Archive of Numerical Software
vol. 5, no. 1, 2017, pages 95–110
DOI: 10.11588/ans.2017.1.27683

The interface for functions in the dune-functions module

Christian Engwer[1], Carsten Gräser[2], Steffen Müthing[3], and Oliver Sander[4]

[1]Universität Münster, Institute for Computational und Applied Mathematics, christian.engwer@uni-muenster.de
[2]Freie Universität Berlin, Institut für Mathematik, graeser@mi.fu-berlin.de
[3]Universität Heidelberg, Institut für Wissenschaftliches Rechnen, steffen.muething@iwr.uni-heidelberg.de
[4]TU Dresden, Institute for Numerical Mathematics, oliver.sander@tu-dresden.de

Received: February 5th, 2016; **final revision:** August 1st, 2016; **published:** March 6th, 2017.

Abstract: The `dune-functions` DUNE module introduces a new programmer interface for discrete and non-discrete functions. Unlike the previous interfaces considered in the existing DUNE modules, it is based on overloading `operator()`, and returning values by-value. This makes user code much more readable, and allows the incorporation of newer C++ features such as lambda expressions. Run-time polymorphism is implemented not by inheritance, but by type erasure, generalizing the ideas of the `std::function` class from the C++11 standard library. We describe the new interface, show its possibilities, and measure the performance impact of type erasure and return-by-value.

1 Introduction

Ever since its early days, DUNE [2, 1] has had a programmer interface for functions. This interface was based on the class `Function`, which basically looked like

C++ code

```
template <class Domain, class Range>
class Function
{
  public:
    void evaluate(const Domain& x, Range& y) const;
};
```

and is located in the file `dune/common/function.hh`. This class was to serve as a model for duck typing, i.e., any object with its interface would be called a function. A main feature was that the result of a function evaluation was *not* returned as a return value. Rather, it was returned using a by-reference argument of the `evaluate` method. The motivation for this design decision was run-time efficiency. It was believed that returning objects by value would, in practice, lead to too many unnecessary temporary objects and copying operations.

Unfortunately, the old interface lead to user code that was difficult to read in practice. For example, to evaluate the n-th Chebycheff polynomial $T(x) = \cos(n \arccos(x))$ using user methods `my_cos` and `my_arccos` would take several lines of code:

C++ code

```
double tmp1,result;
my_arccos.evaluate(x,tmp1);
my_cos.evaluate(n*tmp1, result);
```

Additionally, C++ compilers have implemented return-value optimization (which can return values by-value without any copying) for a long time, and these implementations have continuously improved in quality. Today, the speed gain obtained by returning result values in a by-reference argument is therefore not worth the clumsy syntax anymore (we demonstrate this in Section 4). We therefore propose a new interface based on `operator()` and returning objects by value. With this new syntax, the Chebycheff example takes the much more readable form

C++ code

```
double result = my_cos(n*my_arccos(x));
```

To implement dynamic polymorphism, the old functions interface uses virtual functions. There is an abstract base class

C++ code

```
template <class Domain, class Range>
class VirtualFunction :
  public Function<const Domain&, Range&>
{
public:
  virtual void evaluate(const Domain& x, Range& y) const = 0;
};
```

in the file `dune/common/function.hh`. User functions that want to make use of run-time polymorphism have to derive from this base class. Calling such a function would incur a small performance penalty [3], which may possibly be avoided by compiler devirtualization [5].

The C++ standard library, however, has opted for a different approach. Instead of deriving different function objects from a common base class, no inheritance is used at all. Instead, any object that implements `operator()` not returning `void` qualifies as a function by duck typing. If a function is passed to another class, the C++ type of the function is a template parameter of the receiving class. If the type is not known at compile time, then the dedicated wrapper class

C++ code

```
namespace std
{
  template< class Range, class... Args >
  class function<Range(Args...)>;
}
```

is used. This class uses type erasure to allow to handle function objects of different C++ types through a common interface. There is a performance penalty for calling functions hidden within a `std::function`, comparable to the penalty for calling virtual functions.

The `dune-functions` module [4] picks up these ideas and extends them. The class template `std::function` works nicely for functions that support only pointwise evaluation. However, in a finite element context, a few additional features are needed. First of all, functions frequently need to supply derivatives as well. Also, for functions defined on a finite element grid, there is typically no single coordinate system of the domain space. Function evaluation in global coordinates is possible, but expensive; such functions are usually evaluated in local coordinates of a given

