Type-erased interfaces in dune-functions

Carsten Gräser

2015-09-28
Basic functionality

- Functions mapping Domain to Range
- Differentiable functions
- Evaluate grid functions in local coordinates
Requirements on function interfaces

Basic functionality

- Functions mapping Domain to Range
- Differentiable functions
- Evaluate grid functions in local coordinates

What else?

- Zero overhead if possible
- Selection at runtime if needed
- Easy to use
- Easy to implement new functions
- Easy to extend
template <class D, class R>
class ConstantFunction:
    public VirtualFunction<D, R>
{
    public:

    ConstantFunction(const R& constant):
        constant_(constant)
    {}

    // You cannot override this
    virtual void evaluate(const D& x, R& y) const final
    {
        y = constant_;
    }

    private:
    const R constant_;
Using the function

The cumbersome dune-common way

```cpp
template<class F>
void foo(const F & f)
{
    // Virtual call if F is interface type
    // No virtual call if F is derived type
    // because we used final.
    FieldVector<double, 1> y;
    f.evaluate(x, y);
    y *= h;
}
```

The C++ way

```cpp
template<class F>
void foo(F && f)
{
    y = f(x)*h;
}
...
assembleRHS([](double x){return 1;});
```
Why operator()?

operator() is ...

... not more expensive, the extra copy is removed by RVO

... more readable

... more flexible
  - No need for exact range type
  - \texttt{int(double)} will work with \texttt{FV<double,1>}
  - Works with free functions, lambda expressions

How to select functions at runtime?
Why operator()?

operator() is ...

... not more expensive, the extra copy is removed by RVO

... more readable

... more flexible
  ▶ No need for exact range type
  ▶ `int(double)` will work with `FV<double,1>`
  ▶ Works with free functions, lambda expressions

How to select functions at runtime?

    // Can hold any function with compatible signature
    std::function<double(double)> f;

    f = [](double x){return x+1;};
    double y = f(-42);
Store any function object that...

- ...can be called with Domain
- ...returns values convertible to Range
- ...is a free function, functor, lambda, ...

How does it work?

- Duck typing
- Type erasure
- Value semantics
- Small object optimization
std::function<Range(Domain)>

Store any function object that...
  ▶ ...can be called with Domain
  ▶ ...returns values convertible to Range
  ▶ ...is a free function, functor, lambda, ...

How does it work?
  ▶ Duck typing
  ▶ Type erasure
  ▶ Value semantics
  ▶ Small object optimization

Adopt the same mechanism in dune-functions
  ▶ DifferentiableFunction<Range(Domain)>
  ▶ GridFunction<Range(Domain), ...>
  ▶ LocalFunction<Range(Domain), ...>
Duck typing interfaces

“When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.”

J. W. Riley

No base class, just implement ...

// ... for a simple function
f(x);

// ... for a differentiable function
f(x);
auto df = derivative(f);
df(x);

// ... for a grid function
f(x);
auto fLocal = localFunction(f);
fLocal.bind(element);
fLocal(xLocal);
Duck typing interfaces

“’When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.’”

J. W. Riley

No base class, just implement ...

```
// ... for a simple function
f(x);

// ... for a differentiable function
f(x);
auto df = derivative(f);
df(x);

// ... for a grid function
f(x);
auto fLocal = localFunction(f);
fLocal.bind(element);
fLocal(xLocal);
```

What about type safety?
Duck typing interfaces

“’When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.’”

J. W. Riley

No base class, just implement ...

```cpp
// ... for a simple function
f(x);

// ... for a differentiable function
f(x);
auto df = derivative(f);
df(x);

// ... for a grid function
f(x);
auto fLocal = localFunction(f);
fLocal.bind(element);
fLocal(xLocal);
```

What about type safety? Wait a moment!
Use internal dynamic polymorphism

```cpp
template<class Range, class Domain>
struct Function<Range(Domain)> {

    template<class F>
    Function(F&& f) :
        f_(new FunctionWrapper<Range<Domain>, F>(f))
    {}

    Range operator()(const Domain& x) const
    {
        return f_->operator()(x);
    }

    FunctionWrapperBase<Range<Domain>>* f_;}
```
template<class Range, class Domain>
struct FunctionWrapperBase<Range(Domain)> {
    virtual Range operator() (const Domain& x) const = 0;
};

template<class Range, class Domain, class F>
struct FunctionWrapper<Range(Domain), F> :
    public FunctionWrapperBase<Range(Domain)>
{
    FunctionWrapper(const F& f) : f_(f) {}

    virtual Range operator() (const Domain& x) const {
        return f_(x);
    }

    F f_;  
};
**Value semantics**

- Functions are stored by value
- Functions are returned by value

**Small object optimization**

- Avoid allocation to create wrapper
- Use a static small object buffer on the stack

**Implementation**

- Non-intrusive, duck typing for stored function
- Easy to implement new type erasure classes
- Very similar to Adobe poly by Sean Parent
No base class, no type safety?

- We don’t care for the type safety...
- ...we care for the interface!
- Just check if the type does what we need.
On type safety

No base class, no type safety?

- We don’t care for the type safety...
- ...we care for the interface!
- Just check if the type does what we need.

Concepts

- Describe what you want to do with the type
- List of valid expressions
- Requirements on return types
- Check concept in specialization, static_assert, ...
On type safety

No base class, no type safety?

- We don’t care for the type safety...
- ...we care for the interface!
- Just check if the type does what we need.

Concepts

- Describe what you want to do with the type
- List of valid expressions
- Requirements on return types
- Check concept in specialization, static_assert, ...
- Language support in C++17
On type safety

No base class, no type safety?

- We don’t care for the type safety...
- ...we care for the interface!
- Just check if the type does what we need.

Concepts

- Describe what you want to do with the type
- List of valid expressions
- Requirements on return types
- Check concept in specialization, static_assert, ...
- Language support in C++17
- Can be done in C++11 with some helpers
// Anything that can be called with Args

template<class... Args>
struct Callable {

    template<class F>
    auto require(F&& f) -> decltype(
        f(std::declval<Args>()...) )

};

Inspired by concept checks in range-v3 by Eric Niebler
template<class Signature> struct Function;

// A function mapping Domain to Range
template<class Range, class Domain>
struct Function<Range(Domain)>
    : Refines<Callable<Domain>>
{

template<class F>
auto require(F&& f) -> decltype(
    requireConvertible<Range>(f(std::declval<Domain>())))
);

}
// A differentiable function mapping Domain to Range
// The derivative range is derived from the traits

template<
    class Range,
    class Domain,
    template<class> class DerivativeTraits>
struct DifferentiableFunction<
    Range(Domain),
    DerivativeTraits>
    : Refines<Function<Range(Domain)>>
{
    using DerivativeSignature = ...

    template<class F>
    auto require(F&& f) -> decltype(
        derivative(f),
        requireConcept<Function<DerivativeSignature>>(derivative(f))
    );
};
Using the interface

template<class F>
void foo(F&& f)
{
    using namespace Dune::Functions::Concept;
    using Signature = Range(Domain);

    // Get a nice compiler error for inappropriate F
    static_assert(
        models<DifferentiableFunction<Signature>, F>(),
        "Type does not model Function concept");

    // Store f in polymorphic ...
    Functions::DifferentiableFunction<Signature> f_(f);

    // ... or non-polymorphic type
    // F f_(f);

    // use f
    auto df = derivative(f);
    df(x);
}
// More: www.dune-project.org/modules/dune-functions