Aspects of the Class Structure in Chroma

Bálint Joó (bjoo@jlab.org)
Jefferson Lab, Newport News, VA

given at

HackLatt'06

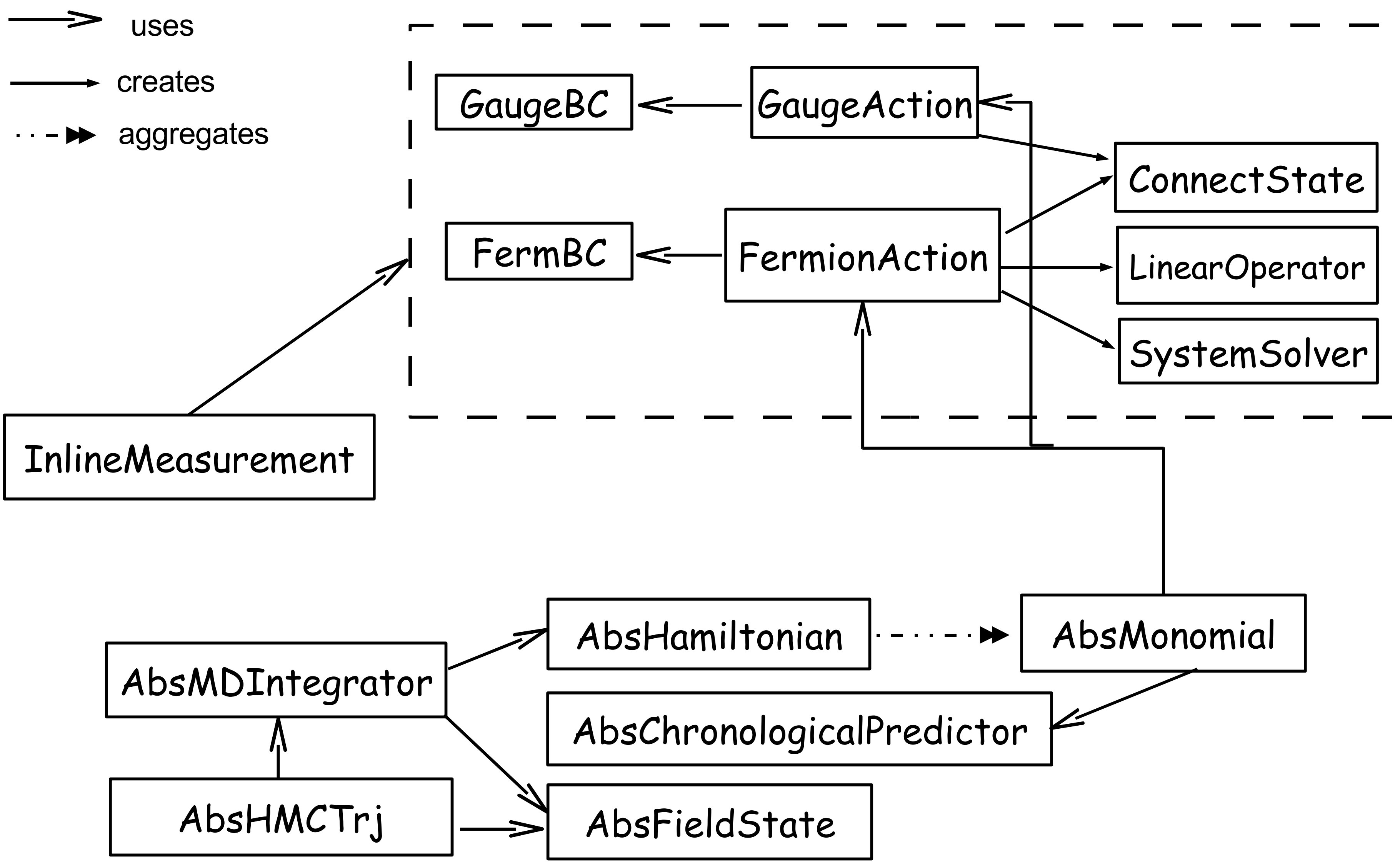
NeSC, Edinburgh

March 29, 2006

Philosophy

- Code as much as possible in terms of abstract / base classes and virtual functions
- As classes are derived try and write 'defaults'
  - Try to write things only once.
  - Refactor rather than duplicate and extend
  - Aldor category default influence
- Force type correctness where possible using the Covariant return rule
  - Mirrored hierarchy trees
- XML and factories - polymorphism of parameters

A Broad Overview of the Base Classes



ConnectState

- In order to be useful raw gauge field states need extra info eg:
  - Boundary conditions
  - link smearing
  - eigenvectors/values
- ConnectState manages this
- Created by
  - FermionAction

```
// Raw Gauge Field
multi1d<LatticeColorMatrix> u(Nd);
FermionAction& S = ...;

Handle<ConnectState> state(
                    S.createState(u) );

Handle<LinearOperator<LatticeFermion> >
    S.linOp(state);
```