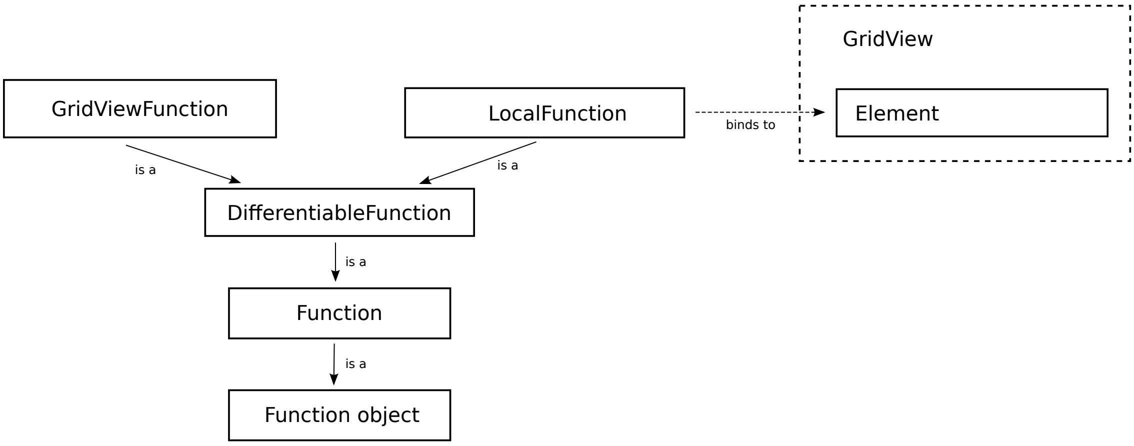

Figure 1: Diagram of the various function interfaces

element. `dune-functions` solves this by introducing `LocalFunction` objects, which can be *bound* to grid elements. When bound to a particular element, they behave just like a `std::function`, but using the local coordinate system of that element.

The paper is structured as follows. The main building blocks for the proposed function interfaces, viz. function objects, concept checks, and type erasure, are introduced in Section 2. Those techniques are then applied to implement the extended interfaces for differentiable functions and grid functions in Section 3. Finally, we present comparative measurements for the current and the proposed interface in Section 4. The results demonstrate that the increased simplicity, flexibility, and expressiveness do not have a negative performance impact.

2 Building blocks for function interfaces

The programmer interface for global functions is given as a set of model classes. These form a conceptual hierarchy (shown in Figure 1), even though no inheritance is involved at all. Implementors of the interface need to write classes that have all the methods and behavior specified by the model classes. This approach results in flexible code. Concept checking is used to produce readable error messages. The following sections explain the individual ideas in detail.

2.1 Function objects and functions

The C++ language proposes a standard way to implement functions. A language construct is called a *function object* if it is an ordinary function, a function pointer, or an object (or reference to an object) of a class providing an `operator()`. In the following we will adopt the common convention to call the latter a *functor*. In the following we will denote a function object as *function* if such a call does not return `void`. In other words, a function `foo` is anything that can appear in an expression of the form

C++ code

```
auto y = foo(x);
```

for an argument `x` of suitable type. Examples of such constructs are free functions

C++ code

```
double sinSquared(double x)
{
  return std::sin(x) * std::sin(x);
}
```

lambda expressions

C++ code

```
auto sinSquaredLambda = [](double x){return std::sin(x) * std::sin(x); };
```

functors

C++ code

```
struct SinSquared
{
  double operator()(double x)
  {
    return std::sin(x) * std::sin(x);
  }
};
```

and other things like bind expressions.

All three examples are functions in the above given sense, i.e., they can be called:

C++ code

```
    double a = sinSquared(3.14);       // free function
    double b = sinSquaredLambda(3.14); // lambda expression
    SinSquared sinSquaredObject;
    double c = sinSquaredObject(3.14); // functor
```

Argument and return value do not have to be `double` at all, any type is possible. They can be scalar or vector types, floating point or integer, and even more exotic data like matrices, tensors, and strings.

To pass a function as an argument to a C++ method, the type of that argument must be explicitly stated in the signature of the called method, or it must be a template parameter.

C++ code

```
template <typename F>
foo(F&& f)
{
  std::cout << "Value of f(42): " << f(42) << std::endl;
}
```

Any of the example functions from above can be used as an argument of the method `foo`:

C++ code

```
foo(sinSquared);            // call with a free function
foo(sinSquaredLambda);      // call with a lambda expression
foo(sinSquaredObject);      // call with a functor
```

2.2 Concept checks

The calls to `F` are fast, because the function calls can be inlined. On the other hand, it is not clear from the interface of `foo` what the signature of `F` should be. What's worse is that if a function object type with the wrong signature is passed, the compiler error will not occur at the call to `foo` but only where `F` is used, which may be less helpful. To prevent this, `dune-functions` provides light-weight concept checks for the concepts it introduces. For example, the following alternative implementation of `foo` checks whether `F` is indeed a function in the sense given above

