A Templated C++ Interface for isl

Sven Verdoolaege
Cerebras Systems
sven@cerebras.net

Oleksandr Zinenko
Google
zinenko@google.com

Manjunath Kudlur
Cerebras Systems
manjunath@cerebras.net

Ron Estrin
Cerebras Systems
ron.estrin@cerebras.net

Tianjiao Sun
Cerebras Systems
tianjiao.sun@cerebras.net

Harinath Kamepalli
Cerebras Systems
harinath@cerebras.net

Abstract

Polyhedral libraries typically support only a very limited collection of types for representing objects, corresponding to broad mathematical classes such as sets, binary relations and functions. Software built on top of these libraries, on the other hand, needs to deal with a plethora of different kinds of objects such as instance sets, access relations and dependence relations. Conceptually, these different kinds of objects can only be combined in very specific ways, but they are all mapped to the same limited collection of types, so that inconsistent combinations can only be detected at run time, if at all. This paper introduces a new templated C++ interface for isl that offers more fine-grained and application-specific types of objects, allowing inconsistencies in use to be detected at compile time, including some that cannot be detected at run time. This makes it easier to understand and correctly write polyhedral code.

1 Introduction

Any sufficiently advanced polyhedral compilation tool needs to deal with a multitude of different types of polyhedral objects such as instance sets, access relations, dependence relations, placements, schedules and memory mappings, as well as different variations and combinations of such objects. Strongly typed programming languages such as C++ (to some extent) allow the compiler to check whether certain operations are valid (at compile time, in the case of C++). However, most polyhedral libraries provide only a limited collection of types, restricting the scope for such compile-time consistency checks.

For example, in `PolyLib` (Wilde 1993) (practically) everything is a `Polyhedron`, to the extent that some authors even talk about "dependence polyhedra" instead of dependence relations. This affords only very limited type-checking, even at run time. Binary operations on `Polyhedrons` can at most compare the dimensionalities of the two objects. The extension to $\mathcal{Z}$-polyhedra (Nookala and Risset 2000) essentially only replaces `Polyhedron` by `ZPolyhedron`. Similarly, in `Omega` (Kelly et al. 1996) everything is a `Relation`, although a distinction is made between sets and binary relations, such that some additional inconsistencies can be detected at run time. In `isl` (Verdoolaege 2010), sets and relations can live in named and nested *spaces* (Verdoolaege 2011), enabling additional consistency checks, but again only at run time. Furthermore, some types of `isl` objects can contain elements from different spaces and then no such additional checks can be performed. Instead, an inappropriate operation usually simply results in an empty set or relation.

This paper introduces a templated C++ interface to `isl`, allowing for a more fine-grained and application-specific typing of polyhedral objects. For example, instead of simply using the "binary relation" type, a type can be used that represents a binary relation between statement instances and array (or tensor) elements, i.e., an access relation. The range of such a relation could then only be intersected with a set representing array elements. The concrete types are defined by the user. The templated interface only provides the infrastructure for defining these types. Since these types are defined at the C++ level, inconsistencies can be detected at *compile time*. Furthermore, the more elaborate types serve as documentation of the kind of objects accepted and returned by a function.

2 Background on `isl`

This section provides some background information on `isl` that will be needed to describe the templated C++ interface. Even though some of the details have not been described in any prior publication, they do not form part of the contributions of the present paper, but only serve to sketch the context in which the contributions were made. Verdoolaege (2016) provides further information on `isl`.

2.1 Spaces

`isl` is a library for manipulating sets of integer tuples. Besides a dimensionality, these tuples can also have a name. The collection of all integer tuples with a given name and dimensionality forms a *space*. For example, the integer tuples representing the elements of an array would typically live in a space with the same name and dimensionality as the array. Objects of the `isl` type `isl_set` contain elements that all live in the same space. Note that the constraints of a set may also reference symbolic constants, which have a

IMPACT 2021, January 20, 2021, Virtual event

fixed but unknown value. These symbolic constants are not considered to be part of a tuple or space. For example, the set

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

describes a two-dimensional array called A of size N in both dimensions, with N a symbolic constant. The name of the tuple is A and its dimensionality is 2.

Binary relations are represented using the isl_map type and contain elements that live in a space of a pair of named tuples. For example, the access relation from a particular statement S to a particular array A can be represented by a binary relation such as

```
[N] -> { S[i,j,k] -> A[i,j] : 0<=i,j,k<N }
```

where the first tuple has the name and dimensionality of the statement and the second those of the array.

Even though the dimensionality of a tuple can attain any value, the number of tuples in a space is limited to only three choices. As already explained, single-tuple objects are represented by an isl_set, while two-tuple objects are represented by an isl_map. It is also possible for an object to have no tuples. Such objects are also represented by an isl_set and describe constraints on symbolic constants. For example,

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

expresses that the symbolic constant N is non-negative.