Raw Field

Knows about BC-s

Factory Function: applies BC-s to field

LinearOperator gets field with BCs applied

ConnectState

- Some Derivations of ConnectState
  - SimpleConnectState (just u and BCs)
  - EigenState (u and e-values & vectors)
  - StoutState (in development)
  - OverlapState(deprecated older version of EigenState)
- Member Functions:
  - getLinks() - return internal fields
  - deriv() - force w.r.t thin (unsmeared links)

FermBCs

- Interface for applying fermionic BCs
- Templated on type of FermionField
- Produced by factory
- 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)

LinearOperator

- 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 OrderedSubset& subset() const = 0;

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

Know which subset to act on

LinearOperator

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

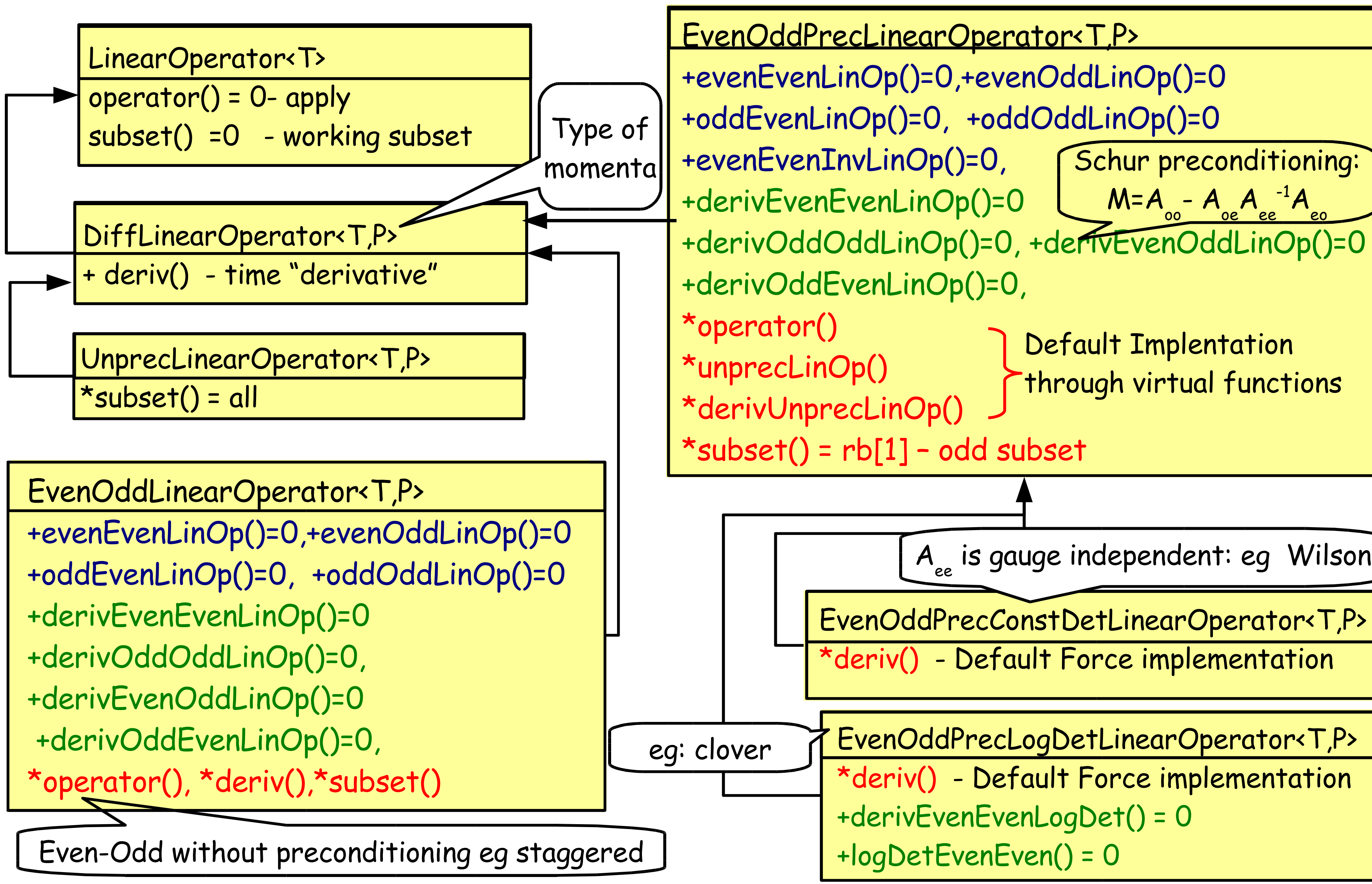
```
// Raw Gauge Field
multi1d<LatticeColorMatrix> u(Nd);
FermionAction& S = ...;

