$Header : /cvsroot/latex-beamer/latex-beamer/emulation/beamertexpower.s$
$53 : 07tantauExp$

TPcolor

A Static C++ Object-Oriented Programming (SCOOP) Paradigm

N. Burrus, A. Duret-Lutz, T. Géraud, D. Lesage, R. Poss
`www.lrde.epita.fr`

MPOOL’03, October 26, 2003

Contents

Context

- Scientific numerical computing
  - ▷ image processing library: Olena
    `http://olena.lrde.epita.fr`
- Requirements:
  - ▷ expressiveness
  - ▷ algorithms ↔ functions ↔ mathematical abstractions
  - ▷ efficiency
- Possible strategies:
  - ▷ to find the right language
  - ▷ to extend an existing language
  - ▷ to use a native language (C++)

OOP and GP in C++

- **OOP: Object-Oriented Programming**
  - ✓ named typing
    - ▷ named classes
    - ▷ explicit inheritance
  - ✓ class hierarchies
  - ✓ inclusion and coercion polymorphism
  - ✓ overloading and overriding
  - ✗ run-time overhead due to polymorphic methods (`virtual`)

- **GP: Generic Programming**

  ✓ structural typing (`template`)
  ✓ abstraction through `template` constructs

  ✓ run-time efficiency due to compile-time computations
    ▷ typing
    ▷ code specialization

  ✓ meta-programming

  ✗ closed-world assumption
  ✗ lack of type constraints on template parameters
  ✗ exact matching on template arguments ➔ limited overloading

Our objective: SCOOP

To mix OOP features with GP efficiency

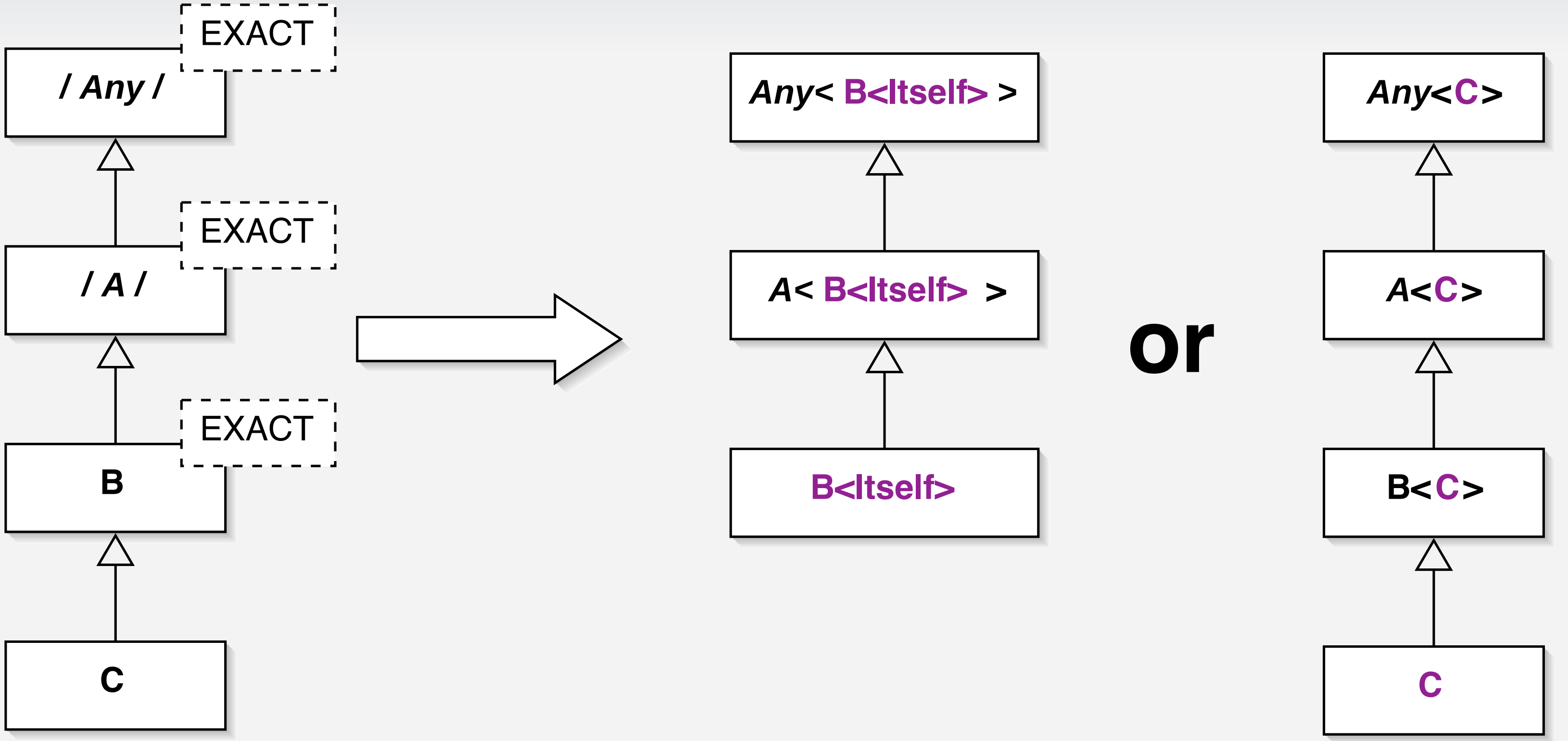
- ✓ class hierarchies **without** type information loss
  - ▷ method overriding
  - ▷ method covariance
  - ▷ multiple inheritance
- ✓ method dispatch **without** run-time overhead
- ✓ overloading like in OOP

