Chroma

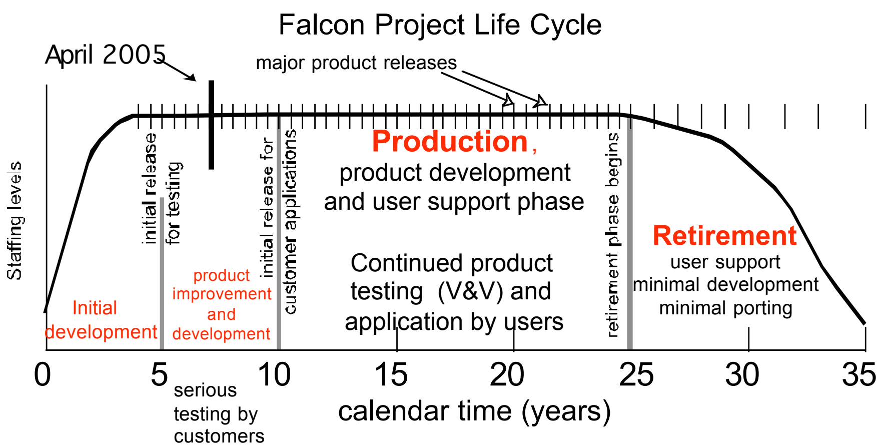
Bálint Joó,
Scientific Computing Group
Jefferson Lab

Chroma

- QDP++ covers ‘nuts and bolts’ for us
  - provides lattice types/expressions
  - hides parallelism
- Chroma provides the infrastructure for constructing LQCD applications
  - Components: Gauge And Fermion Actions, Solvers, MD etc.
  - Higher Level:
    - two full applications: chroma and hmc
    - lots of measurement tasks with XML interface
- NB: This is chroma the LQCD code
  - Not: Chroma(tm) the Lustre Filesystem Management software from Whamcloud.

Chroma will be 10 this year

- first check in is dated Dec 16, 2002
- Chroma is entering ‘middle-age’
- Structurally pretty stable
  - mostly tweaks (new solvers etc)
  - QUDA solver integration



Expected life cycle of the Falcon code
D.E. Post, J.o.P Conf. Series, 125 (2008) 012090
(SciDAC’08 Seattle)

- Another 10 years and Exascale?

Some Design Aims

- Try to capture mathematical structure, through class structure
  - Inheritance, virtual functions
- Use extensible techniques (Patterns)
  - Avoid monster switch statements
  - Use map/factory based creation
- Would like it to be easy to drive from external file
  - Little ‘measurement’ interpreter (Command Pattern)

Capturing Mathematical Structure

- Demonstrate with Even Odd Preconditioning:

Linear Operator: $y = Mx$

```
LinearOperator<T> :

virtual
void operator(T& y,
              const T& x,
              enum PlusMinus isign);
virtual const Subset& subset();
```

'Differentiable' Linear Operator: $y = Mx$, $F = X^\dagger \dot{M} Y$

```
DiffLinearOperator<T,P,Q> :
 virtual void operator(T& y,
              const T& x,
              enum PlusMinus isign);
 virtual const Subset& subset();
 virtual void deriv(P& F,
            const T& X,
            const T& Y,
            enum PlusMinus isign);
```

Schur Even Odd Preconditioned Linear Operator

$$\begin{bmatrix} M_{ee} & 0 \\ 0 & S \end{bmatrix}$$

$$S = M_{oo} - M_{oe} M_{ee}^{-1} M_{eo}$$

```
EvenOddPrecLinearOperator<T,P,Q> :
 virtual void evenOddLinOp(T& y, const T& x, enum PlusMinus isign);

 virtual void oddEvenLinOp(T& y, const T& x, enum PlusMinus isign);

 virtual void oddOddLinOp(T& y, const T& x,  enum PlusMinus isign);

 virtual void evenEvenLinOp(T& y, const T& x, enum PlusMinus isign);

 virtual evenEvenInvLinOp(T& y, const T& x, enum PlusMinus isign);

 virtual void operator()(T& y, const T& x, enum PlusMinus isign) {
   T tmp; oddEvenLinOp(tmp, x, isign);
   T tmp2; evenEvenInvLinOp(tmp2, tmp, isign);
   evenOddLinOp(tmp, tmp2, isign);
   oddOddLinOp(y, x, isign);
   y -= tmp;
 }
```