Handle<ConnectState> state( S.createState(u) );

Handle<LinearOperator<LatticeFermion> >  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

Some Derivations of LinearOperator

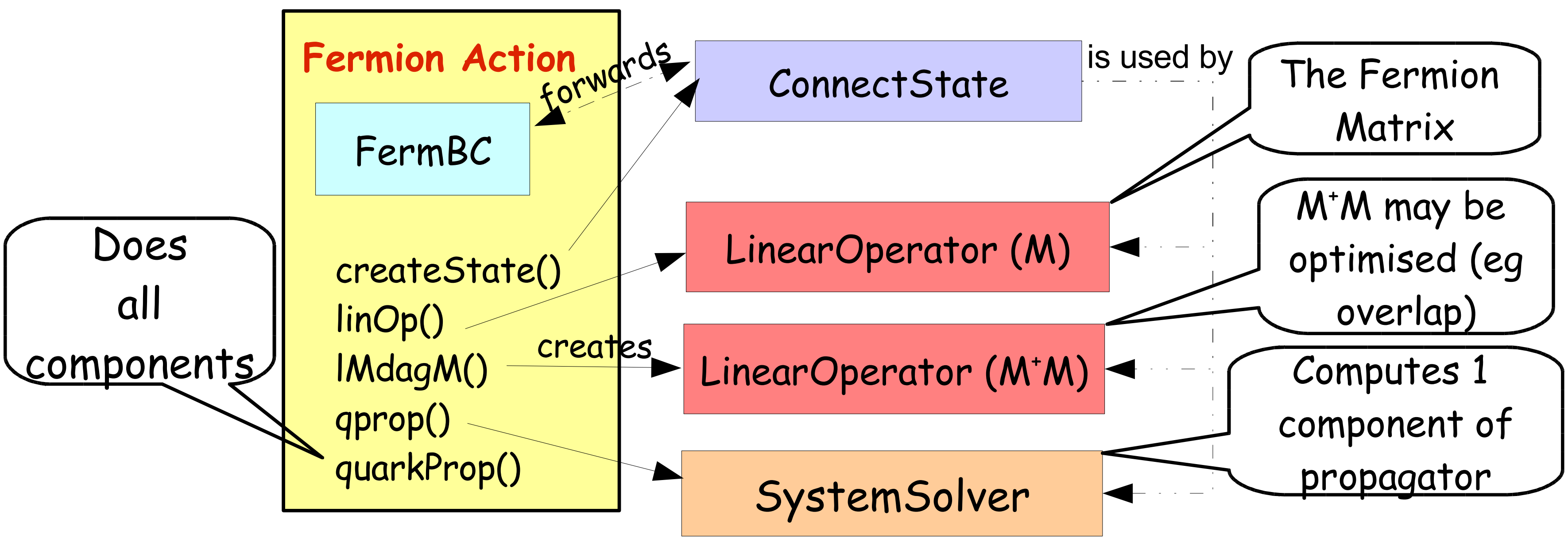


Linear Operators

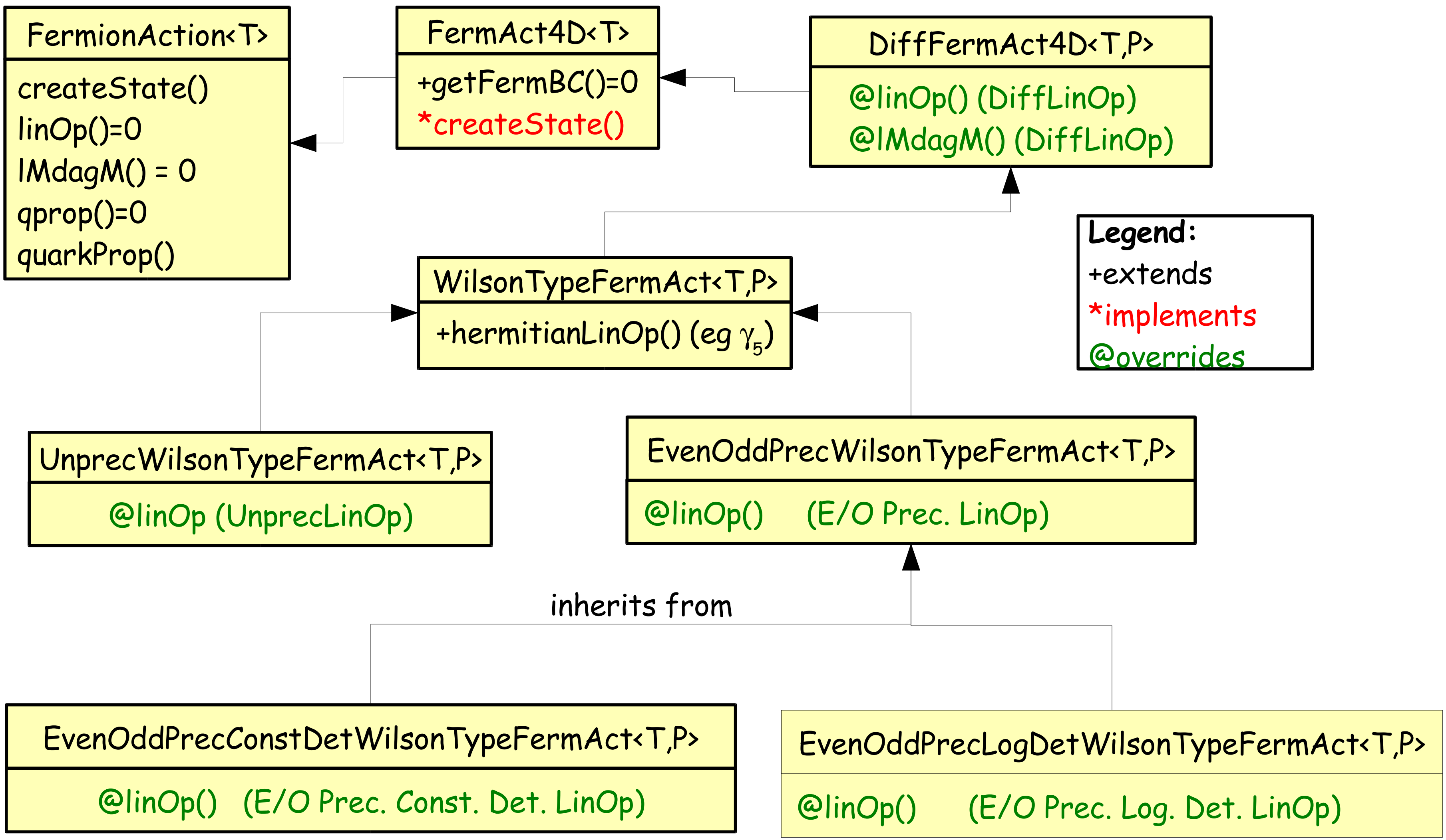
- Similar hierarchy is mirrored with 5D variants
  - convention XXXLinOpArray in name
- Key points
  - Differentiable Linear operator knows how to take derivative wrt to embedded gauge field
  - the second step of chain rule done by ConnectState (deriv wrt thin links)
  - Wilsonesque Hierarchy follows (4D Schur like) Even Odd preconditioning (rather than Hermiticity etc)
  - Workhorse of the fermion sector.

Fermion Actions

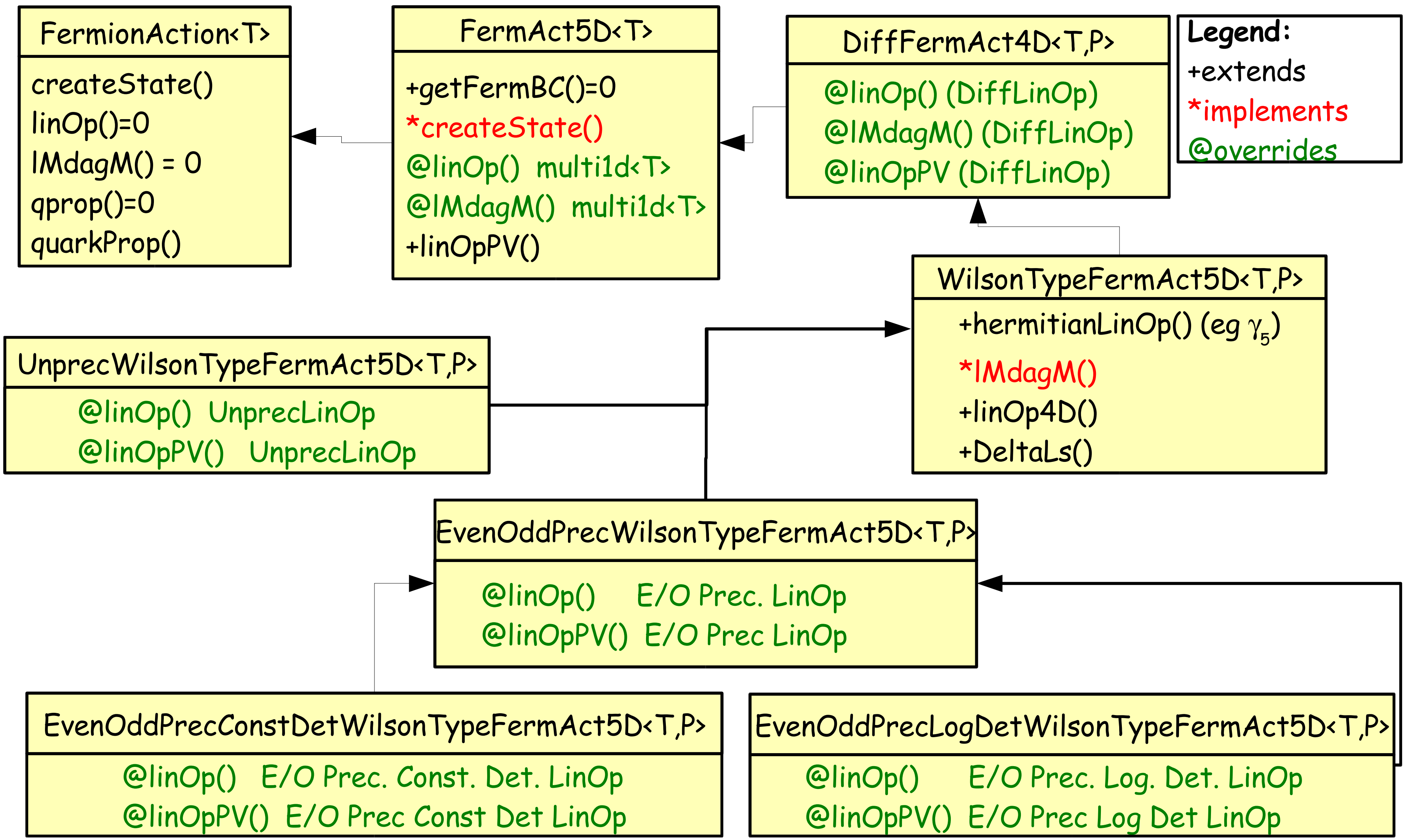
- Manages related Linear Operators, States and propagator Inverters
- Created by Factory pattern
- Not "action" in the true sense, does not know about flavour structure (see monomials later)



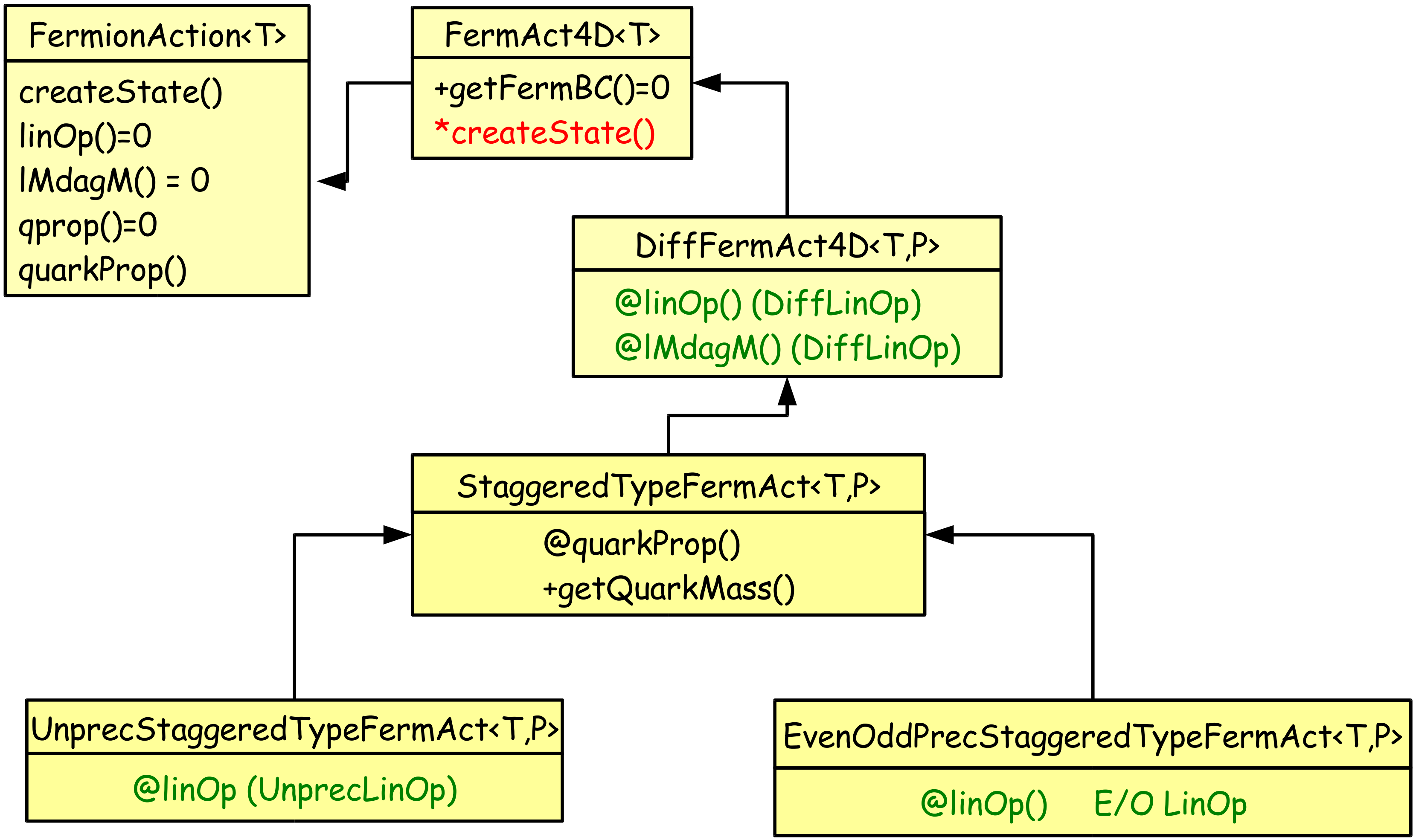