There is currently no way to represent an object in isl with more than two tuples (at the top level). It is, however, possible for a tuple to have a pair of *nested tuples*. The dimensionality of the outer tuple is the sum of the dimensionalities of the nested tuples, which may in turn have further nested tuples. For example, it can be useful to make a distinction between different array references in a statement. To achieve this, a zero-dimensional tuple with a unique name can be used as a reference identifier. Each reference can then be represented by a *tagged access relation*, constructed by mapping a pair of a statement instance and a reference identifier, to an array element. This is done for example by pet (Verdoolaege and Grosser 2012). In isl, the outer tuple can also have a name, but this is rarely used since the tuple is already identified by the names of the leaf tuples. For simplicity, this paper will assume that all tuples with nested tuples are nameless. For example,

```
{ [S[i,j] -> R1[]] -> A[j,i] }
```

maps a pair consisting of an S-statement instance and an R1 reference identifier to an A-array element. The nested tuples S[i,j] and R1[] are named, but the containing tuple [S[i,j] -> R1[]] has no name.

It is often useful to consider objects containing elements from different spaces, e.g., the set of all statement instances or the binary relation containing all accesses. Such objects can be represented by an isl_union_set or an isl_union_map. For example,

```
[N] -> { S[i,j,k] : 0 <= i,j,k < N;
         T[i,j] : 0 <= i,j < N }
```

describes the instances of two statements. Note that while these objects contain elements living in different spaces, they are typically all of the same *kind*. For example, the set of all statement instances has elements in different spaces, each representing a particular statement, but they all represent a statement. This observation is crucial for the usefulness of the templated C++ interface as the purpose of the template arguments is to specify these kinds of tuples. Such a specification will be referred to as a "tuple kind" in the remainder of this paper.

2.2 Type Hierarchy

Besides sets and binary relations, isl also has several types representing explicit functions. The two basic types are the (total) quasi-affine function isl_aff and the (total) quasi-affine polynomial isl_qpolynomial. The domains of these functions are fixed and can have zero or one tuple, while the range is always a nameless single-dimensional tuple. Several type constructors can then be applied to these two types to obtain other types. In particular, the following type constructors are defined.

- multi takes a single-dimensional function type and turns it into a multi-dimensional function type, with a possibly named tuple of arbitrary dimensionality.
- pw takes a total function type on a domain in a fixed space and turns it into a partial function type, supporting a subdivision of the domain, each with a different expression.
- union takes a partial function type with a fixed space and turns it into a partial function type supporting multiple spaces.

For example, the total single-dimensional function

```
{ S[i,j] -> [j] }
```

can be represented by an isl_aff. The total multi-dimensional function

```
{ S[i,j] -> A[j,i] }
```

can be represented by an isl_multi_aff, but not by an isl_aff. The partial multi-dimensional function

```
{ S[i,j] -> A[j,i] : 0 <= i,j <= 10 }
```

can be represented by an isl_pw_multi_aff, but not by an isl_multi_aff. The multi-space partial multi-dimensional function

```
{ S[i,j] -> A[j,i] : 0 <= i,j <= 10;
  S[i,j] -> B[i] : 0 <= i,j <= 10 }
```

can be represented by an isl_union_pw_multi_aff, but not by an isl_pw_multi_aff.

The type constructors can be applied in different orders, but not all possible combinations are currently available

in isl or even make sense. Note that there is a difference between an isl_pw_multi_aff and an isl_multi_pw_aff. The former has a single subdivision of the domain, meaning that in each point of the domain, the function is either completely defined or undefined. The latter has a subdivision of the domain for each single-dimensional function inside the multi-dimensional function, meaning that on a given point in the domain some of these single-dimensional functions may be defined while some others may not. An isl_pw_multi_aff can therefore always be converted to an isl_multi_pw_aff without losing information (but not the other way around), since the subdivision of the domain can simply be pushed down to all single-dimensional functions in the target object.

Actually, there is a minor complication in the case of zero-dimensional functions, since a zero-dimensional function of type isl_pw_multi_aff could be defined on only a subset of the domain, while a zero-dimensional isl_multi_pw_aff has no single-dimensional function to which to attach this definition domain. For this reason, zero-dimensional objects of type isl_multi_pw_aff keep track of an explicit domain. The same applies to zero-dimensional objects of type isl_multi_union_pw_aff.

Every one of the above type constructors constructs a type that contains the input type. That is, every object of the input type is in theory also an object of the result type. In the C interface of isl, no such relationship can be expressed. Instead, the C interface provides conversion functions that convert an object of the more specific type to the more general type without loss of information. The only possible exception is the space of the object. In particular, an object of a union type does not have a specific (domain) space since the elements can live in multiple spaces. The space of an isl_union_pw_aff or an isl_union_pw_multi_aff therefore always has a single, one-dimensional tuple.