C++ code

```cpp
template<class F>
void foo(F&& f)
{
  using namespace Dune;
  using namespace Dune::Functions;

  // Get a nice compiler error for inappropriate F
  static_assert(models<Concept::Function<Range(Domain)>, F>(),
      "Type does not model function concept");

  std::cout << "Value of f(42): " << f(42) << std::endl;
}
```

If `foo` is instantiated with a type that is not a function, say, an `int`, then a readable error message is produced. For example, for the code

C++ code

```cpp
foo(1);     // The integer 1 is not a function object
```

GCC-4.9.2 prints the error message (file paths and line numbers removed)

```
In instantiation of 'void foo(F&&) [with F = int]':
  required from here
error: static assertion failed: Type does not model function concept
    static_assert(models<Function<Range(Domain)>, F>(),
    ^
```

The provided concept checking facility is based on a list of expressions that a type must support to model a concept. The implementation is based on the techniques proposed by E. Niebler [8] and implemented in the range-v3 library [7]. While the `dune-common` module provides the general concept checking facility including the `models()` function that allows to check if a type models a concept, the definitions of the function concepts discussed in this paper are contained in the `dune-functions` module. This includes the concept `Function<Range(Domain)>` for simple functions as introduced above, as well as `DifferentiableFunction<Range(Domain)>`, `GridViewFunction<Range(Domain), GridView>`, and `LocalFunction<Range(Domain), LocalContext>` for the extended function interfaces discussed in Section 3.

2.3 Type erasure and `std::function`

Sometimes, the precise type of a function is not known at compile-time but selected depending on run-time information. This behavior is commonly referred to as *dynamic dispatch*. The classic way to implement this uses inheritance and the `virtual` keyword: All classes implementing functions must inherit from a common base class, and a pointer to this class is then passed around instead of the function itself.

This approach has a few disadvantages. For example, all functors must live on the heap, and a heap allocation is needed for each function construction. Secondly, in a derived class, the return value of `operator()` must match the return value used in the base class (weaker rules hold for pointer or reference types, which are not of interest to us, though). However, it is frequently convenient to also allow return values that are *convertible* to the return value of the base class. This is not possible in C++. As a third disadvantage, interfaces can only be implemented intrusively, and having one class implement more than a single interface is quite complicated.

The C++ standard library has therefore chosen type erasure over inheritance to implement run-time polymorphism. Starting with C++11, the standard library contains a class [6, 20.8.11]

C++ code

```cpp
namespace std
{
  template< class Range, class... Args >
  class function<Range(Args...)>
}
```

that wraps all functions that map a type `Domain` to a type (convertible to) `Range` behind a single C++ type. A much simplified implementation looks like the following:

C++ code

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

  FunctionWrapperBase<Range<Domain>>* f_;
};
```

The classes `FunctionWrapper` and `FunctionWrapperBase` look like this:

C++ code

```cpp
template<class Range, class Domain>
struct FunctionWrapperBase<Range(Domain)>
{
  virtual Range operator() (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() (Domain x) const
  {
    return f_(x);
  }

  F f_;
};
```

Given two types `Domain` and `Range`, any function object that accepts a `Domain` as argument and returns something convertible to `Range` can be stored in a `std::function<Range(Domain)>`. For example, reusing the three implementations of $\sin^2(x)$ from Section 2.1, one can write

C++ code

```
std::function<double(double)> polymorphicF;
polymorphicF = sinSquared;          // assign a free function
polymorphicF = sinSquaredLambda;    // assign a lambda expression
polymorphicF = SinSquared();        // assign a functor
double a = polymorphicF(3.14); // evaluate
```

Note how different C++ constructs are all assigned to the same object. One can even use

C++ code

```
polymorphicF = [](double x) -> int { return floor(x); };
   // okay: int can be converted to double
```

but not

C++ code