4D Derivations of Fermion Action



5D Derivations of Fermion Action



Staggered Derivations of FermionAction

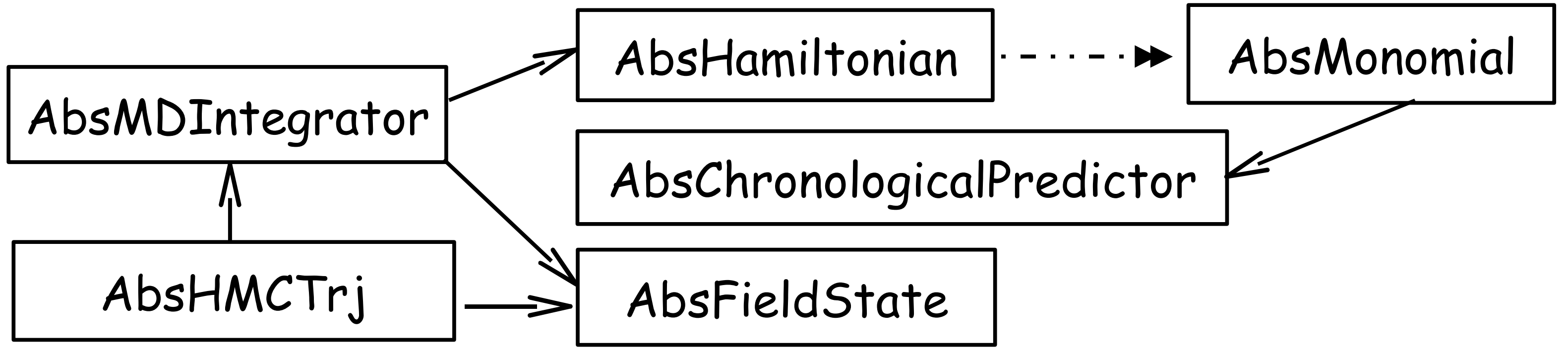


Notes on Fermion Action

- From DiffFermAct onwards, inheritence tree shadows inheritance of Linear Operators.
- Travelling towards the leaves of inheritance tree
  - Type "Restriction" allows specialisation of say qprop()
- Travelling towards root of the tree
  - Type information loss
    - Don't know which branch we came up on
  - Need C++ RTTI to be able to recover type info
    - Use C++ dynamic_cast<> mechanism to attempt to go down a particular branch

HMC Sector

- Actual HMC part is quite simple – mostly in terms of abstract classes
- Key classes:
  - Monomial, Hamiltonian, FieldState
  - Integrator, HMC
- The code for this is in chroma/lib/update/molecdyn



AbsFieldState<P,Q>

- This state of fields is a phase space field state
  - The templates P and Q specify types of canonical momenta and coordinates