If, after applying some type constructor to a type, some other type constructor is applied to both the input and the result type, then the result of applying both type constructors also contains the result of only applying the second type constructor. Together with the special case of isl_multi_pw_aff generalizing isl_pw_multi_aff, this results in the type hierarchy shown in Figure 1. These subtype relationships are marked explicitly in the code using __isl_subclass annotations. These annotations are only used during the generation of the Python and the C++ bindings. Note that an isl_union_pw_multi_aff cannot necessarily be converted to an isl_multi_union_pw_aff because the former can have function values in different tuples (or no tuple), while the latter can only have function values in a single tuple. Also note that a given type may be a subclass of more than one other type and that the order of the superclasses (as shown in Figure 1) is important. In particular, as will be explained in Section 2.3, for generating the Python bindings it is important that the class hierarchy can be linearized. A

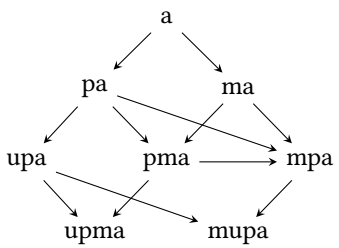

Figure 1. The hierarchy of classes obtained from isl_aff using zero or more applications of type constructors, with subclasses pointing to zero or more superclasses. If there is more than one superclass, then they are considered to be ordered from left to right. The primitive class is abbreviated by 'a'. The type constructors are abbreviated by their first letters.

similar hierarchy could be defined for isl_qpolynomial but not all corresponding types are currently defined in isl and this hierarchy is not currently exported to the bindings.

The isl_union_set and isl_union_map types described above can be seen to have been derived from isl_set and isl_map using the same union type constructor. A different type constructor, basic, is also applied to these two types to derive *specializations*. In particular, isl_basic_set and isl_basic_map are sets and binary relations that can be described using only projection and a conjunction of affine constraints. Finally, isl_point is a further specialization of isl_basic_set describing a single element.

The multi type constructor is also applied to the isl_id and isl_val types, representing identifiers and (rational) values, to obtain the isl_multi_id and isl_multi_val types. These types do not currently have an __isl_subclass annotation.

2.3 Python Interface

Historically, the Python interface provided the first foreign language bindings for isl.[1] It is only relevant here because the Python and the C++ interface are meant to be as similar to each other as possible and because the two interfaces are generated using the same infrastructure. In particular, an automated generator, written in C++, is used that parses the (annotated) C headers and produces the foreign language bindings.

Even though some effort was made to keep the C interface clean, in hindsight some functions should not have been added. Some other functions may also not be relevant for foreign language bindings. Therefore, all types and functions that should be exported to the foreign language bindings

[1]The Python interface that comes with isl should not be confused with the islpy library, which was introduced by Klöckner (2014) and is available from http://documen.tician.de/islpy/.

are explicitly annotated with `__isl_export` (if not already annotated with `__isl_subclass`). Functions are exported as methods of the Python class corresponding to the type that appears at the start of the function name. If the first argument of the function is not of the same type, then a static member function is generated. This is also called a "named constructor". In some cases, where the functionality of the function is clear from the remaining arguments, the function is exported as `__isl_constructor` and a proper constructor is generated instead.

In the Python interface, a type that is marked as being an `__isl_subclass` of some other type is effectively created as a subclass of that other type. For example, `isl.aff` is a subclass of both `isl.pw_aff` and `isl.multi_aff`. The order is important here because Python 3 needs to be able to linearize the classes in the hierarchy. The order of the subclasses is based on a predefined linearization. In particular, the type constructors are applied to previously defined types in a specific order: `pw`, `union` and `multi`. This results in the linearization

a, pa, upa, ma, pma, upma, mpa, mupa,

using the abbreviations in Figure 1.

Each Python object encapsulates a pointer to the corresponding C object. Each method call is implemented using a call to the corresponding C function. However, the Python method can be called with objects of subclasses as arguments (including `self`), while the corresponding C function needs to be called with pointers to C objects of the exact expected types. Each method therefore checks the `__class__` of the arguments and constructs new objects of the appropriate types if there is a mismatch. Each class then also has constructors for constructing an object from an object of a subclass, which call the corresponding C conversion functions.

2.4 C++ Interface

In theory, it would be possible to use run-time type information to apply the same sort of dynamic type checks that are being performed by the Python interface to support calls on objects of subclasses. However, when the C++ interface was contributed to `isl`, a choice was made to not expose the conceptual type hierarchy of Section 2.2 in the C++ class hierarchy, but instead to provide implicit conversion constructors from objects of conceptual subclasses. The constructors that take some other argument are marked `explicit` such that they do not get called implicitly. The availability of implicit conversion constructors means that, just like in the Python interface, a method can be called with objects of subclasses since these objects will automatically get converted to objects of the expected type. However, unlike the Python interface, this does not apply to the implicit `this` argument. That is, a method available in a superclass cannot be called on an object of a subclass. This means there is a difference with the Python interface and an asymmetry in the order of the arguments. For example, if `a` is an `isl_union_set` and `b` is an `isl_set`, then `a.intersect(b)` can be performed, but `b.intersect(a)` cannot. Instead `b` would have to be explicitly converted to an `isl_union_set` first. This inconsistency will be resolved in Section 4.2.

3 Design Goals

This section describes the main design goals of the templated C++ interface.