```
polymorphicF = [](double x) -> std::complex<double>
   { return std::complex<double>(x,0); };
   // error: std::complex<double> cannot be converted to double
```

Looking at the implementation of `std::function`, one can see that virtual functions are used *internally*, but it is completely hidden to the outside. The copy constructor accepts any type as argument. In a full implementation the same is true for the move constructor, and the copy and move assignment operators. For each function type `F`, an object of type `FunctionWrapper<Range(Domain),F>` is constructed, which inherits from the abstract base class `FunctionWrapperBase<Range(Domain)>`.

Considering the implementation of `std::function` as described here, one may not expect any run-time gains for type erasure over virtual methods. While `std::function` itself does not have any virtual methods, each call to `operator()` does get routed through a virtual function. Additionally, each call to the copy constructor or assignment operator invokes a heap allocation. The virtual function call is the price for run-time polymorphism. It can only be avoided in some cases using smart compiler devirtualization.

To alleviate the cost of the heap allocation, `std::function` implements a technique called small object optimization. In addition to the pointer to `FunctionWrapper`, a `std::function` stores a small amount of raw memory. If the function is small enough to fit into this memory, it is stored there. Only in the case that more memory is needed, a heap allocation is performed. Small object optimization is therefore a trade-off between run-time and space requirements. `std::function` needs more memory with it, but is faster *for small objects*.

Small object optimization is not restricted to type erasure, and can in principle be used wherever heap allocations are involved. However, with an inheritance approach this nontrivial optimization would have to be exposed to the user code, while all its details are hidden from the user in a type erasure context.

While we rely on `std::function` as a type erasure class for global functions, this is not sufficient to represent extended function interfaces as discussed below. To this end `dune-functions` provides utility functionality to implement new type erasure classes with minimal effort, hiding, e.g., the details of small objects optimization. This can be used to implement type erasure for extended function interfaces that go beyond the ones provided by `dune-functions`. Since these utilities are not function-specific, they can also support the implementation of type-erased interfaces in other contexts. Similar functionality is, e.g., provided by the *poly* library (which is part of the *Adobe Source Libraries* [9]), and the *boost type erasure* library [10].

3 Extended function interfaces

The techniques discussed until now allow to model functions

$$f : \mathcal{D} \to \mathcal{R} \tag{1}$$

between a domain $\mathcal{D}$ and a range $\mathcal{R}$ by interfaces using either static or dynamic dispatch. In addition to this, numerical applications often require to model further properties of a function like differentiability or the fact that it is naturally defined locally on grid elements. In this section we describe how this is achieved in the `dune-functions` module using the techniques described above.

3.1 Differentiable functions

The extension of the concept for a function (1) to a differentiable function requires to also provide access to its derivative

$$Df : \mathcal{D} \to L(\mathcal{D}, \mathcal{R}) \tag{2}$$

where, in the simplest case, $L(\mathcal{D}, \mathcal{R})$ is the set of linear maps from the affine hull of $\mathcal{D}$ to $\mathcal{R}$.

Example 1. For a function $f : \mathbb{R}^n \to \mathbb{R}^m$ the derivative Df maps each vector $x \in \mathcal{D} = \mathbb{R}^n$ to a linear map $Df(x)$ from $\mathcal{D} = \mathbb{R}^n$ to $\mathcal{R} = \mathbb{R}^m$. Since we can identify any linear map $M : \mathbb{R}^n \to \mathbb{R}^m$ with a matrix $M \in \mathbb{R}^{m\times n}$ such that the value $M(y)$ of M at $y \in \mathbb{R}^n$ is given by the matrix vector product $M(y) = My \in \mathbb{R}^m$ we have identified $L(\mathcal{D}, \mathcal{R})$ with $\mathbb{R}^{m\times n}$. Hence Df is itself a function mapping vectors from $\mathbb{R}^n$ to matrices from $\mathbb{R}^{m\times n}$. Notice that $Df(x) \in \mathbb{R}^{m\times n}$ is commonly referred to as the *Jacobian matrix* of f at x.

To provide access to derivatives, the `dune-functions` module extends the ideas from the previous section in a natural way. A C++ construct is a differentiable function if, in addition to having `operator()` as described above, there is a free method `derivative` that returns a function that implements the derivative. The typical way to do this will be a friend function as illustrated in the class template `Polynomial`:

C++ code

```
template<class K>
class Polynomial
{
public:

  Polynomial() = default;
  Polynomial(const Polynomial& other) = default;
  Polynomial(Polynomial&& other) = default;

  Polynomial(std::initializer_list<double> coefficients) :
    coefficients_(coefficients) {}

  Polynomial(std::vector<K>&& coefficients) :
      coefficients_(std::move(coefficients)) {}

  Polynomial(const std::vector<K>& coefficients) :
    coefficients_(coefficients) {}

  const std::vector<K>& coefficients() const
  { return coefficients_; }

  K operator() (const K& x) const
  {
    auto y = K(0);
    for (size_t i=0; i<coefficients_.size(); ++i)
      y += coefficients_[i] * std::pow(x, i);
    return y;
```



```
  }

  friend Polynomial derivative(const Polynomial& p)
  {
    std::vector<K> dpCoefficients(p.coefficients().size()-1);
    for (size_t i=1; i<p.coefficients_.size(); ++i)
      dpCoefficients[i-1] = p.coefficients()[i]*i;
    return Polynomial(std::move(dpCoefficients));
  }

private:
  std::vector<K> coefficients_;
};
```

To use this class, write

C++ code

```
auto f = Polynomial<double>({1, 2, 3});
double a = f(3.14);
double b = derivative(f)(3.14);
```

Note, however, that the `derivative` method may be expensive, because it needs to compute the entire derivative function. It is therefore usually preferable to call it only once and to store the derivative function separately.

C++ code