- GaugeFieldState – specialises P and Q to be of type multi1d<LatticeColorMatrix>
- The HMC related classes act on AbsFieldState-s
  - AbsHamiltonian and AbsMonomial compute things on states
  - AbsHMCTrj and AbsIntegrator evolve the states

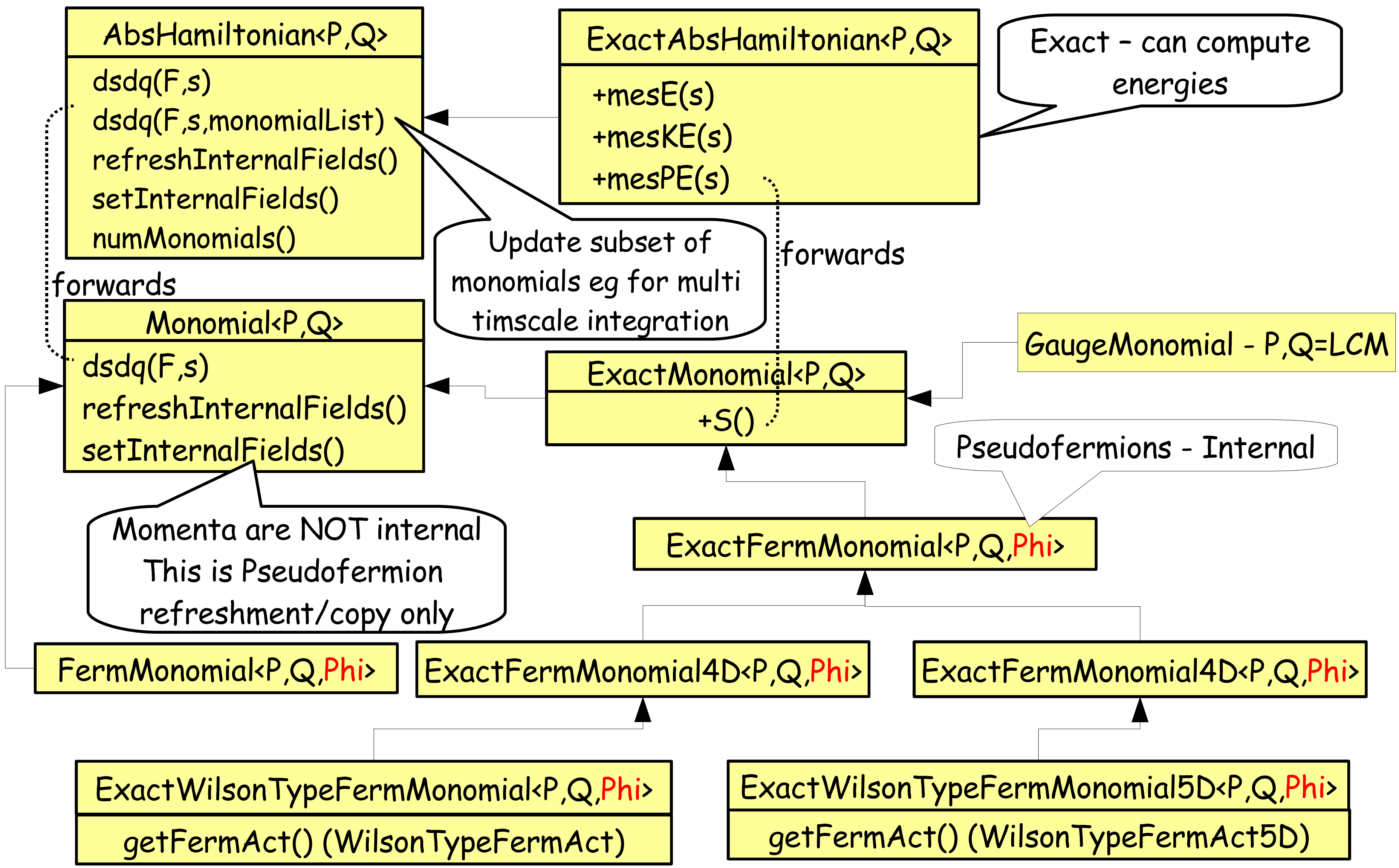
Hamiltonians and Monomials

- We evolve the Hamiltonian System

  $H(p,q) = (\frac{1}{2})\, p^2 + \sum_i S_i$

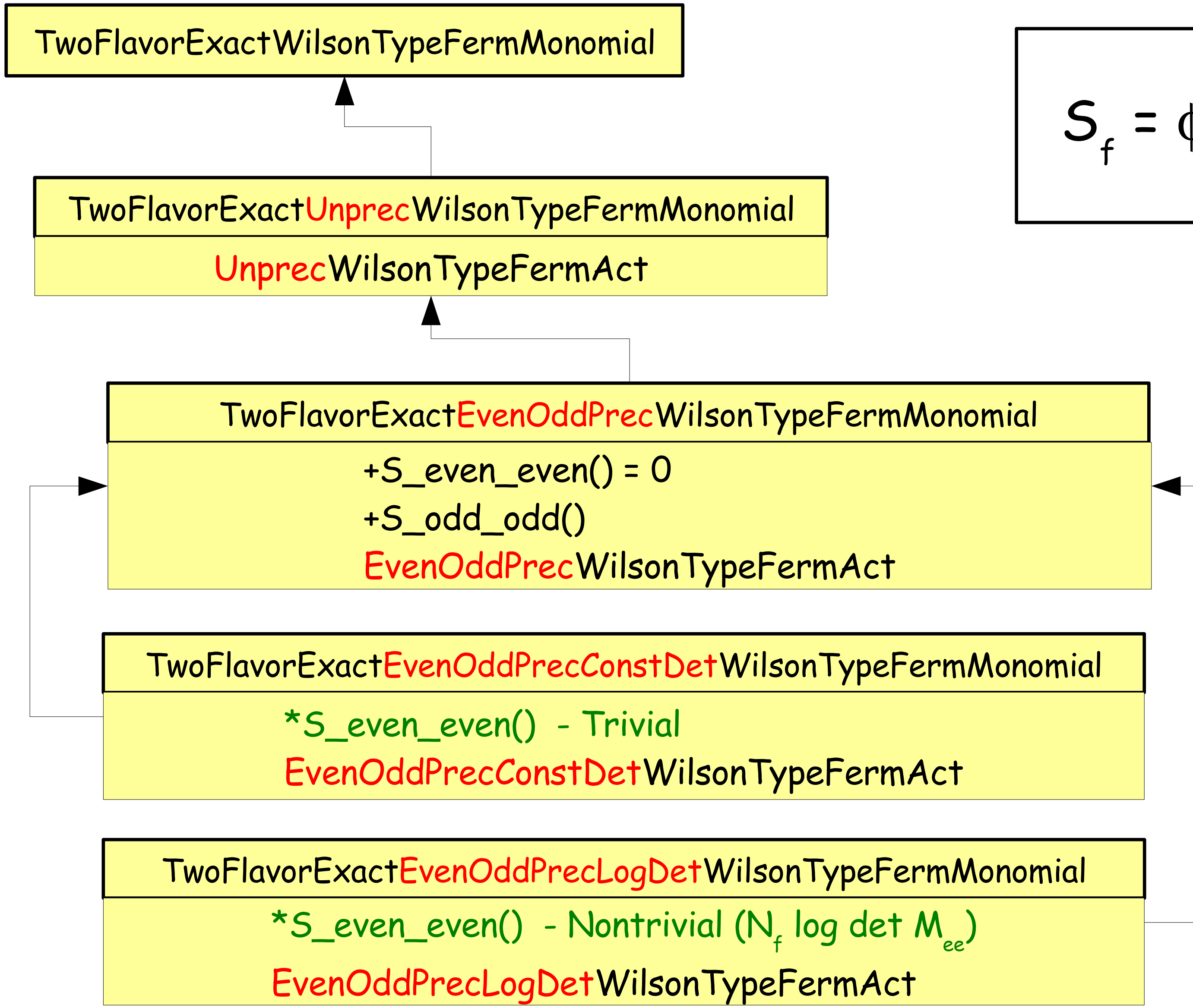
- We refer to S_i as Monomials (blame Tony!)
- In each Monomial can contribute
  - MD Force
  - Contribution to the Energy (if it is "exact")
- In terms of classes the Hamiltonian aggregates the Forces and Energies of its component Monomials.
- The hard work is in the Monomials

Hamiltonian & Monomial



Two Flavour Fermionic Monomials

$$S_f = \phi^{+}(M^{+}M)^{-1}\phi$$



Rational One Flavour Like Monomials

$$S_f = \phi \, (M^+ M)^{-a/b} \, \phi = \phi \, (\sum p_i \, [M^+ M + q_i \,]^{-1} \,) \, \phi$$