Description of SCOOP

- ▷ Static hierarchies
- ▷ Abstract classes and interfaces
- ▷ Expressing constraints on types
- ▷ Argument covariance
- ▷ Polymorphic typedefs
- ▷ Multimethods

Static Hierarchies

- Generalization of the **?** trick
- Class parameter `EXACT` ↔ type of the object effectively instantiated



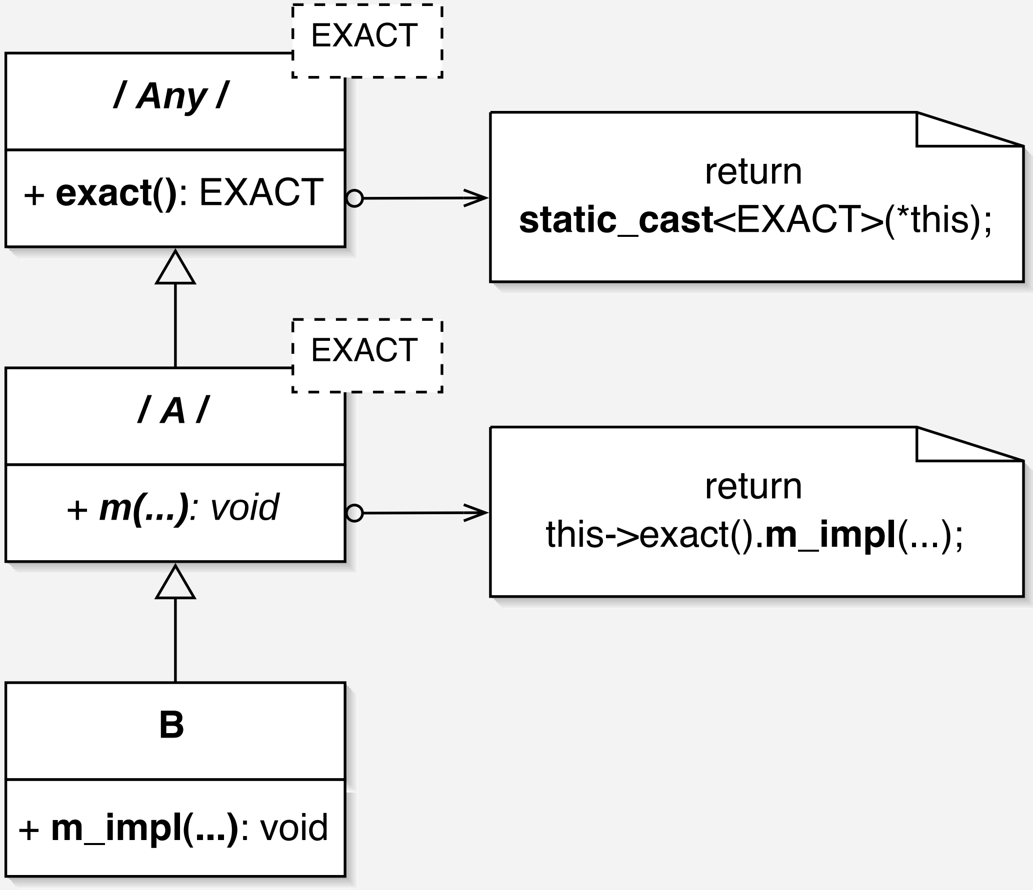
Meta Hierarchy

Instantiation of B

Instantiation of C

- ▷ Abstract, **concrete extensible** and final classes
- ▷ Meta-hierarchies: unfolding into **distinct effective hierarchies**
  ➔ distinct base classes

Abstract Classes



- `exact()` mechanism
- manual dispatch
- full-static dispatch

Expressing Constraints on Function Types

```
void foo(A& arg)
{}


void foo(B& arg)
{}
```