Default Implementation

Capturing mathematical structure

- Of course force term can also be done like this:
  - ie: derivEvenEvenLinOp()
  - derivOddEvenLinOp(), etc...
  - then code the full deriv() in terms of these
- Structure also applies to things like quark prop calculation

Solve on 1 checkerboard, with modified source

$$Mx = y \quad \Rightarrow \quad \begin{aligned} S\ x_o &= y_o - M_{oe} M_{ee}^{-1} y_e \\ x_e &= M_{ee}^{-1} \left(y_e - M_{eo} x_o \right) \end{aligned}$$

Reconstruct on other checkerboard.

- And HMC:

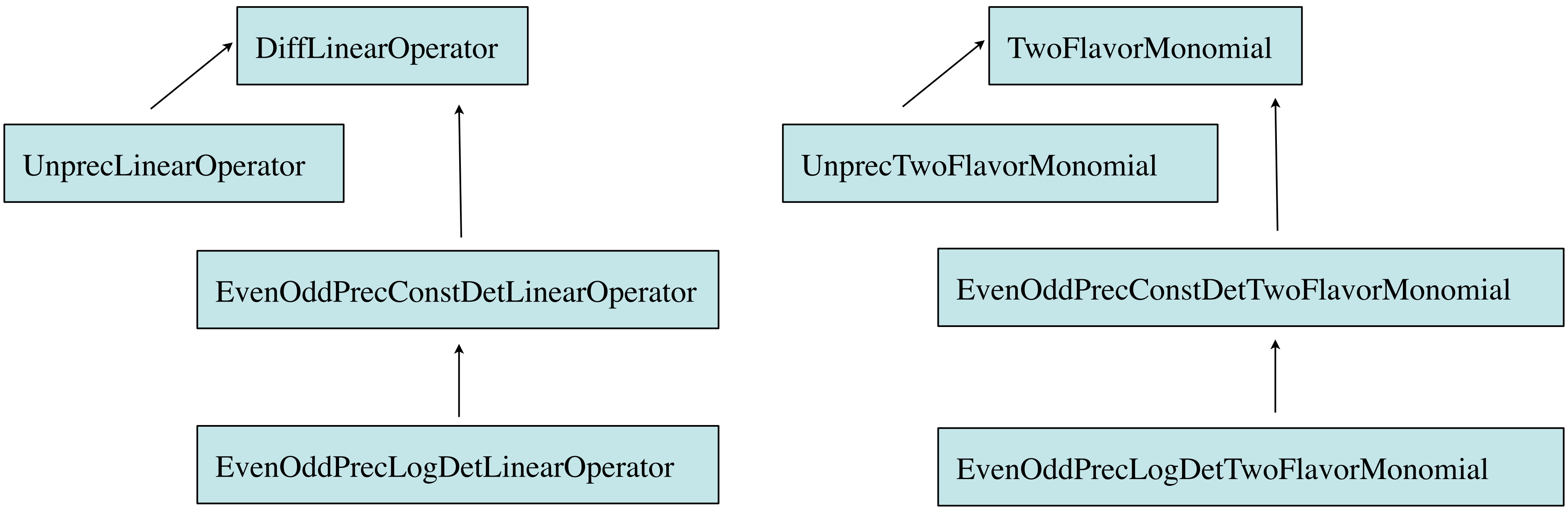
$$S = 2\ \mathrm{Tr}\ \mathrm{Ln}\ M_{ee} - \psi_o^\dagger \left(S^\dagger S \right)^{-1} \psi_o$$