```
auto df = derivative(f);
double b = df(3.14);
```

Functions supporting these operations are described by `Concept::DifferentiableFunction`, provided in the `dune-functions` module.

When combining differentiable functions and dynamic polymorphism, the `std::function` class cannot be used as is, because it does not provide access to the `derivative` method. However, it can serve as inspiration for more general type erasure wrappers. The `dune-functions` module provides the class

C++ code

```
template<class Signature,
         template<class> class DerivativeTraits=DefaultDerivativeTraits,
         size_t bufferSize=56>
class DifferentiableFunction;
```

in the file `dune/functions/common/differentiablefunction.hh`. Partially, it is a reimplementation of `std::function`. The first template argument of `DifferentiableFunction` is equivalent to the template argument `Range(Args...)` of `std::function`, and `DifferentiableFunction` implements a method

C++ code

```
Range operator() (const Domain& x) const;
```

This method works essentially like the one in `std::function`, despite the fact that its argument type is fixed to `const Domain&` instead of `Domain`, because arguments of a "mathematical function" are always immutable. Besides this, it also implements a free method

C++ code

```
friend DerivativeInterface derivative(const DifferentiableFunction& t);
```

that wraps the corresponding method of the function implementation. It allows to call the `derivative` method for objects whose precise type is determined only at run-time:

C++ code

```
DifferentiableFunction<double(double)> polymorphicF;
polymorphicF = Polynomial<double>({1, 2, 3});
double a = polymorphicF(3.14);
auto polymorphicDF = derivative(polymorphicF);
double b = polymorphicDF(3.14);
```

While the domain of a derivative is $\mathcal{D}$, the same as the one of the original function, its range is $L(\mathcal{D},\mathcal{R})$. Unfortunately, it is not feasible to always infer the best C++ type for objects from $L(\mathcal{D},\mathcal{R})$. To deal with this, `dune-functions` offers the `DerivativeTraits` mechanism that maps the signature of a function to the range type of its derivative. The line

C++ code

```
using DerivativeRange = DerivativeTraits<Range(Domain)>::Range;
```

shows how to access the type that should be used to represent elements of $L(\mathcal{D},\mathcal{R})$. The template `DefaultDerivativeTraits` is specialized for common combinations of Dune matrix and vector types, and provides reasonable defaults for the derivative ranges. However, it is also possible to change this by passing a custom `DerivativeTraits` template to the interface classes, e.g., to allow optimized application-specific matrix and vector types or use suitable representations for other or generalized derivative concepts.

Currently the design of `DifferentiableFunction` differs from `std::function` in that it only considers a single argument, but this can be vector valued.

3.2 GridView functions and local functions

A very important class of functions in any finite element application are discrete functions, i.e., functions that are defined piecewisely with respect to a given grid. Here, a grid covering a domain $\mathcal{D}$ is a decomposition into a set of subdomains in the sense that

$$\overline{\mathcal{D}} = \bigcup_{e \in \mathcal{G}} \bar{e}.$$

The subdomains $e \in \mathcal{G}$ are called *grid elements* or *cells* of the grid. In many applications with $\Omega \subset \mathcal{R}^d$ those elements are disjoint, open, nonempty d-dimensional polyhedra, possibly with curved boundaries. In Dune terminology, the k-dimensional faces of those polyhedra are called *grid entities* of codimension $d-k$. Hence the elements are the entities of codimension 0 while vertices, edges, and facets of those polyhedra are grid entities of codimension $k = d$, $k = d-1$, and $k = 1$, respectively. For each element $e \in \mathcal{G}$ we assume that there is a so-called *reference element* $\hat{e}$ and a bijective parametrization $\Phi_e : \hat{e} \to \bar{e}$. We call $\xi \in \hat{e}$ the local coordinate of $x \in \overline{\mathcal{D}}$ with respect to an element $e \in \mathcal{G}$ if $x \in \bar{e}$ and $\Phi_e(\xi) = x$.

For a given grid $\mathcal{G}$ on $\mathcal{D}$ a function $f : \mathcal{D} \to \mathcal{R}$ is called a discrete function if it has a natural piecewise definition with respect to the decomposition. Such functions are typically too expensive to evaluate in global coordinates, e.g., at points $x \in \mathcal{D}$ directly. Luckily this is hardly ever necessary. Instead one often knows an element e and local coordinates $\xi \in \bar{\hat{e}}$ of x with respect to e that allow to evaluate $f(x) = f(\Phi_e(\xi))$ cheaply. Formally, this means that we have localized versions

$$f_e = f \circ \Phi_e : \hat{e} \to \mathcal{R}, \qquad e \text{ is element of the grid.}$$

To support this kind of function evaluation, Dune has provided interfaces in the style of

C++ code

