A Templated C++ Interface for isl

Sven Verdoolaege[1] Oleksandr Zinenko[2] Manjunath Kudlur[1]
Ron Estrin[1] Tianjiao Sun[1] Harinath Kamepalli[1]

[1]Cerebras Systems

[2]Google

January 20, 2021

Outline

Polyhedral Compilation

Polyhedral Compilation

Analyzing and/or transforming programs using the *polyhedral model*

Polyhedral Model

Abstract representation of a program

- instance based
  - ⇒ statement *instances*
  - ⇒ array *elements*
- compact representation based on polyhedra or similar objects
  - ⇒ integer points in unions of parametric polyhedra
  - ⇒ Presburger sets and relations
- parametric
  - ⇒ description may depend on constant symbols

Polyhedral Model

Typical constituents of program representation

- Instance Set
  - ⇒ the set of all statement instances
- Access Relations
  - ⇒ the array elements accessed by a statement instance
- Dependences
  - ⇒ the statement instances that depend on a statement instance
- Schedule
  - ⇒ the relative execution order of statement instances

Illustrative Example: Matrix Multiplication

```
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
S1:     C[i][j] = 0;
        for (int k = 0; k < K; k++)
S2:         C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

- Instance Set (set of statement instances)

$$\{\, \mathtt{S1}[i,j] : 0 \le i < M \wedge 0 \le j < N; \mathtt{S2}[i,j,k] : 0 \le i < M \wedge 0 \le j < N \wedge 0 \le k < K \,\}$$

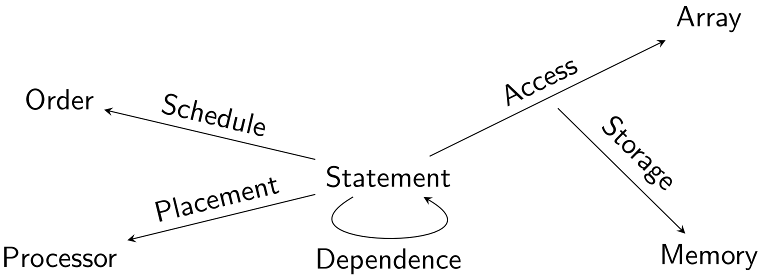
- Access Relations (accessed array elements; W: write, R: read)

$$W = \{\, \mathtt{S1}[i,j] \to \mathtt{C}[i,j]; \mathtt{S2}[i,j,k] \to \mathtt{C}[i,j] \,\}$$
$$R = \{\, \mathtt{S2}[i,j,k] \to \mathtt{C}[i,j]; \mathtt{S2}[i,j,k] \to \mathtt{A}[i,k]; \mathtt{S2}[i,j,k] \to \mathtt{B}[k,j] \,\}$$

- Schedule (relative execution order)

$$\{\, \mathtt{S1}[i,j] \to [i,j,0,0]; \mathtt{S2}[i,j,k] \to [i,j,1,k] \,\}$$

Polyhedral Objects



+ many more

Polyhedral Compiler

A sufficiently advanced polyhedral compiler needs to handle many kinds of polyhedral objects

This can cause confusion:

- exactly what kind of object does this function expect?
- does this operation on these objects make sense?

In statically typed languages (such as C++)
⇒ use types

What types are available in

- `PolyLib` (Wilde 1993),
- `Omega` (Kelly et al. 1996) and
- `isl` (V. 2010)?

Types Offered by Polyhedral Libraries: `PolyLib`

In `PolyLib`, every set or binary relation is represented by a `Polyhedron`.

⇒ no differentiation at compile time

⇒ even at run time, only dimensionality can be checked

Does it make sense to intersect these two objects?