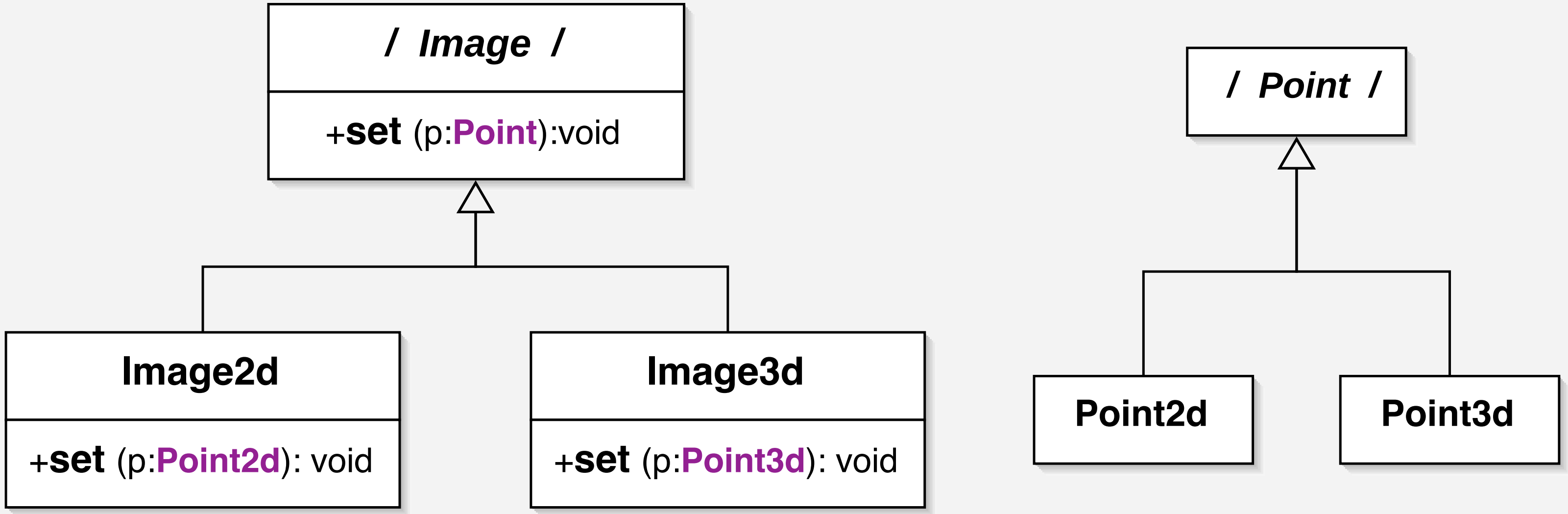
Classical OOP

```
template <class EXACT>
void foo(A<EXACT>& arg)
{}

template <class EXACT>
void foo(B<EXACT>& arg)
{}
```

SCOOP

- Similar code in OOP and SCOOP
- EXACT must conform to A<EXACT>
  ➔Close to F-bounded polymorphism
- Contrary to GP, **overloading remains possible**

Method Covariance



- Requires dynamic checks in OOP (using `dynamic_cast` in C++)

Simple approach

```
template <class Exact>
struct Image
  : public Any<Exact>
{
  template <class P>
  void set(Point<P>& p) {
    exact().set_impl(p.exact());
  }
};
```

```
template <class Exact>
struct Image2d
  : public Image<..>
{
  template <class P>
  void set_impl(Point2d<P>& p) {
    // ...
  }
};
```

- **Compilation** fails if subclasses cannot handle the given point type

Polymorphic typedefs

```
template <class Exact>
struct Image
  : public Any<Exact>
{
  typedef typename
    traits<Exact>::point_type
      point_type;

  point_type get() {
    // calls get_impl()
  }
};
```

```
struct traits<Image2d... {
  typedef Point2d<> point_type;
}

template <class Exact>
struct Image2d
  : public Image<..>
{
  point_type get_impl() {
    // ...
  }
};
```

- A parallel hierarchy of **traits** (**?**) is defined
- With some additional tools we can get **virtual types**

Multimethods

```
template <class I1, class I2>
void algo1(Image<I1>& ima1,
           Image<I2>& ima2)
{
  // ima1 and ima2 are
  // downcasted manually
  algo2(ima1.exact(),
        ima2.exact());
}
```