Parallel Inheritance Trees

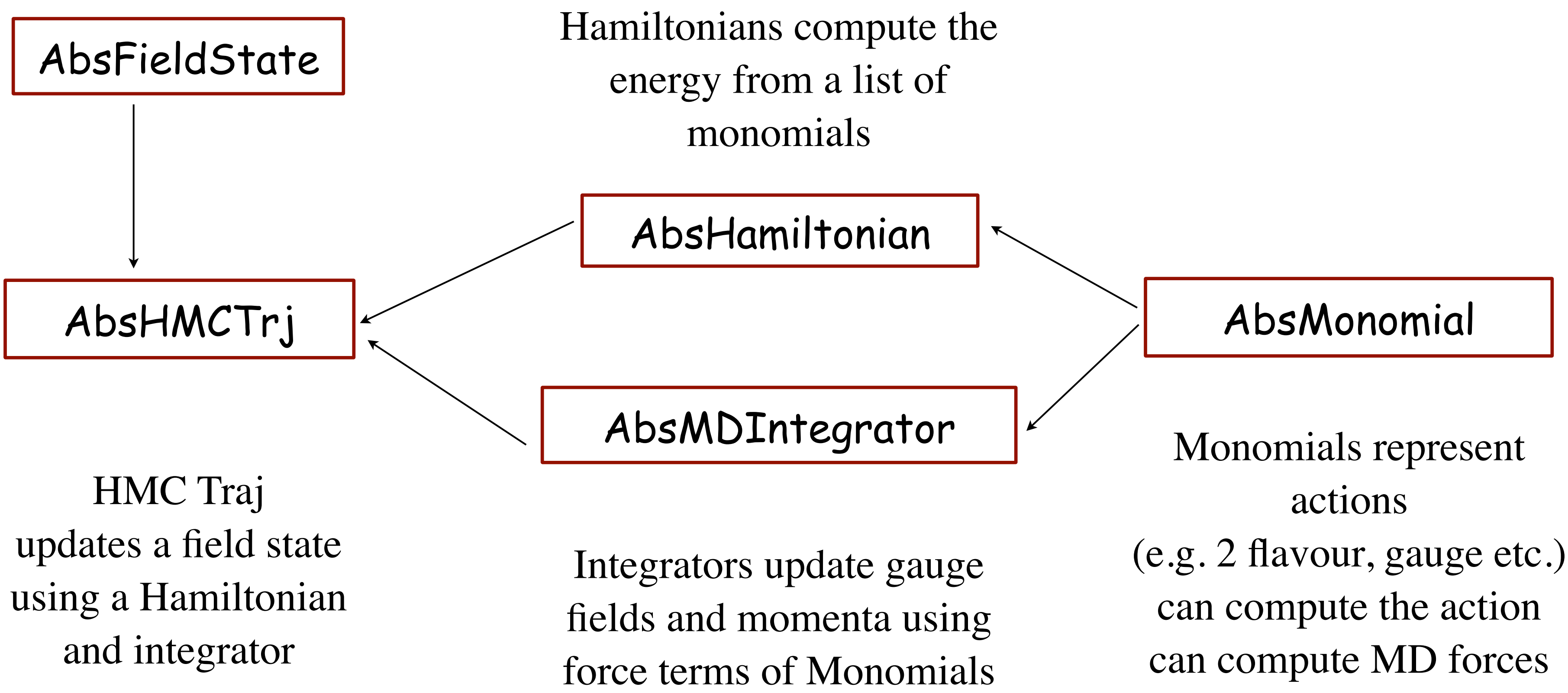
- Capture ‘sameness of structure’ amongst different components (Linear Operators, QProp solvers, Monomials etc)