- a and b can be used to implement Nroots approach
- Rational approximation expressed as PFE
- Use Multi Mass Solver Internally
- Similar Hierarchy to Two Flavour Monomials
- Not yet split EvenOddPrec into ConstDet and LogDet

Hasenbusch Like Monomials

$$S_f = \phi^{+} [M_2 (M^{+}M)^{-1} M_2^{+}] \phi$$

- Implements Two Flavour Hasenbusch Like Ratio of determinants

$$\det(M^{+}M) / \det (M_2^{+}M_2)$$

- Does not automatically include term to cancel the determinant with M_2
- Need to add this in with a normal 2 flavor monomial.

LogDetEvenEven Monomials

- A monomial that simulates

$$\det (M_{ee})^N = N \log \det M_{ee}$$

- for Clover like actions (clover is only one so far)
- Factor even-even part of the clover term out and use Nroots or Hasenbusch acceleration for the odd-odd part only
- Downside:
  - in clover case duplicates storage of clover term
  - May also duplicate computation with EvenEven part

Chronological Solvers

- Two flavour monomials can make use of chronological predictors
- A chronological predictor is a solver starting guess STRATEGY
- Strategies available
  - Zero Guess
  - Previous Solution
  - Linear Extrapolation from last two solutions
  - Minimal Residual Extrapolation

MD Integrators

- Function objects -- ie use operator()
  - destructively change/evolve AbsFieldState - s
- share crucial components in a namespace, eg:
  - leapP() : $p_{new} = p_{old} + dt\ F$; leapQ(): $q_{new} = q_{old} + dt\ p$
- Integrators make use of Hamiltonian to compute forces for all of or some of the monomials
- Multi timescale integrators – Thanks Carsten!
  - Can put sets of monomials on different timescales
  - Cannot split a single (eg rational) monomial onto many timescales yet! This is work in progress.

Run time binding with XML

- Allows mix and match of fermion actions, boundaries, etc at run time.
- XML bound to strings in param structs.
- acts as polymorphic parameter structure.
- Factories used to create correct objects when needed

