A Templated C++ Interface for isl

Sven Verdoolaege\textsuperscript{1}  Oleksandr Zinenko\textsuperscript{2}  Manjunath Kudlur\textsuperscript{1}  Ron Estrin\textsuperscript{1}  Tianjiao Sun\textsuperscript{1}  Harinath Kamepalli\textsuperscript{1}

\textsuperscript{1}Cerebras Systems  \textsuperscript{2}Google

January 20, 2021
Outline

1 Introduction and Motivation
   - Polyhedral Compilation
   - Polyhedral Object Types

2 Templated Interface
   - Design Goals
   - Basic Idea
   - Further Specializations
   - Explicit Template Arguments
   - Template Argument Class Hierarchy
   - Implementation and Experience

3 Conclusion
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

```java
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)
  \[
  \{ S1[i,j] : 0 \leq i < M \land 0 \leq j < N ; S2[i,j,k] : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K \} 
  \]

- **Access Relations** (accessed array elements; \( W \): write, \( R \): read)
  \[
  W = \{ S1[i,j] \rightarrow C[i,j] ; S2[i,j,k] \rightarrow C[i,j] \} 
  \]
  \[
  R = \{ S2[i,j,k] \rightarrow C[i,j] ; S2[i,j,k] \rightarrow A[i,k] ; S2[i,j,k] \rightarrow B[k,j] \} 
  \]

- **Schedule** (relative execution order)
  \[
  \{ S1[i,j] \rightarrow [i,j,0,0] ; S2[i,j,k] \rightarrow [i,j,1,k] \} 
  \]
Polyhedral Objects

+ many more
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?

\[
\begin{array}{cccccc}
4 & 5 \\
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 \\
1 & -1 & 0 & 1 & -1 & 0 \\
1 & 0 & -1 & 1 & -1 & 0 \\
1 & 5 & & & & \\
0 & 1 & -1 & 0 & 0 & 0 \\
\end{array}
\]
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
    \[
    \{ \left[ i, j \right] : 0 \leq i < n \text{ and } 0 \leq j < n \} 
    \]
  - binary relations
    \[
    \{ \left[ i \right] \rightarrow \left[ i \right] \} 
    \]
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 fine-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*

```cpp
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:

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

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

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

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

  ```cpp
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

```c++
#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

```cpp
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

```cpp
#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::union_set<Domain> &uset) const;
```

Further Specializations

Consider storage map (from access to memory)

\[
\{ [S2[i, j, k] \rightarrow C[i, j]] \rightarrow Mem_C[i, j] \}
\]

(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] \rightarrow Mem_C[i, j] \}?

\[ \Rightarrow isl::map::domain_factor_domain \]

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

\[ \Rightarrow \text{Domain not specific enough} \]

\[ \Rightarrow \text{delete'd from map<Domain, Range>} \]

\[ \Rightarrow \text{included in more specific specialization} \]

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`)

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

Use:

```cpp
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”

```cpp
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`

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

Use:

```cpp
void f(const isl::typed::space<ST> &space) {
    auto set = space.universe_set_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


References II