Linear Operators

Monomials



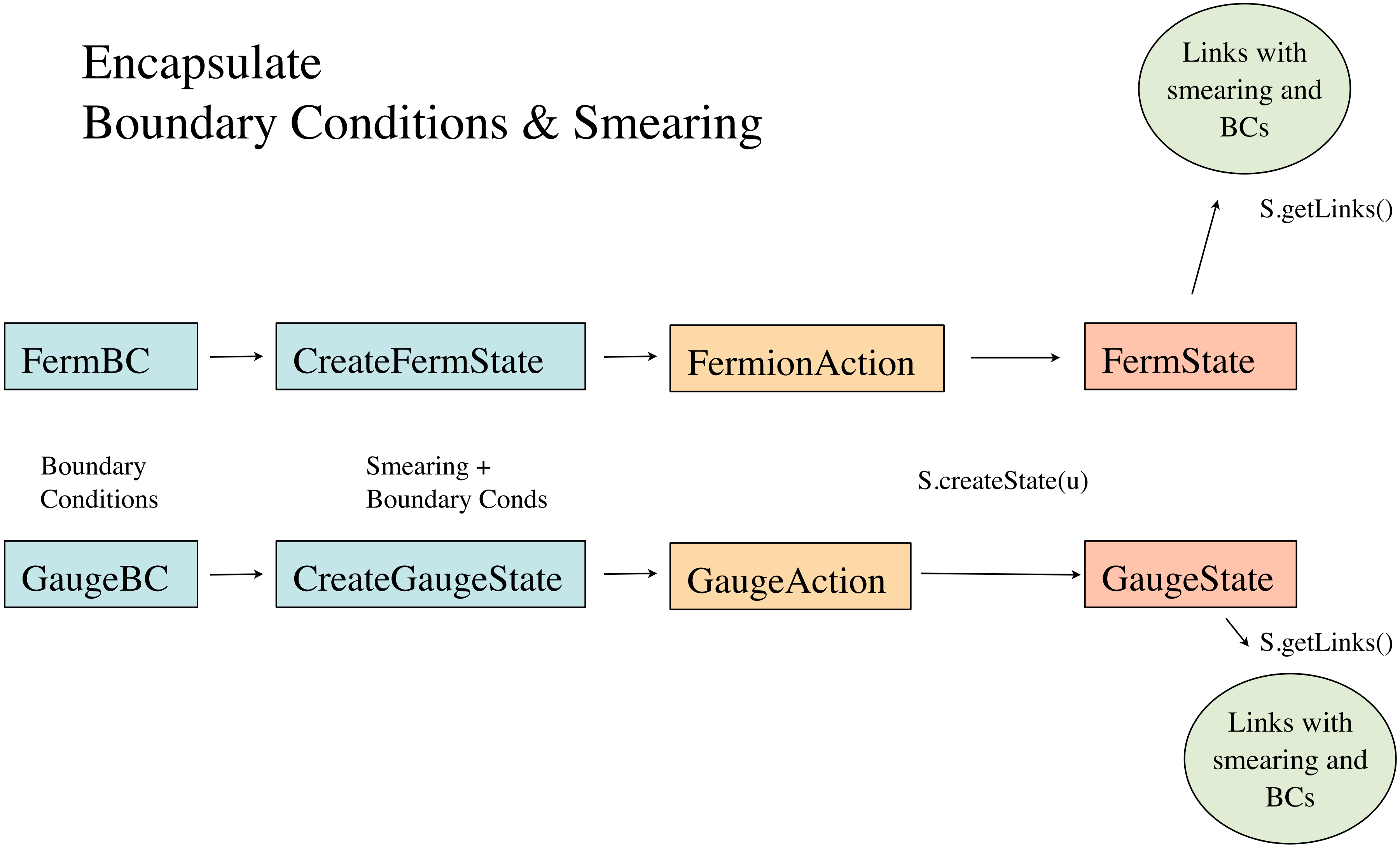
Chroma Key Base Classes: HMC



- Abstract means: templated on Gauge/Momentum types
- HMC written in terms of abstractions
- One needs concrete implementations as well of course.