```
void evaluateLocal(Codim<0>::Entity element,
                   const LocalCoordinates& x,
                   Range& y);
```

Given an element `element` and a local coordinate `x`, such a method would evaluate the function at the given position, and return the result in the third argument `y`. This approach is currently used, e.g., in the grid function interfaces of the discretization modules `dune-pdelab` and `dune-fufem`.

There are several disadvantages to this approach. First, we have argued earlier that return-by-value is preferable to return-by-reference. Hence, an obvious improvement would be to use

C++ code

```
Range operator()(Codim<0>::Entity element, cost LocalCoordinates& x);
```

instead of the `evaluateLocal` method. However, there is a second disadvantage. In a typical access pattern in a finite element implementation, a function evaluation on a given element is likely to be followed by evaluations on the same element. For example, think of a quadrature loop that evaluates a coefficient function at all quadrature points of a given element. Function evaluation in local coordinates of an element can involve some setup code that depends on the element but not on the local coordinate `x`. This could be, e.g., pre-fetching of those coefficient vector entries that are needed to evaluate a finite element function on the given element, or retrieving the associated shape functions.

In the approaches described so far in this section, this setup code is executed again and again for each evaluation on the same element. To avoid this we propose the following usage pattern instead:

C++ code

```
auto localF = localFunction(f);
localF.bind(element);
auto y = localF(xLocal);            // evaluate f in element-local coordinates
```

Here we first obtain a *local function*, which represents the restriction of `f` to a single element. This function is then bound to a specific element using the method

C++ code

```
void bind(Codim<0>::Entity element);
```

This is the place for the function to perform any required setup procedures. Afterwards the local function can be evaluated using the interface described above, but now using local coordinates with respect to the element that the local function is bound to. The same localized function object can be used for other elements by calling `bind` with a different argument. Functions supporting these operations are called *grid view functions*, and described by `Concept::GridViewFunction` The local functions are described by `Concept::LocalFunction`. Both concepts are provided in the `dune-functions` module.

Since functions in a finite element context are usually at least piecewise differentiable, grid view functions as well as local functions provid the full interface of differentiable functions as outlined in Section 3.1. To completely grasp the semantics of the interface, observe that strictly speaking localization does not commute with taking the derivative. Formally, a localized version of the derivative is given by

$$(Df)_e : \hat{e} \to L(\mathcal{D}, \mathcal{R}), \qquad (Df)_e = (Df) \circ \Phi_e. \tag{3}$$

In contrast, the derivative of a localized function is given by

$$D(f_e) : \hat{e} \to L(\hat{e}, \mathcal{R}), \qquad D(f_e) = ((Df) \circ \Phi_e) \cdot D\Phi_e.$$

However, in the `dune-functions` implementation, the derivative of a local function does by convention always return values *in global coordinates*. Hence, the functions `dfe1` and `dfe2` obtained by

C++ code

```
auto df = derivative(f);
auto dfe1 = localFunction(df);
dfe1.bind(element);

auto fe = localFunction(f);
fe.bind(element);
auto dfe2 = derivative(fe);
```

both *behave the same*, implementing $(Df)_e$ as in (3). This is motivated by the fact that $D(f_e)$ is hardly ever used in applications, whereas $(Df)_e$ is needed frequently. To express this mild inconsistency in the interface, a local function uses a special `DerivativeTraits` implementation that forwards the derivative range to the one of the corresponding global function.

Again, type erasure classes allow to use grid view and local functions in a polymorphic way. The class

C++ code

```
template<class Signature,
         class GridView,
         template<class> class DerivativeTraits=DefaultDerivativeTraits,
         size_t bufferSize=56>
class GridViewFunction;
```

stores any function that models the `GridViewFunction` concept with given signature and grid view type. Similarly, functions modeling the `LocalFunction` concept can be stored in the class

C++ code

```
template<class Signature,
         class Element,
         template<class> class DerivativeTraits=DefaultDerivativeTraits,
         size_t bufferSize=56>
class LocalFunction;
```

These type erasure classes can be used in combination:

C++ code