3.1 Compile-time Consistency Checks

The most important design goal is to be able to detect inconsistencies at compile time. For example, at the function definition level, a function accepting an access relation should not be able to get called with a dependence relation or any other type of relation. Similarly, at the `isl` interface level, it should not be possible to intersect an access relation with a dependence relation.

3.2 Application Independent

The interface should leave it up to the user application to define the different types of objects that the application uses. Otherwise, `isl` would need to be aware of all possible applications. That is, the interface should not define the concept of an access relation, but should instead make it possible for the application to do so.

3.3 Compatibility with Plain C++ Interface

A function taking an object from the plain C++ interface should also take an object from the templated C++ interface. The other direction should not be allowed since that would subvert the consistency checks afforded by the templated C++ interface.

Switching a function from the plain to the templated C++ interface should be as easy as possible. Ideally, it should be as easy as adding template arguments to the original types. Unfortunately, it is not possible in C++ to use the same name for both a regular type and a template type. This means that either different names should be used or that the plain and the templated C++ interface cannot be used together. The latter would imply that an entire code base would have to be switched over at once, which is impractical.

3.4 Ease of Use

The user should not be required to add more annotations than strictly necessary. Clearly, when a new object is being constructed from scratch (i.e., not as a result of applying an operation to some other object) or when a non-templated object is converted to a templated object, the tuple kinds need to be specified. Also, the argument and return types of a function should typically specify tuple kinds as well, even if just for documentation purposes, although it can be avoided by using template arguments and/or `auto`. Other

than these two cases, however, a user should not have to specify any tuple kinds.

4 Changes to Plain C++ Interface

This section describes some changes to the plain C++ interface that are also instrumental for the templated interface.

4.1 Subclasses Based on Type Functions

The original plain C++ interface exports some `isl` types that each represents a particular type of object. There are also some types in the C interface that represent multiple (related) types of objects and that have a `get_type` function to determine the specific subtype of an object, in particular `isl_ast_expr`, `isl_ast_node` and `isl_schedule_node`. These were not originally exported by the plain C++ (or Python) interface. The generator has been extended to export them as actual subclasses, in both the C++ and Python interfaces, based on the return value of the `get_type` function. In the Python interface, the `__new__` method of the superclass is overridden to create an object of the appropriate subclass. In the C++ interface, `isa` and `as` methods are introduced in the superclass to check whether an object actually belongs to the subclass and to convert it to that subclass if it does. Note that the types that are exported in this way do not involve any tuple information and therefore do not need to be handled by the templated interface. However, the name of the `as` method is relevant for the renaming described in Section 4.3 below.

4.2 Inherit Methods from Superclasses

As explained in Section 2.4, the conceptual `isl` type hierarchy is not expressed as such in the C++ interface. This means that methods available in a superclass may not be available on objects of a subclass. This happens in particular to methods of a superclass that take more general arguments than a method with the same name in the subclass. For example, the `intersect` method in `isl::set` only takes some other `isl::set`, while the same method in `isl::union_set` can take a more general `isl::union_set`. Note that this particular sort of methods would get hidden by the subclass method anyway, so making `isl::set` a subclass of `isl::union_set` would not help.

The solution is to copy methods from ancestors to subclasses. In particular, if some method is available in an ancestor that is not directly available in a subclass, because it has either a different name or different arguments, then it is also made available in the subclass. Instead of calling the corresponding C function, the copied methods convert `this` to an object of the superclass and then call the corresponding C++ method on the result. If a method with the same signature appears in more than one ancestor, then it is copied from the "closest" ancestor, where an ancestor is considered closer if the distance in the class hierarchy is smaller or the distance is the same and it appears more to the left in Figure 1. This selection mechanism may not always result in the same variant of the method getting called as in the Python bindings, but it is a reasonable choice and it would be difficult to mimic Python exactly.

Note that if there already was a method with the same name but different arguments, then adding a method copied from an ancestor may result in ambiguities that did not exist before the copying. In particular, an object that belongs to a subclass of the original argument type would get converted automatically to the required type, but if it can also be converted to the argument type of the copied method, then the compiler will no longer perform the conversion. In such cases, the generator will therefore add additional variants of the method to the subclass, one for each subclass of the original argument type.

4.3 Renamed Exports of Constructors

In the C interface, most functions for creating an object of a given `isl` type start with the name of that type. This means they appear as (named) constructors in the C++ interface. When calling such constructors, the C++ `isl` object type needs to be spelled out since there is no object from which to obtain the type. This is already a bit verbose in the plain C++ interface, but in the templated interface, it is even worse since there the type will also involve the tuple kinds. Many of these functions are therefore (re-)exported based on newly introduced names that start with the name of the first argument type instead of that of the return type. In most cases, the name of the result type still appears somewhere else in the new function name. For example, the `isl_set_universe` function, which gets exported as the named constructor `universe` in `isl::set`, is re-exported through the new `isl_space_universe_set` name, which gets exported as the method `universe_set` in `isl::space`.