Fermion and Gauge States

Encapsulate
Boundary Conditions & Smearing



FermBCs

- Interface for applying fermionic BCs
- Managed/Used by FermionAction and other GaugeBCs and FermBCs (eg Schroedinger Functional)
- Main memebrs:
  - **modifyU(u)** – Apply boundaries to gauge field
  - **modifyF(psi)** – Apply boundaries to fermion field
  - **zero(F)** – Zero Force on boundary (eg Schroedinger functional)

Linear Operators

- BaseType for matrices
- Templated on Fermion Type
- Function Object (has overloaded operator())

Target Vector

Source Vector

PLUS apply M
MINUS apply M^{+}

```
template<typename T>

class LinearOperator
{
public:
  virtual void operator() (T& chi, const T& psi, enum PlusMinus isign) const = 0;

  virtual const Subset& subset() const = 0;

  // ... others omitted for lack of space
};
```

Know which subset to act on

System Solvers

- Attempt to encapsulate various inverter strategies
  - Single systems: SystemSolver< FermionType >
  - Multi-mass: MultiSystemSolver< FermionType >

SystemSolver<T>

MultiSystemSolver<T>

LinOpSytemSolver<T>

$$M\psi = \chi$$

LinOpMultiSystemSolver<T>

MdagMSystemSolver<T>

$$M^\dagger M\phi = \chi$$

MdagMMultiSystemSolver<T>

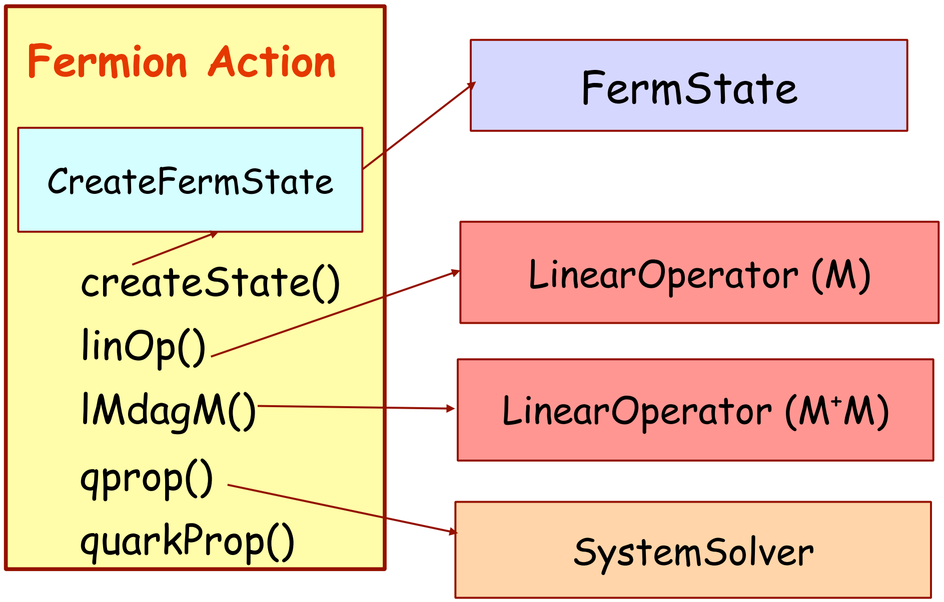
```
template<typename T> class SystemSolver {
public:
 virtual SystemSolverResults_t operator()(T& psi, const T& chi) const=0;
 virtual const Subset& subset() const=0;
};
```

operator() - performs solve

```
template<typename T> class MultiSystemSolver {
public:
 virtual SystemSolverResults_t operator()(multi1d<T>& psi, const multi1d<Real>& shifts,
                                          const multi1d<T>& chi) const=0;
 virtual const Subset& subset() const=0;
};
```