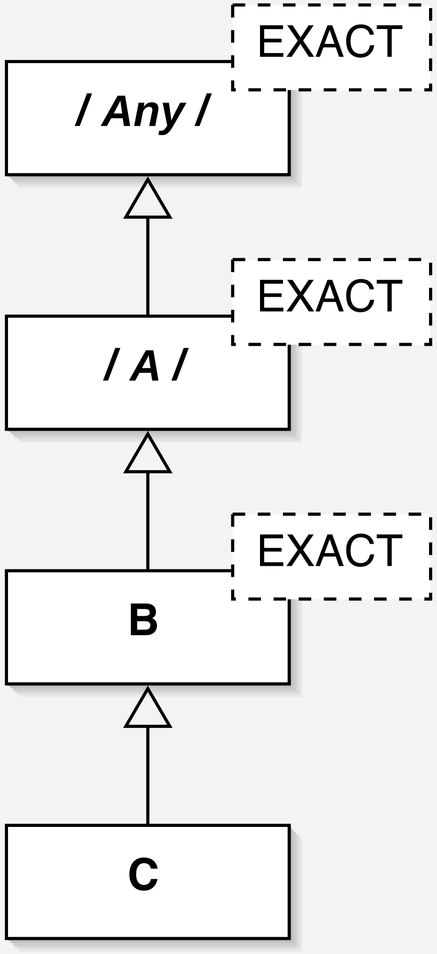
```
template <class I1, class I2>
void algo2(Image<I1>& ima1,
           Image<I2>& ima2)
{}

template <class I1, class I2>
void algo2(Image2d<I1>& ima1,
           Image3d<I2>& ima2)
{}

// other versions of algo2
```

- Only **`algo2(Image, Image)`** can be called in classical C++
- Multimethod dispatch is performed by overloading in SCOOP

Conclusion

- ✗ Some GP drawbacks
  - ▷ Closed world
  - ▷ Longer compilation time than OOP
  - ▷ Error messages for the **user** better than GP but worse than OOP
- ✓ Complete paradigm combining OOP and GP benefits
  - ▷ High expressiveness
  - ▷ High performance
- ✓ Suitable for large scale applications
  - ▷ Design your OO application
  - ▷ Write it down in SCOOP
  - ▷ Implemented in Olena

Static Hierarchy Implementation



```
// Hierarchy apparel

struct Itself
{ };

// find_exact utility macro
#define find_exact(Type) //...

template <class EXACT>
class Any
{
  // ...
};
```

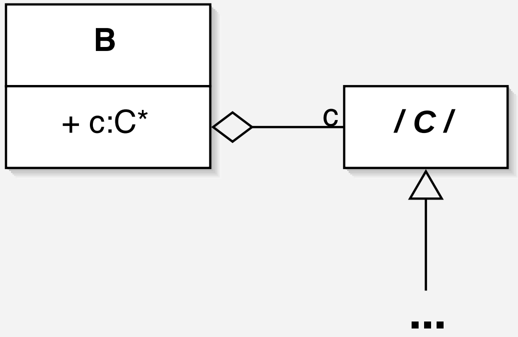
```
// Hierarchy

// purely abstract class
template <class EXACT>
class A: public Any<EXACT>
{
  // ...
};

// extensible concrete class
template <class EXACT=Itself>
class B
 :public A<find_exact(B)>
{
  // ...
};

// final class
class C: public B<C>
{
  // ...
};
```

`find_exact` mechanism

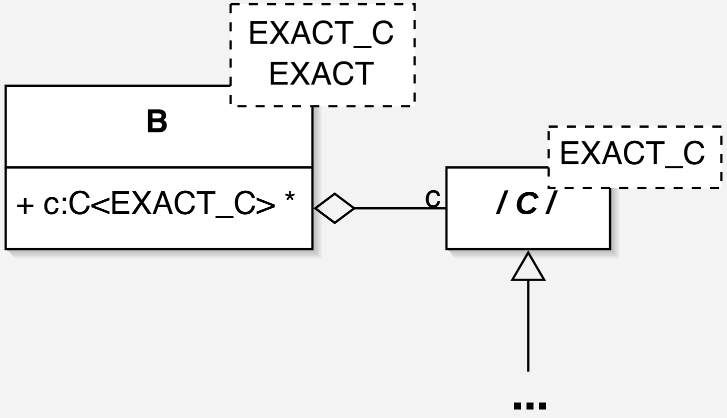
```
let FindExact($Type$, $EXACT$) =
  if $EXACT \neq Itself$
    then $EXACT$
    else $Type < Itself >$
```

```
// default version
template <class T, class EXACT>
struct FindExact
{
  typedef EXACT ret;
};

// version specialized for EXACT=Itself
template <class T>
struct FindExact<T, Itself>
{
  typedef T ret;
};

// find_exact utility macro
#define find_exact(Type) typename \
    FindExact<Type<Exact>, Exact>::ret
```

Associations



aggregation in classical OOP



aggregation in SCOOP

- ✓ very close to OOP
- ✓ stronger typing than GP design patterns in (**?**)
- ✗ no way to change type of the aggregated object at run-time

Related work

- ▷ dynamic dispatch overhead:
  - ✓ emulated by the Barton & Nackman trick (**?**)

- ▷ lack of type constraint in GP:
  - ✓ structural check (**?**)
  - ✓ Barton & Nackman trick (**?**)

- ▷ OOP design patterns translated into GP (**?**)

- ▷ virtual types into GP (**?**)