Many conversion functions in the C interface have names of the form *new*`_from_`*old*. Most of these are or could be exported as constructors because they simply convert one representation of a mathematical object to some other representation. Some of these functions, however, are not capable of converting all possible input objects (in a faithful way) and then an export as an (unnamed) constructor would not be appropriate. For example, the function `isl_multi_pw_aff_from_pw_multi_aff` can convert any input object and is therefore exported as a constructor, but `isl_pw_multi_aff_from_multi_pw_aff` returns an object defined on the shared definition domain of the elements of the input object, which may not be the same as the input object. The latter function is therefore not exported as a constructor.

When (re-)exporting such functions using names that start with the first argument type, each of which results in a proper method, this difference needs to expressed in a different way. In particular, the functions that could be

exported as constructors get *old*_to_*new* as a new name, while the others get *old*_as_*new* as a new name. The as-name is inspired by the as-method of Section 4.1. There is, however, a specific set of functions where picking the right name is not that obvious. In particular, a function such as `isl_union_map_from_multi_union_pw_aff` can only convert input objects that live in a map space, so in the plain C++ interface, an as-name is more appropriate. However, in the templated interface, the method would only be available in objects with two tuples, so a to-name seems more appropriate. For consistency between the interfaces, the as-name is used in such cases.

5 Templated C++ Interface

This section describes the main contribution of this paper, the actual templated C++ interface.

5.1 Types

Each plain C++ type involving tuples gets a corresponding template type in the templated interface, with a number of template parameters that is equal to the number of tuples involved. In order to ease the transition from the plain interface to the templated interface, the name of the template type is the same as that of the plain C++ type. It therefore needs to be placed in a different namespace, namely `isl::typed`. This results in rather long fully qualified type names, but, if needed, an application can easily use a namespace alias.

Since some of the C++ types can involve different numbers of tuples, the templated types are defined with a variable number of template parameters and (partial) specializations are provided for each possible number of tuples. In fact, for ease of implementation, all templated types are defined with a variable number of template parameters, even if only one specific number of tuples is allowed. In particular:

- `space` can have 0, 1 or 2 tuples;
- set types `basic_set`, `set` and `point` can have 0 or 1 tuple;
- map types `basic_map` and `map` can have 2 tuples;
- the `fixed_box` type, which represents a rectangular approximation of a set or of the range of a map, can have 1 or 2 tuples;
- the `aff` type can have 1 or 2 tuples, but the last tuple is always a one-dimensional unnamed tuple;
- `val` and `id` are also considered as always having a single one-dimensional unnamed tuple, mainly because the `multi_val` and `multi_id` types are derived from them.

A special `Anonymous` tuple kind is introduced to represent the one-dimensional unnamed tuples and all specializations of `isl::typed::aff`, as well as the single specialization of both `isl::typed::val` and `isl::typed::id` have this tuple kind as the final (or only) template argument.

Besides the types listed above, those that are derived from them using some combination of the type constructors also get a corresponding templated type. The number of tuples of each derived type is the same as that of the type from which it is derived. The only modification is that the `multi` type constructor changes the `Anonymous` tuple to a generic tuple.

Each of the templated type specializations has a constructor that takes an object of the corresponding plain C++ type. This constructor is marked `private` and a static method called `from` is provided for calling the constructor indirectly. This ensures that the template arguments always need to be specified explicitly when constructing a templated object from a plain object. Otherwise, an object of a plain type could be passed to a function expecting a templated type and template type deduction would conjure up the right template arguments, bypassing the consistency checks. In any template specialization with a single `Anonymous` tuple kind, the constructor is not marked `private` to allow an automatic construction from the corresponding plain type. In these cases, the tuple kind is required to be `Anonymous` anyway and there is no point in requiring users to spell this out.

While, in general, a plain type object should not get converted automatically to an object of a template type, it should be possible to use an object of a templated type where one of a plain type is expected. The specializations are therefore made to be subclasses of the corresponding plain types. An alternative would be to provide an implicit conversion operator, but this does not have quite the same effect, especially for functions already relying on implicit conversion operators in the plain interface. Deriving from the plain types also allows some methods that do not need any further modifications to be reused directly.

For example, the specialization of `isl::typed::map` looks as follows.

```
template <typename Domain, typename Range>
struct map<Domain, Range> : public isl::map {
  map() = default;
 private:
  map(const isl::map &obj) : isl::map(obj) {}
 public:
  static map from(const isl::map &obj) {
    return map(obj);
  }
  /* ... */
};
```

The user can then define (or simply use) full specializations of this partial specialization for specific types of binary relations. For example, an access relation and a dependence relation type could be defined as follows.

```
struct Statement {};
struct Array {};
using access_relation =
  isl::typed::map<Statement, Array>;
```

```
using dependence_relation =
  isl::typed::map<Statement, Statement>;
```

Here, `Statement` and `Array` are dummy classes, whose only purpose is to be able to differentiate between full specializations of `isl::typed::map`. No objects of these dummy classes ever need to be created, at least not at run time. Any attempt to pass an `access_relation` object to a function expecting a `dependence_relation` will result in a compile-time error, as desired.

