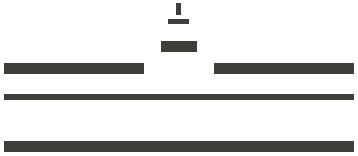

Westfälische
Wilhelms-Universität
Münster

PDELab cleanup in the light of dune-functions

wissen.leben
WWU Münster

Christian Engwer

28.09.2015,
DUNE User-Meeting

Lesson learned...

- Value semantics

Lesson learned...

- Value semantics
- Value semantics

Lesson learned...

- Value semantics
- Value semantics
- Value semantics

Lesson learned...

- Value semantics
- Value semantics
- Value semantics
- Type erasure

Lesson learned...

- Value semantics
- Value semantics
- Value semantics
- Type erasure
- Functional programming

Lesson learned...

- Value semantics → better readability
- Value semantics
- Value semantics
- Type erasure
- Functional programming

Lesson learned...

- Value semantics → better readability
- Value semantics → better maintainability
- Value semantics
- Type erasure
- Functional programming

Lesson learned...

- Value semantics → better readability
- Value semantics → better maintainability
- Value semantics → suprise ;-) better optimization
- Type erasure
- Functional programming

Lesson learned...

- Value semantics → better readability
- Value semantics → better maintainability
- Value semantics → suprise ;-) better optimization
- Type erasure → less boiler plate code
- Functional programming

anybody heard of PDELab-boilerplate?

Lesson learned...

- Value semantics → better readability
- Value semantics → better maintainability
- Value semantics → suprise ;-) better optimization
- Type erasure → less boiler plate code
- Functional programming → don't think about typedefs

Lesson learned...

- Value semantics → better readability
- Value semantics → better maintainability
- Value semantics → suprise ;-) better optimization
- Type erasure → less boiler plate code
- Functional programming → don't think about typedefs
- (Use bind where ever possible)

Plans for PDELab

- Play nicely with dune-functions functions
- Add extended interface for pdelab, including setTime:
- use callables to implement Local-Operator parameter classes
- adopt dune-functions GridFunctionSpaceBasis concepts

Plans for PDELab

- Play nicely with dune-functions functions
  - interpolate from any callable
  - create a discrete function, which fulfills the dune-functions interface:
    - create leaf functions DiscreteGridViewFunction
    - create trees of discrete functions
- Add extended interface for pdelab, including setTime:
  - type erasure class to the extended pdelab interface, including setTime
  - free standing setTime function
- use callables to implement Local-Operator parameter classes
- adopt dune-functions GridFunctionSpaceBasis concepts
  - ...

Plans for PDELab

- Play nicely with dune-functions functions
  - interpolate from any callable
  - create a discrete function, which fulfills the dune-functions interface:
    - create leaf functions DiscreteGridViewFunction
    - create trees of discrete functions
- Add extended interface for pdelab, including setTime:
  - type erasure class to the extended pdelab interface, including setTime
  - free standing setTime function
- use callables to implement Local-Operator parameter classes
- adopt dune-functions GridFunctionSpaceBasis concepts
  - ...

... will require C++-11 complient compilers

Interpolate

Interpolate from any callable

- Fallback support for old-style DiscreteGridFunction implementations
- Directly support lambdas and any other callable as input to `interpolate`
- Allow to map from `FieldVector<RF,N>` to `PowerGridFunctionSpace<ScalarSpace<GV,...>,10>`

Interpolate

Interpolate from any callable

- Fallback support for old-style DiscreteGridFunction implementations
- Directly support lambdas and any other callable as input to `interpolate`
- Allow to map from `FieldVector<RF,N>` to `PowerGridFunctionSpace<ScalarSpace<GV,...>,10>`

```
// interpolate from old-style scalar function
{
    ConstGridFunction<GV,double> f(gv,1);
    Dune::PDELab::interpolate(f,gfs,x);
}
```