```
4 5
1     1     0     0     0
1     0     1     0     0
1    -1     0     1    -1
1     0    -1     1    -1

1 5
0     1    -1     0     0
```

Types Offered by Polyhedral Libraries: `Omega`

In `Omega`, every set or binary relation is represented by a `Relation`.

⇒ no differentiation at compile time

⇒ at run time, differentiation between tuple size(s) as well as between

- sets

  ```
  { [i, j] : 0 <= i < n and 0 <= j < n }
  ```

- binary relations

  ```
  { [i] -> [i] }
  ```

Types Offered by Polyhedral Libraries: `isl` (plain C++)

In `isl`, every set is represented by an `isl::set` or an `isl::union_set` and every binary relation is represented by an `isl::map` or an `isl::union_map`.

⇒ differentiation between sets and binary relations at compile time

⇒ at run time, differentiation between tuple size(s) and tuple name(s) (for `isl::set` and `isl::map`)

```
{ S2[i, j, k] : 0 <= i < M and 0 <= j < N and 0 <= k < K }
{ S1[i, j] -> C[i, j] }
```

`isl::union_set` and `isl::union_map` objects may contain elements with different tuple sizes and/or names.

```
{ S1[i, j] -> C[i, j]; S2[i, j, k] -> C[i, j] }
```

⇒ no run-time checks

⇒ still maps *statement instances* to *array elements*

⇒ need for more find-grained types

Design Goals

- differentiation between different kinds of sets and binary relations

  ⇒ detect problems such as
    - access relation passed to function expecting dependence relation
    - range of access relation intersected with statement instances
- at compile time
- application specific
  - ⇒ `isl` provides general framework
  - ⇒ application defines concrete types
- compatible with plain C++ interface

  ⇒ allow gradual transition to templated interface
- easy to use

  ⇒ no annotations beyond what is strictly required

Basic Idea: Template Types

- Introduce template type for each plain type involving tuples
  - ⇒ same name
  - ⇒ different name space: `isl::typed`
- Every type has 0 or more template parameters, one for each tuple, specifying tuple *kind*

```
template <>
struct set<> : public isl::set { /* ... */ }
```

⇒ constraints on constant symbols, e.g., `{ : M, N, K > 0 }`

```
template <typename Domain>
struct set<Domain> : public isl::set { /* ... */ }
```

⇒ for example: `struct ST {}; isl::typed::set<ST>;`

```
template <typename Domain, typename Range>
struct map<Domain, Range> : public isl::map { /* ... */ }
```

⇒ for example: `struct AR {}; isl::typed::map<ST, AR>;`

Basic Idea: Template Types

- Every type has 0 or more template parameters, one for each tuple, specifying tuple *kind*

```
template <>
struct set<> : public isl::set { /* ... */ }
```

⇒ constraints on constant symbols, e.g., `{ : M, N, K > 0 }`

```
template <typename Domain>
struct set<Domain> : public isl::set { /* ... */ }
```

⇒ for example: `struct ST {}; isl::typed::set<ST>;`

```
template <typename Domain, typename Range>
struct map<Domain, Range> : public isl::map { /* ... */ }
```

⇒ for example: `struct AR {}; isl::typed::map<ST, AR>;`

  - Template type derived from plain type for interoperability
  - Constructor is `private` to avoid bypassing checks
  - Template argument corresponds to *kind* of tuple (e.g., statement, array), not specific tuple name/size

Basic Idea: Template Types

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

struct ST {}; struct AR {};
void f(const isl::typed::union_map<ST, AR> &access);
void g(const isl::typed::union_map<ST, ST> &dep);
void h(const isl::typed::union_map<ST, AR> &access) {
        f(access);
        g(access);
}
```

clang error:

```
error1.cc:10:2: error: no matching function for call to 'g'
        g(access);
        ^
error1.cc:6:6: note: candidate function not viable: no known
   ↪ conversion from 'const union_map<[...], AR>' to 'const
   ↪ union_map<[...], ST>' for 1st argument
void g(const isl::typed::union_map<ST, ST> &dep);
```

Basic Idea: Template Methods

- Introduce template method for each plain method taking and/or returning objects involving tuples
- Shared template parameter for shared tuple kind

```
template <typename Domain, typename Range>
struct map<Domain, Range> : public isl::map {
  /* ... */
  inline typed::map<Domain, Range> intersect_domain(
    const typed::set<Domain> &set) const;
  inline typed::map<Domain, Range> intersect_range(
    const typed::set<Range> &set) const;
  template <typename Range2>
  inline typed::map<Domain, Range2> apply_range(
    const typed::map<Range, Range2> &map2) const;
}
```

⇒ exploit template argument deduction

Basic Idea: Template Methods

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

struct ST {}; struct AR {};
void h(const isl::typed::union_map<ST, AR> &access,
       const isl::typed::union_set<ST> instances) {
    access.intersect_domain(instances);
    access.intersect_range(instances);
}
```

clang error:

```
error2.cc:9:9: error: no matching member function for call to '
   ↪ intersect_range'
        access.intersect_range(instances);
        ~~~~~~~~~~~~~~~~~~~~~~
isl/typed_cpp.h:10731:42: note: candidate function not viable: no
   ↪ known conversion from 'const union_set<ST>' to 'const union_set
   ↪ <AR>' for 1st argument
  inline typed::union_map<Domain, Range> intersect_range(const typed
```

Further Specializations

Consider storage map (from access to memory)

`{ [S2[i, j, k] -> C[i, j]] -> Mem_C[i, j] }`

(`S2[i, j, k]` and `C[i, j]` are nested tuples in domain tuple)

How to extract mapping from statement to memory `{ S2[i, j, k] -> Mem_C[i, j] }`?
⇒ `isl::map::domain_factor_domain`

Corresponding template method cannot be made available in `map<Domain, Range>`

- ⇒ Domain not specific enough
- ⇒ delete'd from `map<Domain, Range>`
- ⇒ included in more specific specialization

```
template <typename Domain, typename Range, typename Range2>
struct map<pair<Domain, Range>, Range2> : public isl::map {
  /* ... */
  inline typed::map<Domain, Range2> domain_factor_domain() const
}
```

Explicit Template Arguments: “Constructors”

Consider `isl::set::universe`

⇒ construct a universe `isl::set` from a description of the tuple (specified by an `isl::space`)

```
template <typename Domain>
struct set<Domain> : public isl::set {
  static inline typed::set<Domain> universe(
    const typed::space<Domain> &space);
}
```

Use:

```
void f(const isl::typed::space<ST> &space) {
        auto set = isl::typed::set::universe<ST>(space);
}
```

⇒ both type and template arguments need to be spelled out explicitly

Explicit Template Arguments: “Constructors”

```
void f(const isl::typed::space<ST> &space) {
        auto set = isl::typed::set::universe<ST>(space);
}
```

⇒ both type and template arguments need to be spelled out explicitly

Introduce alternative name `isl::space::universe_set`

```
template <typename Domain>
struct space<Domain> : public isl::space {
  inline typed::set<Domain> universe_set() const;
}
```

Use:

```
void f(const isl::typed::space<ST> &space) {
        auto set = space.universe_set();
}
```

Explicit Template Arguments: Set Range Tuple

`isl::map::set_range_tuple` replaces tuple identifier of range of map

⇒ changes meaning of tuple

⇒ potentially changes tuple kind

⇒ tuple kind needs to be specified explicitly

```
template <typename Domain, typename Range>
struct map<Domain, Range> : public isl::map {
  template <typename Range2>
  inline typed::map<Domain, Range2> set_range_tuple(
    const std::string &id) const;
}
```

⇒ no template argument deduction on `Range2`

Template Argument Class Hierarchy

Some specializations may be special cases of other specializations.

⇒ allow users to define class hierarchy on template arguments

⇒ copy relationship to corresponding template types

For example, user may define a function that takes either array elements or memory elements

⇒ derive array and memory template argument from common base class

```
template <typename Domain, typename Range>
struct map<Domain, Range> : public isl::map {
  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) {}
}
```

Implementation and Experience

Two “independent” implementations

1. within the context of `Tensor Comprehensions` (Vasilache et al. 2019)
    - external to `isl`
    - automatically generated from parsed plain C++ headers
2. within the context of `DTG` (V. et al. 2020)
    - part of `isl` (publicly available soon)
    - automatically generated from parsed C headers

Transition to templated interface benefits from

- automatic type deduction for local variables (`auto`)
- extra local variables to store different kinds of data

Benefits	compile-time checks documentation
Drawbacks	minor increase in compilation time minor increase in binary size

Conclusion

Templated C++ interface for `isl`

- ⇒ compile-time checks using fine-grained, user controlled types
- ⇒ extra information in template arguments allows for checks that were not even available at compile time
- ⇒ available soon

References I

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

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

V., Sven (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`.

V., Sven, 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`.

References II

Vasilache, Nicolas, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven V., 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.

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