5.2 Methods

Since each template type specialization derives from the corresponding plain type, all methods are inherited from the plain type. These do not offer any consistency checks, but for some methods, in particular those derived from unary property functions in the C interface, resulting in methods with no arguments, this is sufficient. For other methods, specialization specific versions are added that enforce relationships between the argument types and the return type, if any.

The methods are generated based on a table describing the behavior of each method in terms of its effect on the tuples. Since the naming of `isl` functions is fairly systematic, the behavior of methods with the same name is usually the same and therefore only needs to be specified once. Methods that have the result type in their names also have this part removed when looking up their behavior. Each behavior is described as a sequence of signatures, where a signature consists of several sequences of 0, 1 or 2 abstract tuple kinds, one such sequence for the return type and one for each argument. The abstract tuple kind sequences for the arguments are grouped into a sequence of their own. The abstract tuple kinds themselves are described by placeholders, each of which will have a corresponding template parameter in the generated bindings. That is, if the same placeholder appears multiple times, then the corresponding (concrete) tuple kinds will be required to be the same. When generating a method for a particular specialization of a template type, the matching signature in the behavior will be used. For a regular (non-static) method, a signature matches if the abstract tuple kind sequence for the first argument matches that of the specialization, i.e., if the lengths are the same and if the placeholders in the first argument can be mapped to (consistent) parts in the type specialization. Different placeholders are mapped independently and they can map to the same part.

For example, for a simple binary operation such as the method `intersect`, the bindings generator has the following entry.

```
{ "intersect",
  { bin_params, bin_set, bin_map } },
```

The signatures that appear in this behavior are defined as follows, where the abstract tuple kinds of the return type appear before those of the list of arguments.

```
Signature bin_params = { { }, { { }, { } } };
Signature bin_set =
{ { Domain }, { { Domain }, { Domain } } };
Signature bin_map =
{ { Domain, Range },
  { { Domain, Range }, { Domain, Range } } };
```

That is, both the second argument type and the return type are equal to the first argument type. While generating the `isl::typed::map<Domain, Range>` specialization, only the `bin_map` signature matches and the following method declaration is generated.

```
inline typed::map<Domain, Range> intersect(
  const typed::map<Domain, Range> &m2) const;
```

The methods `intersect_domain` and `intersect_range` intersect the domain or the range of a binary relation. They have signatures

```
{ { Domain, Range },
  { { Domain, Range }, { Domain } } };
```

and

```
{ { Domain, Range },
  { { Domain, Range }, { Range } } };
```

respectively.

Some methods require extra template parameters for their arguments and/or return types. For example, `apply_range` applies some (other) binary relation to the range of a binary relation and has the following signature.

```
{ { Domain, Range2 },
  { { Domain, Range }, { Range, Range2 } } };
```

For `isl::typed::map<Domain, Range>`, the method below is generated.

```
template <typename Range2>
inline typed::map<Domain, Range2> apply_range(
  const typed::map<Range, Range2> &m2) const;
```

The method `set_range_tuple` changes the range tuple identifier of a binary relation or function. For objects with two tuples, it has the following signature.

```
{ { Domain, Range },
  { { Domain, Leaf }, { Anonymous } } };
```

Here, `Leaf` is a special placeholder that can only be matched with a template parameter. In particular, no method will be generated in some of the further specializations described below where the range has a nested pair of tuples. This corresponds to the current restriction of not having names on tuples containing nested tuples. For the specialization `isl::typed::map<Domain, Range>`, the signature above results in the generation of the following method.

```
template <typename Range>
inline typed::map<Domain, Range>
  set_range_tuple(
```

```
    const typed::id<Anonymous> &id) const;
```

Notice that the template parameter Range only appears in the return type and can therefore not be deduced automatically, but must instead be specified explicitly. This makes sense since a change in tuple identifier may result in a change in tuple kind and the new tuple kind needs to be specified explicitly. The id argument has type typed::id<Anonymous>, but this can be constructed automatically from an isl::id of the plain interface.

In some rare cases, the behavior of a method depends on the type on which it is invoked. For example, on a set or binary relation, gist has the same tuple behavior as intersect, but on any type derived from aff, the second argument refers to the domain of the function. It therefore has one of the following two signatures, one for 1-tuple functions, i.e., those that only depend on symbolic constants, and one for 2-tuple functions, i.e., those that have a proper domain.

```
{ { Range }, { { Range }, { } } }
{ { Domain, Range },
  { { Domain, Range }, { Domain } } }
```

The unary space method is another example. This method usually returns an object with the same tuple configuration as the object on which it is invoked, but for any type obtained using the union type constructor, the result has 0 tuples. A further exception is that if the multi type constructor was applied on top (i.e., the type is multi_union_pw_aff), then the result has 1 tuple, corresponding to the single or last tuple of the input.