Interpolate

Interpolate from any callable

- Fallback support for old-style DiscreteGridFunction implementations
- Directly support lambdas and any other callable as input to `interpolate`
- Allow to map from `FieldVector<RF,N>` to `PowerGridFunctionSpace<ScalarSpace<GV,...>,10>`

```
// interpolate from lambda
{
    typedef Dune::FieldVector<typename GV::ctype, GV::dimension> Domain;
    auto f = [](const Domain& x) { return 1.0; };
    auto lf = Dune::Functions::makeGridViewFunction(f, gv);
    Dune::PDELab::interpolate(lf,gfs,x);
    Dune::PDELab::interpolate(f,gfs,x);
}
```

Interpolate

Interpolate from any callable

- Fallback support for old-style DiscreteGridFunction implementations
- Directly support lambdas and any other callable as input to `interpolate`
- Allow to map from `FieldVector<RF,N>` to `PowerGridFunctionSpace<ScalarSpace<GV,...>,10>`

```
double init_fnkt(const Dune::FieldVector<double,2> & x)
{ return 1.0; }
  ...

// interpolate from global function
{
    auto f = init_fnkt;
    auto lf = Dune::Functions::makeGridViewFunction(f, gv);
    Dune::PDELab::interpolate(lf,gfs,x);
    Dune::PDELab::interpolate(x_component_A,gfs,x);
    Dune::PDELab::interpolate(f,gfs,x);
}
```

Dune::PDELab::DiscretGridViewFunction

- differentiable function on a `GridView`
- Modelled according to dune-functions concepts
- PDELab GridFunctionSpace + Coefficients vector

```
enum { dim = GV::dimension };
using namespace Dune::PDELab;
// make a grid function space
using QkFEM = QkLocalFiniteElementMap<GV,float,double,k>;
using QkGFS = GridFunctionSpace<GV,QkFEM>;
QkFEM qkfem(gv);
QkGFS qkgfs(gv,qkfem);
// create a coefficient vector
using Vector = typename BackendVectorSelector<QkGFS,double>::Type;
Vector x(qkgfs);
// ... and initialize the data ...
```

Dune::PDELab::DiscretGridViewFunction

- differentiable function on a `GridView`
- Modelled according to dune-functions concepts
- PDELab GridFunctionSpace + Coefficients vector

```
// make global function
using DiscreteFunction = DiscreteGridViewFunction<QkGFS,Vector>;
DiscreteFunction dgvf(qkgfs,x);
// make local functions
auto localf = localFunction(dgvf);
// do something
for (const auto & e : elements(gv))
{
    localf.bind(e);
    value = localf(e.geometry().center());
    if (localf.maxDiffOrder >= 1) // statically evaluated
        auto jacobian = derivative(localf)(e.geometry().center());
    localf.unbind();
}
```

Status-Update

- ✓ Play nicely with dune-functions functions
  - ✓ interpolate from any callable
  - ✓ create a discrete function, which fulfills the dune-functions interface:
    - ✓ create leaf functions DiscreteGridViewFunction
    - ✓ create trees of discrete functions
- ▸ Add extended interface for pdelab, including setTime:
  - ▸ type erasure class to the extended pdelab interface, including setTime
  - ▸ free standing setTime function
- ▸ use callables to implement Local-Operator parameter classes
- ▸ adopt dune-functions GridFunctionSpaceBasis concepts

Callables for parameter classes

Status quo: write your own parameter class

```
template<typename GV, typename RF>
class PoissonModelProblem
{
  typedef Dune::PDELab::ConvectionDiffusionBoundaryConditions::Type BCType;

public:
  typedef Dune::PDELab::ConvectionDiffusionParameterTraits<GV,RF> Traits;

  //! tensor diffusion coefficient
  typename Traits::PermTensorType
  A (const typename Traits::ElementType& e, const typename Traits::DomainType
  {
    typename Traits::PermTensorType I;
    for (std::size_t i=0; i<Traits::dimDomain; i++)
      for (std::size_t j=0; j<Traits::dimDomain; j++)
        I[i][j] = (i==j) ? 1 : 0;
    return I;
  }

  //! velocity field
  typename Traits::RangeType
  b (const typename Traits::ElementType& e, const typename Traits::DomainType
  {
    typename Traits::RangeType v(0.0);
```