```
GridViewFunction<double(GlobalCoordinate), GridView> polymorphicF;
polymorphicF = f;
auto polymorphicLocalF = localFunction(polymorphicF);
polymorphicLocalF.bind(element);
LocalCoordinate xLocal = ... ;
auto y = polymorphicLocalF(xLocal);
```

Notice that, as described above, the `DerivativeTraits` used in `polymorphicLocalF` are not the same as the ones used by `polymorphicF`. Instead, they are a special implementation forwarding to the global derivative range even for the domain type `LocalCoordinate`.

4 Performance measurements

In this last chapter we investigate how the interface design for functions in Dune influences the run-time efficiency. Two particular design choices are expected to be critical regarding execution speed: (i) returning the results of function evaluations by value involves temporary objects and copying unless the compiler is smart enough to remove those using return-value-optimization. In the old interface, such copying could not occur by construction, (ii) using type erasure instead of virtual functions for dynamic polymorphism. While there are fewer reasons to believe that this may cause changes in execution time, it is still worthwhile to check empirically.

As a benchmark we have implemented a small C++ program that computes the integral

$$I(f) := \int_0^1 f(x)\,dx$$

for different integrands, using a standard composite mid-point rule. We chose this problem because it is very simple, but still an actual numerical algorithm. More importantly, most of the time is spent evaluating the integrand function. Finally, hardly any main memory is needed, and hence memory bandwidth limitations will not influence the measurements. We have deliberately omitted tests for derivatives and piecewise functions. As these use return-by-value and type erasure in much the same way as function evaluation does, we do not expect much additional information from such additional tests.

The example code is a pure C++11 implementation with no reference to DUNE. The relevant interfaces from DUNE are so short that it was considered preferable to copy them into the benchmark code to allow easier building. The code is available in a single file attached to this pdf document, via the icon in the margin.

To check the influence of return types with different size we used integrands of the form

$$f : \mathbb{R} \to \mathbb{R}^N, \qquad f(x)_i = x + i - 1, \qquad i = 1, \dots, N,$$

for various sizes N. This special choice was made to keep the computational work done inside of the function to a minimum while avoiding compiler optimizations that replace the function call by a compile-time expression. The test was performed with $n = \lfloor 10^8/N \rfloor$ subintervals for the composite mid-point rule leading to n function evaluations, such that the timings are directly comparable for different values of N.

For the test we implemented four variants of function evaluation:

(a) Return-by-value with static dispatch using plain `operator()`,

(b) Return via reference with static dispatch using `evaluate()`,

(c) Return-by-value with dynamic dispatch using `std::function::operator()`,

(d) Return via reference with dynamic dispatch using `VirtualFunction::evaluate()`, as in the introduction.

The test was performed with $N = 1, \dots, 16$ components for the function range, using `double` to implement the components. We used GCC-4.9.2 and Clang-3.6 as compilers, as provided by the Linux distribution Ubuntu 15.04. To avoid cache effects and to eliminate outliers we did a warm-up run before each measured test run and selected the minimum of four subsequent runs for all presented values.