5.3 Example Use

As a first trivial example, suppose the user application has defined a type ST representing statement instances and a type AR representing array elements. Given an access relation called access of type isl::typed::union_map<ST, AR> and a description of the statement instances called instances of type isl::typed::union_set<ST>, the statement

```
access.intersect_domain(instances);
```

will compile, while the statement

```
access.intersect_range(instances);
```

will not. Note that because these are objects (potentially) containing elements in multiple spaces, without the compile-time error, the problem would not even be detected at run time. Instead, the result would simply be empty.

The code fragment below is another small illustration of the use of the templated C++ interface. It defines a generic function for constructing a rectangular box based on lower and upper bounds in each dimension. The bounds can be either fixed values (multi_val<Domain>) or symbolic constants (multi_aff<Domain>), but they need to live in the same kind of space and this is enforced by the compiler. For example, trying to construct a box where the lower bound refers to statement instances and the upper bound refers to array elements will result in a compile-time error. Note that in this example, the bounds should also live in exactly the same space, but this is not enforced at compile time (see also Section 5.6). The body is written in a templated interface friendly way, calling the universe_set method on a space rather than calling the named constructor isl::typed::set<Domain>::universe, but the exact same code also works for the plain interface.

```
template <template <typename...> class T,
          template <typename...> class U,
          typename Domain>
auto construct_box(const T<Domain> &lower,
  const U<Domain> &upper)
{
  auto res = lower.space().universe_set();
  res = res.lower_bound(lower);
  res = res.upper_bound(upper);
  return res;
}
```

5.4 Further Specializations

If some behavior is described for a method, but no matching signature can be found, then the method is explicitly delete'd from the templated type to hide the method inherited from the corresponding plain type. This can happen in particular for methods involving nested tuples. Take, for example, the method domain_factor_domain. This method is applied to an object with nested tuples in its domain and projects out the second of these nested tuples, promoting the first of the nested tuples to the top level. For example, applying this method to a tagged access relation projects out the tags and results in a regular access relation. The signature is as follows.

```
{ { Domain, Range },
  { { { Domain, Domain2 }, Range } } };
```

While generating isl::typed::map<Domain, Range>, the abstract tuple kind sequence of the first (and single) argument cannot be matched against that of the specialization, because the Domain in the specialization is a template parameter that does not provide access to any nested tuples. The method is therefore delete'd. However, it should still be possible to call such methods and therefore the partial specialization is further specialized as follows to provide a matching for Domain and Domain2 in the above signature.

```
isl::typed::map<pair<Domain, Domain2>, Range>
```

While generating this specialization, the need for a further specialization will become apparent for methods applied to objects with nested tuples in the range. Since there are currently no functions in isl that specifically operate on

objects with doubly nested tuples, no further specializations of this sort will be required.

Another reason that the matching might fail is demonstrated by methods such as `deltas`. This method computes the differences between the range and the domain of an element in a map. This means the domain and range tuples need to be the same. The tuple kinds therefore also need to be the same. That is, the signature is as follows.

```
{ { Domain }, { { Domain, Domain } } };
```

Again, this does not match the partial type specialization `isl::typed::map<Domain, Range>`. The signature has a single `Domain` placeholder and it would need to map to both `Domain` and `Range` in the type specialization. A further `isl::typed::map<Domain, Domain>` therefore needs to be generated as well, along with further specializations of the previous sort. In total, six partial specializations of `isl::typed::map` currently get generated.

5.5 Template Argument Class Hierarchy

In a user application, it can sometimes be useful to consider some tuple kind to be a special case of some other kind. For example, a scalar could be considered to be a special kind of an array so that a function accepting an access to an array can also accept an access to a scalar, but not the other way around. In order to lift the subclass relationship to the level of the templated `isl` types (at least to some extent), an additional constructor is added to each specialization that accepts more specific specializations as input. In particular, for each template parameter of the type specialization a corresponding template parameter for this new constructor is introduced. The new parameter is then required to be a subclass of the original type template parameter. For example, for `isl::typed::map<Domain, Range>`, this constructor has the following form.

```
template <typename Arg1, typename Arg2,
          typename std::enable_if<
            std::is_base_of<Domain, Arg1>{} &&
            std::is_base_of<Range, Arg2>{},
          bool>::type = true>
map(const map<Arg1, Arg2> &obj) :
  isl::map(obj) {}
```

5.6 Limitations

Not all inconsistencies can be detected at compile time. For example, the input to the `isl_map_deltas` function needs to live in a space with identical tuple types. This still needs to be checked at run time since at compile time the two tuples can only be enforced to belong to the same kind. However, this compile-time check is already a major improvement over having no consistency checks at all. It would also be difficult to impose compile-time checks on specific tuples, since these are usually only defined at run time.

The compile-time checks can be circumvented by taking a pointer to an object and modifying the plain type part. This could happen accidentally in functions taking a pointer to a plain `isl` object, but it is fairly rare for `isl` objects to be passed by pointer. Since `isl` objects behave like values, with operations performed on an object returning a different object rather modifying the original object, they are usually also passed to functions by value.