Callables for parameter classes

Status quo: write your own parameter class

```
template<typename GV, typename RF>
class PoissonModelProblem
{
  typedef Dune::PDELab::ConvectionDiffusionBoundaryConditions::Type BCType;

public:
  typedef Dune::PDELab::ConvectionDiffusionParameterTraits<GV,RF> Traits;

  //! tensor diffusion coefficient
  typename Traits::PermTensorType
  A (const typename Traits::ElementType& e, const typename Traits::DomainType& x) const
  {
    typename Traits::PermTensorType I;
    for (std::size_t i=0; i<Traits::dimDomain; i++)
      for (std::size_t j=0; j<Traits::dimDomain; j++)
        I[i][j] = (i==j) ? 1 : 0;
    return I;
  }

  //! velocity field
  typename Traits::RangeType
  b (const typename Traits::ElementType& e, const typename Traits::DomainType& x) const
  {
    typename Traits::RangeType v(0.0);
    return v;
  }

  //! sink term
  typename Traits::RangeFieldType
  c (const typename Traits::ElementType& e, const typename Traits::DomainType& x) const
  {
    return 0.0;
  }

  //! source term
  typename Traits::RangeFieldType
  f (const typename Traits::ElementType& e, const typename Traits::DomainType& x) const
  {
    const auto& xglobal = e.geometry().global(x);
```

Callables for parameter classes

Status quo: write your own parameter class

```
template<typename GV, typename RF>
class PoissonModelProblem
{
  typedef Dune::PDELab::ConvectionDiffusionBoundaryConditions::Type BCType;

public:
  typedef Dune::PDELab::ConvectionDiffusionParameterTraits<GV,RF> Traits;

  //! tensor diffusion coefficient
  typename Traits::PermTensorType
  A (const typename Traits::ElementType& e, const typename Traits::DomainType& x) const
  {
    typename Traits::PermTensorType I;
    for (std::size_t i=0; i<Traits::dimDomain; i++)
      for (std::size_t j=0; j<Traits::dimDomain; j++)
        I[i][j] = (i==j) ? 1 : 0;
    return I;
  }

  //! velocity field
  typename Traits::RangeType
  b (const typename Traits::ElementType& e, const typename Traits::DomainType& x) const
  {
    typename Traits::RangeType v(0.0);
    return v;
  }

  //! sink term
  typename Traits::RangeFieldType
  c (const typename Traits::ElementType& e, const typename Traits::DomainType& x) const
  {
    return 0.0;
  }

  //! source term
  typename Traits::RangeFieldType
  f (const typename Traits::ElementType& e, const typename Traits::DomainType& x) const
  {
    const auto& xglobal = e.geometry().global(x);
    if (xglobal[0]>0.25 && xglobal[0]<0.375 && xglobal[1]>0.25 && xglobal[1]<0.375)
      return 50.0;
    else
      return 0.0;
  }

  //! boundary condition type function
  BCType
  bctype (const typename Traits::IntersectionType& is, const typename Traits::IntersectionDomainType& x) const
  {
    typename Traits::DomainType xglobal = is.geometry().global(x);

    if (xglobal[1]<1E-6 || xglobal[1]>1.0-1E-6)
      {
        return Dune::PDELab::ConvectionDiffusionBoundaryConditions::Neumann;
      }
    if (xglobal[0]>1.0-1E-6 && xglobal[1]>0.5+1E-6)
      {
        return Dune::PDELab::ConvectionDiffusionBoundaryConditions::Neumann;
      }
    return Dune::PDELab::ConvectionDiffusionBoundaryConditions::Dirichlet;
  }

  //! Dirichlet boundary condition value
  typename Traits::RangeFieldType
  g (const typename Traits::ElementType& e, const typename Traits::DomainType& x) const
  {
    typename Traits::DomainType xglobal = e.geometry().global(x);
    xglobal -= 0.5;
    return exp(-xglobal.two_norm2());
  }

  //! Neumann boundary condition
  typename Traits::RangeFieldType
  j (const typename Traits::IntersectionType& is, const typename Traits::IntersectionDomainType& x) const
  {
    typename Traits::DomainType xglobal = is.geometry().global(x);

    if (xglobal[0] > 1.0 - 1E-6 && xglobal[1] > 0.5 + 1E-6) {
      return -5.0;
    } else {
      return 0.0;
    }
  }

  //! outflow boundary condition
  typename Traits::RangeFieldType
  o (const typename Traits::IntersectionType& is, const typename Traits::IntersectionDomainType& x) const
  {
    return 0.0;
  }
};
```