FermionActions

- Manages related Linear Operators, States and Solvers
- Not “action” in the true sense, does not know about flavour structure



Using Linear Operator

- Created by FermionAction (factory method)
- Typical Use Pattern:

```
// Raw Gauge Field
multi1d<LatticeColorMatrix> u(Nd);
typedef QDP::LatticeFermion T;
typedef QDP::multi1d<LatticeColorMatrix> P;
typedef QDP::multi1d<LatticeColorMatrix> Q;
FermionAction<T,P,Q>& S = ...;

Handle< FermState<T,P,Q> > state( S.createState(u) );

Handle<LinearOperator<T> >  M( S.linOp(state) ) ;

LatticeFermion y, x;
gaussian(x);

 (*M)(y, x, PLUS);
```

Create state for Fermion Kernel

Create LinearOperator (fix in links)

De-reference Handle and apply lin. op: y = M x

SystemSolverArray-s

- Similar Idea to SystemSolvers, but 5D fermions
- LinOpSystemSolverArray<T> to solve with M
  - works on multi1d<T> for 5D
- Similarly
  - MdagMSystemSolverArray<T> for $M^{\dagger}M$
  - MdagMMultiSystemSolverArray<T> for shifted

Qprop System Solvers

- Qprop-s are a special kind of system solver
  - solve for 1 component of a 4d quark propagator
    - For 5D actions deal with 5D source construction and 4D projection post solve
    - eg: DWFQprop, FermActQprop, ContFrac5DQprop

- QpropT-s are a 5D construction
  - solve for 1 component of a 5D quark prop, but don't project down
    - really this is just the same as LinOpSysSolverArray?
    - eg: FermAct5DQprop<T>, PrecFermAct5DQprop<T>

Choosing Implementations: Factories

- It is great to be able to code most of our code in terms of base classes, virtual functions and defaults
- However, somewhere the code must live for the implementations:
  - e.g. 2 Flavor Clover Action, DWF Linear Operator, Omelyan 2nd order Integrator etc.
- Various implementations can have different parameters:
  - e.g. Wilson Fermions, vs. Clover Fermions (c_sw)
  - e.g. Generic CG solver, vs. solver from QUDA
- Need a uniform way, to create the various objects
  - while allowing their implementations to vary
  - Textbook Object Oriented Construction Pattern: Factory

What do we mean?

What we don’t want:

```
switch(solver_type) {
case CG:
    invcg(M,x,y, params);
    break;
case BICG:
   invbicg(M,x,y,params);
   break;
case RELIABLE_BICG:
   invrelbicg(M,x,y,params);
   break;
// ... other case
default:
   // what's sensible? CG?
   // cross fingers...
   invcg(M,x,y,params);
  break;
};
```

- Why is this bad ?
  - everywhere we need a solver we may need to repeat the switch statement
  - adding a new solver can become painful: edit every switch statement
  - we would need a monster parameter structure, covering all possible solvers
  - what is a sensible default?

Object Factories

- Provide a uniform way to select and construct implementations of a given base class

Key

parameters in XMLReader

```xml
<InvertParams>
   <invType>CG_INVERTER</invType>
   <RsdCG>1.0e-7</RsdCG>
   <MaxCG>1000</MaxCG>
</InvertParams>
```

"CG_INVERTER"

theLinOpSystemSolverFactory

("CG_INVERTER", (*createCGInverter)())

("BICGSTAB_INVERTER", (*createBiCGStabInverter)())

Product (pointer to)

Chroma::LinOpSystemSolver<> *

Factory Advantages

- Encapsulate solver in a function-object (functor)
  - Use a factory to make the object
  - The created object knows what solver it is
    - no switch statement, just: `(*solver)(out,in)`
  - The object can have its own parameters rather than one big parameter struct for all solvers.
  - To add a new type of object (solver), one needs only to
    - add the source for the new type of object
    - register in the relevant factory
    - everywhere that kind of object was used before, will now be able to use the new object
  - Contrast with old way: would have had to find every 'switch' statement with that object type and add a new case.