6 Practical Experience

The first version of the templated C++ interface was developed in the context of `Tensor Comprehensions` (Vasilache et al. 2019), while a second version was developed in the context of `DTG` (Verdoolaege, Kudlur, et al. 2020) and will be made available as part of `isl`.[2] The second version is more mature and it is this more mature version that is described in this paper. The differences between the two versions are not described since the first version is unsupported and not widely used.

Moving from the plain C++ interface to the templated interface requires some changes to the code base, but this can be done incrementally. Clearly, the argument and/or return types of the functions where the consistency checks provided by the templated types are desired need to be adjusted accordingly. As long as automatic type deduction is used for local variables, i.e., they are declared `auto`, typically only minor changes are needed inside the affected functions. Any explicit type specification needs to be adjusted, but most constructor calls can be replaced prior to the switch to templated types by exploiting the renamed exports of Section 4.3. Occasionally, an extra variable needs to be introduced because the same variable was being used to hold two different types of values that still have the same type in the plain interface. In effect, the extra consistency checks provided by the templated types uncover the original inappropriate reuse of the same variable.

As of yet, no bugs have been uncovered in either `Tensor Comprehensions` or `DTG` as a direct result of switching to the templated interface. However, some bugs have been fixed in `DTG` before that would have been detected by the templated interface. Also, the main purpose is to make it *easier* for a developer to write correct code, through both the extra documentation and the compile-time consistency checks. However, since development of `Tensor Comprehensions` has ceased and since the use of the templated interface in `DTG` is new, this expected outcome has yet not been evaluated.

As can be expected, the use of more templates does somewhat increase the compilation time. A `make -j 10` after a

[2] The first version is available from https://github.com/facebookresearch/TensorComprehensions/pull/604. At the time of writing, the second version is only privately available, but is scheduled for upstreaming to the public repository before the 0.24 release.

`make clean` on `DTG` using gcc 9.3.0 on an 8-core i7-6700 increases the wallclock time from about 3m55s to about 4m07s after simply including the templated C++ header and to a further 4m11s after actually using it in parts of the code. On the same experiment, the total "user" time increases from about 22m30s to 23m40s and 24m35s. The size of the (stripped) main executable also increases from 3.0MB to 3.2MB.

7 Conclusions

The concept of spaces allows `isl` to perform more consistency checks of operations than other polyhedral libraries. However, these checks are only performed at run time and cannot be performed on "union" types. The templated C++ interface presented in this paper introduces a more fine-grained, user-controlled type system. In particular, it makes a distinction between spaces with zero, one or two tuples as well as between functions with one or two tuples. Additionally, it allows the user to define their own application-specific `isl` types, allowing more consistency checks to be performed at compile time.

Acknowledgments

The original plain C++ interface was contributed by Tobias Grosser and Michael Kruse. The linearization of the function types was implemented by Tobias Grosser. The idea for a templated interface was partly inspired by an unpublished C++ library developed by Armin Größlinger that provides operations for sets defined by formulas over polynomials and that allows for the specification of nested spaces using templates. The first version of the design and implementation of the templated C++ interface for `isl` introduced by this paper was created while the first author was a visiting researcher working for Facebook and the second author was a research engineer at Inria in the context of `Tensor Comprehensions`.

References

Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and David Wonnacott (Nov. 1996). *The Omega Library*. Tech. rep. University of Maryland.

Andreas Klöckner (2014). "Loo.Py: Transformation-based Code Generation for GPUs and CPUs." In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. ARRAY'14. Edinburgh, United Kingdom: ACM, 82:82–82:87. DOI: 10.1145/2627373.2627387.

Sunder Phani Kumar Nookala and Tanguy Risset (May 2000). *A Library for Z-polyhedral Operations*. Tech. rep. PI-1330. IRISA, Rennes, France.

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen (Oct. 2019). "The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically." In: *ACM Trans. Archit. Code Optim.* 16.4, 38:1–38:26. DOI: 10.1145/3355606.

Sven Verdoolaege (2010). "isl: An Integer Set Library for the Polyhedral Model." In: *Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

Sven Verdoolaege (Apr. 2011). "Counting Affine Calculator and Applications." In: *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*. Chamonix, France. DOI: 10.13140/RG.2.1.2959.5601.

Sven Verdoolaege (2016). *Presburger Formulas and Polyhedral Compilation*. DOI: 10.13140/RG.2.1.1174.6323.

Sven Verdoolaege and Tobias Grosser (Jan. 2012). "Polyhedral Extraction Tool." In: *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France. DOI: 10.13140/RG.2.1.4213.4562.

Sven Verdoolaege, Manjunath Kudlur, Rob Schreiber, and Harinath Kamepalli (Jan. 2020). "Generating SIMD Instructions for Cerebras CS-1 using Polyhedral Compilation Techniques." In: *10th International Workshop on Polyhedral Compilation Techniques (IMPACT'20)*. Bologna, Italy. DOI: 10.5281/zenodo.4295955.

Doran K. Wilde (1993). *A Library for doing polyhedral operations*. Tech. rep. 785. IRISA, Rennes, France, 45 p.