Callables for parameter classes

- allow to use lambas and/or discrete functions as parameterization
- allow to extend/overwrite existing parameter classes
- (add a bind method)

```
// Define parameter functions f,g,j and \partial\Omega_D/N
auto f = Dune::Functions::makeGridViewFunction (
    [] (const Domain& x) {
        return (x[0]>0.25 && x[0]<0.375) ? 50.0 : 0.0;
    }, gv);
auto g = Dune::Functions::makeGridViewFunction (
    [] (const Domain& x) { return std::exp(-x.two_norm2()); }, gv);

using Dune::PDELab::ConvectionDiffusionParameters;
using Dune::PDELab;
auto problem =
    Parameters::merge(
        defineSourceTerm(localFunction(f)),
        defineDirichletBoundaryValue(localFunction(g)),
        // ... + other parameters
      );
```

Callables for parameter classes

- allow to use lambas and/or discrete functions as parameterization
- allow to extend/overwrite existing parameter classes
- (add a bind method)

```
// Define parameter functions f,g,j and \partial\Omega_D/N
auto f = Dune::Functions::makeGridViewFunction (
    [] (const Domain& x) {
        return (x[0]>0.25 && x[0]<0.375) ? 50.0 : 0.0;
    }, gv);
auto g = Dune::Functions::makeGridViewFunction (
    [] (const Domain& x) { return std::exp(-x.two_norm2()); }, gv);

using Dune::PDELab::ConvectionDiffusionParameters;
using Dune::PDELab;
auto problem =
    Parameters::overwrite(
        Dune::PDELab::ConvectionDiffusionModelProblem<GV,R>(), // <==
        defineSourceTerm(localFunction(f)),
        defineDirichletBoundaryValue(localFunction(g))
      );
```

Callables for parameter classes

Defining possible Parameters for a model:

```
namespace ConvectionDiffusionParameters {
    DefinePDELabParameterName(DiffusionTensor, A);
    DefinePDELabParameterName(VelocityField, b);
    DefinePDELabParameterName(SinkTerm, c);
    DefinePDELabParameterName(SourceTerm, f);
    DefinePDELabParameterName(BoundaryCondition, bctype);
    DefinePDELabParameterName(DirichletBoundaryValue, g);
    DefinePDELabParameterName(NeumannBoundaryValue, j);
    DefinePDELabParameterName(OutflowBoundaryValue,o);
} // end namespace ConvectionDiffusionParameters
```

- `DefinePDELabParameterName(Name,VarName)` unfolds to a class storing a single callable `VarName`
- forwards all `bind(element)` calls to those parameters which have their own bind
- `Dune::PDELab::Parameters::merge` merges all parameters using inheritence

Status-Update II

- ✓ Play nicely with dune-functions functions
  - ✓ interpolate from any callable
  - ✓ create a discrete function, which fulfills the dune-functions interface:
    - ✓ create leaf functions DiscreteGridViewFunction
    - ✓ create trees of discrete functions
- ✗ Add extended interface for pdelab, including setTime:
  - ► type erasure class to the extended pdelab interface, including setTime
  - ► free standing setTime function
- ✓ use callables to implement Local-Operator parameter classes
- ► adopt dune-functions GridFunctionSpaceBasis concepts

