R_i, D) - R_i^+\big) \circ R_j \circ \big(C(R_i, D) - R_i^+\big) = R_j$$

holds.

Bibliography

Bagnara, R., P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library. `http://www.cs.unipr.it/ppl/`. [62]

Barthou, D., A. Cohen, and J.-F. Collard (2000). Maximal static expansion. *Int. J. Parallel Program. 28*(3), 213–243. [72]

Barvinok, A. and K. Woods (2003, April). Short rational generating functions for lattice point problems. *J. Amer. Math. Soc. 16*, 957–979. [55]

Beletska, A., D. Barthou, W. Bielecki, and A. Cohen (2009). Computing the transitive closure of a union of affine integer tuple relations. In *COCOA '09: Proceedings of the 3rd International Conference on Combinatorial Optimization and Applications*, Berlin, Heidelberg, pp. 98–109. Springer-Verlag. [65, 66, 71, 72, 73]

Boulet, P. and X. Redon (1998). Communication pre-evaluation in HPF. In *EUROPAR'98*, Volume 1470 of *Lecture Notes in Computer Science*, pp. 263–272. Springer-Verlag, Berlin. [55]

Bygde, S. (2010, March). Static WCET analysis based on abstract interpretation and counting of elements. Licentiate thesis. [59, 63]

Cook, W., T. Rutherford, H. E. Scarf, and D. F. Shallcross (1991, August). An implementation of the generalized basis reduction algorithm for integer programming. Cowles Foundation Discussion Papers 990, Cowles Foundation, Yale University. available at `http://ideas.repec.org/p/cwl/cwldpp/990.html`. [62]

De Loera, J. A., D. Haws, R. Hemmecke, P. Huggins, and R. Yoshida (2004, January). Three kinds of integer programming algorithms based on barvinok's rational functions. In *Integer Programming and Combinatorial Optimization: 10th International IPCO Conference*, Volume 3064 of *Lecture Notes in Computer Science*, pp. 244–255. [55]

Detlefs, D., G. Nelson, and J. B. Saxe (2005). Simplify: a theorem prover for program checking. *J. ACM 52*(3), 365–473. [55]

Feautrier, P. (1988). Parametric integer programming. *RAIRO Recherche Opérationnelle 22*(3), 243–268. [55, 56, 60]

Feautrier, P. (1991). Dataflow analysis of array and scalar references. *International Journal of Parallel Programming 20*(1), 23–53. [55, 57]

Feautrier, P. (1992, December). Some efficient solutions to the affine scheduling problem. Part II. multidimensional time. *International Journal of Parallel Programming 21*(6), 389–420. [57]

Feautrier, P., J. Collard, and C. Bastoul (2002). Solving systems of affine (in)equalities. Technical report, PRiSM, Versailles University. [57, 58]

Galea, F. (2009, November). personal communication. [62]

Karr, M. (1976). Affine relationships among variables of a program. *Acta Informatica 6*, 133–151. [62]

Kelly, W., V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott (1996a, November). The Omega calculator and library. Technical report, University of Maryland. [70]

Kelly, W., V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott (1996b, November). The Omega library. Technical report, University of Maryland. [76]

Kelly, W., W. Pugh, E. Rosser, and T. Shpeisman (1996c). Transitive closure of infinite graphs and its applications. In C.-H. Huang, P. Sadayappan, U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua (Eds.), *Languages and Compilers for Parallel Computing, 8th International Workshop, LCPC'95, Columbus, Ohio, USA, August 10-12, 1995, Proceedings*, Volume 1033 of *Lecture Notes in Computer Science*, pp. 126–140. Springer. [65, 69, 72, 73, 74, 75, 76]

Meister, B. (2004, December). *Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization*. Ph. D. thesis, Université Louis Pasteur. [59]

Meister, B. and S. Verdoolaege (2008, April). Polynomial approximations in the polytope model: Bringing the power of quasi-polynomials to the masses. In J. Sankaran and T. Vander Aa (Eds.), *Digest of the 6th Workshop on Optimization for DSP and Embedded Systems, ODES-6*. [59]

Nelson, C. G. (1980). *Techniques for program verification*. Ph. D. thesis, Stanford University, Stanford, CA, USA. [55]

Schrijver, A. (1986). *Theory of Linear and Integer Programming*. John Wiley & Sons. [67]

Seghir, R. and V. Loechner (2006, October). Memory optimization by counting points in integer transformations of parametric polytopes. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, CASES 2006, Seoul, Korea*. [63]

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing 1*(2), 146–160. [70]

Verdoolaege, S. (2006). barvinok, version 0.22. Available from http://freshmeat.net/projects/barvinok/. [55]

Verdoolaege, S. (2009, April). An integer set library for program analysis. Advances in the Theory of Integer Linear Optimization and its Extensions,AMS 2009 Spring Western Section Meeting, San Francisco, California, 25-26 April 2009. [64]

Verdoolaege, S., K. Beyls, M. Bruynooghe, and F. Catthoor (2005). Experiences with enumeration of integer projections of parametric polytopes. In R. Bodik (Ed.), *Proceedings of 14th International Conference on Compiler Construction, Edinburgh, Scotland*, Volume 3443 of *Lecture Notes in Computer Science*, Berlin, pp. 91–105. Springer-Verlag. [55, 63]

Verhaegh, W. F. J. (1995). *Multidimensional Periodic Scheduling*. Ph. D. thesis, Technische Universiteit Eindhoven. [62]