Figure 2.A shows the execution time in milliseconds over N when compiling with GCC-4.9 and the compiler options `-std=c++11 -O3 -funroll-loops`. One can observe that the execution time is the same for variants (a) and (b) and all values of N. We conclude that for static dispatch there is no run-time overhead when using return-by-value, or, more precisely, that the compiler is able to optimize away any overhead. Comparing the dynamic dispatch variants (c) and (d) we see that for small values of N there is an overhead for return-by-value with type erasure compared to the classic approach using inheritance and virtual functions. This is somewhat surprising since pure return-by-value does not impose an overhead, and dynamic dispatch happens for both variants.

Guessing that the compiler is not able to optimize the nested function calls in the type erasure interface class `std::function` to full extent, we repeated the tests using *profile guided optimization*.

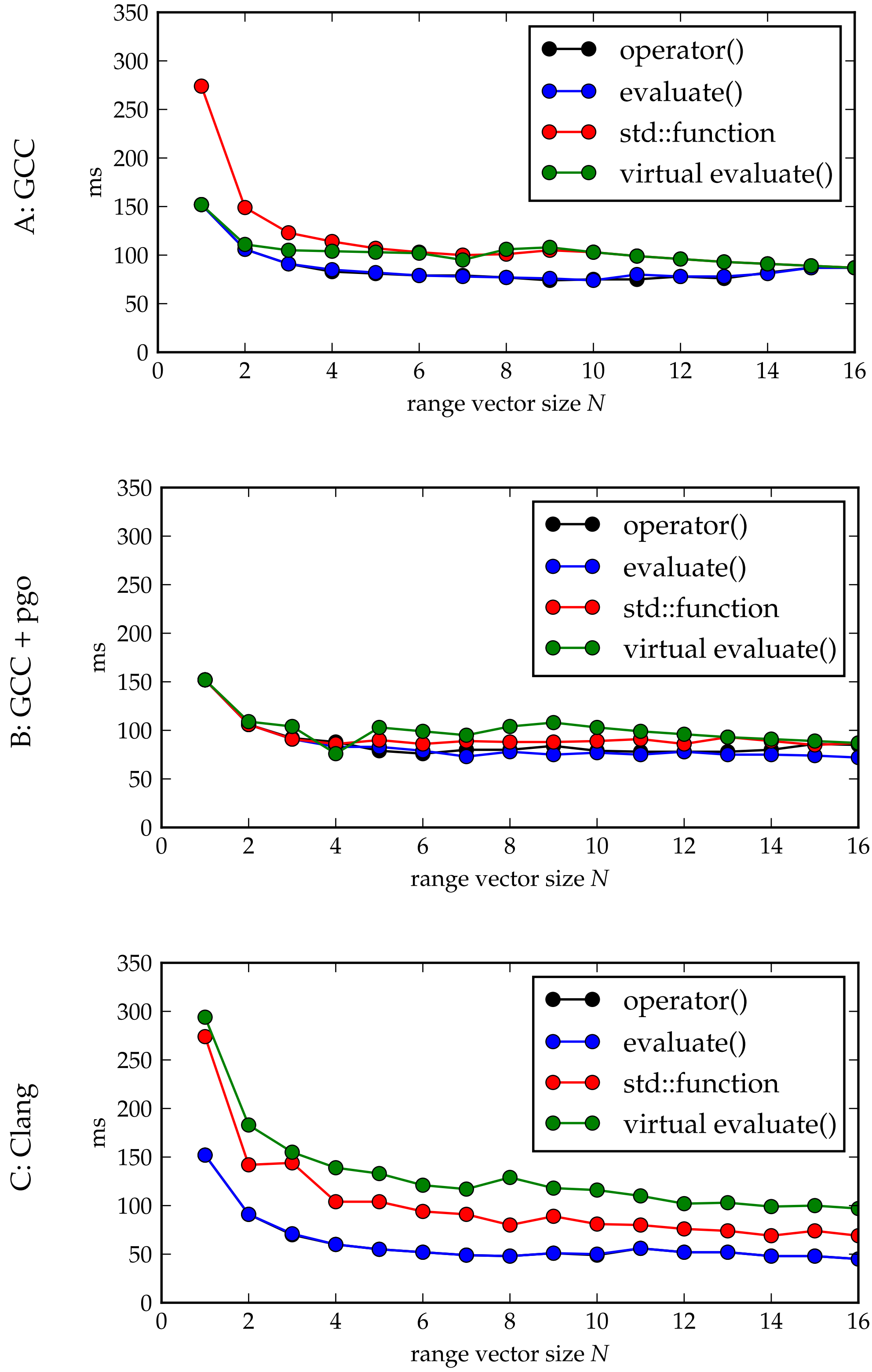

Figure 2: Timings for $\lfloor 10^8/N \rfloor$ function calls over varying vector size N using (A) GCC, (B) GCC with profile-guided optimization, and (C) Clang.

To this end the code was first compiled using the additional option `-fprofile-generate`. When running the obtained program once, it generates statistics on method calls that are used by subsequent compilations with the additional option `-fprofile-use` to guide the optimizer. The results depicted in Figure 2.B show that the compiler is now able to generate code that performs equally well for variant (c) and (d). In fact variant (c) is sometime even slightly faster.

Finally, Figure 2.C shows results for Clang-3.6 and the compiler options `-std=c++11 -O3 -funroll-loops`. Again variants (a) and (b) show identical results. In contrast, variant (c) using `std::function` is now clearly superior compared to variant (d). Note that we only used general-purpose optimization options and that this result did not require fine-tuning with more specialized compiler flags.

5 Conclusion

We have presented a new interface for functions in Dune, which is implemented in the new `dune-functions` module. The interface follows the ideas of function objects and `std::function` from the C++ standard library, and generalises these concepts to allow for differentiable functions and discrete grid functions. For run-time polymorphism we offer corresponding type erasure classes similar to `std::function`. The performance of these new interfaces was compared to existing interfaces in Dune. When using the optimization features of modern compilers, the proposed new interfaces are at least as efficient as the old ones, while being much easier to read and use.

References

[1] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for adaptive and parallel scientific computing. Part II: Implementation and tests in DUNE. *Computing*, 82(2–3):121–138, 2008.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for adaptive and parallel scientific computing. Part I: Abstract framework. *Computing*, 82(2–3):103–119, 2008.

[3] K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, pages 306–323. ACM, 1996.

[4] C. Engwer, C. Gräser, S. Müthing, and O. Sander. Dune-functions module. `http://www.dune-project.org/modules/dune-functions`.

[5] J. Hubička. Devirtualization in C++. online blog, `http://hubicka.blogspot.de/2014/01/devirtualization-in-c-part-1.html`, 2014. (at least) seven parts, last checked on Dec. 8. 2015.

[6] International Organization for Standardization. ISO/IEC 14882:2011 Programming Language C++, 9 2011.

[7] E. Niebler. Range-v3 library. `https://github.com/ericniebler/range-v3`.

[8] E. Niebler. Concept checking in C++11. online blog, `http://ericniebler.com/2013/11/23/concept-checking-in-c11`, 2013. last checked on Dec. 8. 2015.

[9] S. Parent, M. Marcus, and F. Brereton. Adobe source libraries. `http://stlab.adobe.com/`.

[10] S. Watanabe. Boost type erasure library. `http://www.boost.org/doc/libs/release/libs/type_erasure/`.