Factory Implementation

- STL ‘map’ class used to create mapping between
  - a string (KEY) to identify which class to instantiate
  - a function to create the object, given XML parameters
  - the function must be ‘registered’ in the factory.
- We use an object factory implementation from the LOKI library (Alexandrescu et. al.)

Registration Functions

```
//! Creation function. Lives in eoprec_clover_fermact.cc
WilsonTypeFermAct<LatticeFermion,
        multi1d<LatticeColorMatrix>,
        multi1d<LatticeColorMatrix> >* createFermAct4D(XMLReader& xml_in,
                                                        const std::string& path)
{
      return new EvenOddPrecCloverFermAct(CreateFermStateEnv::reader(xml_in, path),
                                          CloverFermActParams(xml_in, path));
}

const std::string name = "CLOVER";   // Name to use
static bool registered = false;      // set to true when registering

bool registerAll()
{
  bool success = true;
  if (! registered)  {
   success &= Chroma::TheWilsonTypeFermActFactory::Instance().registerObject(name,
                                                              createFermAct4D);

   registered = true;
  }
  return success;
}
```

Measurements

- Aim: Encapsulate measurements as objects (rather than functions)
  - uniform interface
  - can create from a ‘description’
  - chroma application: a simple interpeter to cycle through these
- Very simple class: InlineMeasurement
- Has only 2 public methods:
  - operator(update_no) -- do the measurement
  - getFrequency() -- how often should the measurement be done
    - Originally from HMC when one didn’t want to measure on every trajectory

Named Objects

- Measurement Tasks are discrete ‘objects’
- Useful to share data between multiple measurements:
  - create a source in one task, and use it in another
- “Named Objects” were designed to do this.
  - Have a global ‘store’
  - Tasks can
    - create objects, with a name (string)
    - lookup/delete objects (using the name)
- Have special tasks (Measurements) to I/O named objects
  - Divorces I/O from the measurements themseves

Named Objects in Code and XML

eg: source creation:

```
TheNamedObjMap::Instance().create<LatticePropagator>(params.named_obj.source_id);
TheNamedObjMap::Instance().getData<LatticePropagator>(params.named_obj.source_id) =
                                                                    quark_source;

TheNamedObjMap::Instance().get(params.named_obj.source_id).setFileXML(file_xml);
TheNamedObjMap::Instance().get(params.named_obj.source_id).setRecordXML(record_xml);
```

In XML:

MAKE_SOURCE creates object

```
<elem>
  <Name>MAKE_SOURCE</Name>
  ...
  <NamedObject>
    <source_id>sh_source</source_id>
  </NamedObject>
 </elem>
 <elem>
  <Name>PROPAGATOR</Name>
   ...
  <NamedObject>
    <source_id>sh_source</source_id>
    <prop_id>sh_prop_0</prop_id>
  </NamedObject>
</elem>
```

Special "Measurement" Writes named object

```
<elem>
 <Name>QIO_WRITE_NAMED_OBJECT</Name>
   ...
 <NamedObject>
  <object_id>sh_prop_0</object_id>
  <object_type>LatticePropagator</object_type>
 </NamedObject>
 <File>
  <file_name>./sh_prop_0</file_name>
  <file_volfmt>MULTIFILE</file_volfmt>
 </File>
</elem>
```

Stopping point

- Discussed
  - Capturing mathematical structure with inheritance
  - some of the main Chroma class abstractions
  - Measurements
- Discussed Factories, for creating instances of these
- Possible continuations
  - QDP++ and Chroma and GPUs
  - Design Patterns in Chroma
  - XML Writing Guide
  - Tutorials 2 and 3