GridFunctionSpaceBasis

- clean definition of internal interfaces
- *finally* we can implement special versions to benefit from simple structure
- separation between the actual basis and the parameterization

→ avoid loads of internal hassle

GridFunctionSpaceBasis

- clean definition of internal interfaces
- *finally* we can implement special versions to benefit from simple structure
- separation between the actual basis and the parameterization

→ avoid loads of internal hassle

Only ideas up to now!

GridFunctionSpaceBasis

- clean definition of internal interfaces
- *finally* we can implement special versions to benefit from simple structure
- separation between the actual basis and the parameterization

→ avoid loads of internal hassle

```
using namespace Dune;
auto basis_description = PDELab::q2_basis(gv, backend(istl_fieldvector));
auto my_basis = basis_description.createBasis();
```

GridFunctionSpaceBasis

- clean definition of internal interfaces
- *finally* we can implement special versions to benefit from simple structure
- separation between the actual basis and the parameterization

→ avoid loads of internal hassle

```
using namespace Dune;
auto my_basis =
    PDELab::q2_basis(gv, backend(istl)).createBasis();
```

Example Taylor-Hood

```
auto th_desc = composite_basis(
    vector_basis(
        q2_basis(gv),
    ),
    q1_basis(gv),
    backend(istl)
);
```

Flat Vector : $(v_{x,1}, v_{x,2}, v_{x,3}, \ldots, v_{y,1}, \ldots, p_0, \ldots)$

Example Taylor-Hood

```
auto th_desc = composite_basis(
    vector_basis(
        q2_basis(gv),
        ordering(interleaved),
    ),
    q1_basis(gv),
    backend(istl)
);
```

Flat Vector, interleaved v : $(v_{x,1}, v_{y,1}, v_{x,2}, v_{y,2}, \ldots, p_0, \ldots)$

Example Taylor-Hood

```
auto th_desc = composite_basis(
    vector_basis(
        q2_basis(gv),
        ordering(interleaved),
        blocking(none),
    ),
    q1_basis(gv),
    merging(lexicographic),
    blocking(lexicographic),
    backend(istl)
);
```

Block-Vector, locally interleaved $v : ([v_{x,1}, v_{y,1}, v_{x,2}, v_{y,2}, \ldots], [p_0, \ldots])$

Example Taylor-Hood

```
auto th_basis = composite_basis(
    vector_basis(
        q2_basis(gv),
        ordering(interleaved),
        blocking(static),
    ),
    q1_basis(gv),
    backend(istl)
);
```

Locally Block-Vector, interleaved $v : ([v_{x,1}, v_{y,1}], [v_{x,2}, v_{y,2}], \ldots, p_0, \ldots)$

(see Oli's `MultiBlockTypeMatrix` example)

Status-Update III

- ✓ Play nicely with dune-functions functions
  - ✓ interpolate from any callable
  - ✓ create a discrete function, which fulfills the dune-functions interface:
    - ✓ create leaf functions DiscreteGridViewFunction
    - ✓ create trees of discrete functions
- ✗ Add extended interface for pdelab, including setTime:
  - ▸ type erasure class to the extended pdelab interface, including setTime
  - ▸ free standing setTime function
- ✓ use callables to implement Local-Operator parameter classes
- ✗ adopt dune-functions GridFunctionSpaceBasis concepts

  → ???

Lookout

When will this land in PDELab?

- after the PDELab 2.4 release
- definitely before 2016

Can I use this now already?

- there is a git branch `feature/new-parameters-from-callable`
- implementation might still change significantly

Future plans?

- Cleanup the code
- GridFunctionSpaceBasis design and implementation
- get rid of more explicit template parameters