```
<Monomials>
 <elem>
   <Name>
   <InvertParam/>
   <FermionAction>
     <FermAct>WILSON</FermAct>
      <FermBC/>
   </FermionAction>
   <ChronologicalPredictor>
     <Name>LAST_SOLUTION_4D_PREDICTOR</Name>
   </ChronologicalPredictor>
  </elem>
</Monomials>
```

Factory product Key

```
struct TwoFlavorWilsonTypeFermMonomialParams
{
  TwoFlavorWilsonTypeFermMonomialParams();

  // Constructor from XML
  TwoFlavorWilsonTypeFermMonomialParams(
       XMLReader& in,
       const std::string&  path);

  InvertParam_t inv_param;
  std::string ferm_act;
  std::string predictor_xml;
};
```

"Inline" Measurements

- Originally designed to allow inline measurements from within gauge evolution algorithms
- Function objects
  - operator() called to perform the measurements
  - takes Output XML writer as parameter
  - Communication between measurements through named objects
    - essentially a virtual filesystem forced by slowness of QIO performance on QCDOC – writing objects to scratch directories takes the age of the universe

Named Objects

- Templated type to encapsulate objects
- Follows QIO structure: eg has File and Record XML
- Named objects stored in a map
  - associates name with named object
  - create/delete/lookup methods to manipulate map
- Special Inline Measurements to read/write objects to/from disk and named object maps.
- Divorces I/O from measurements completely
- Recent change: even input gauge field comes from named objects.

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>
```

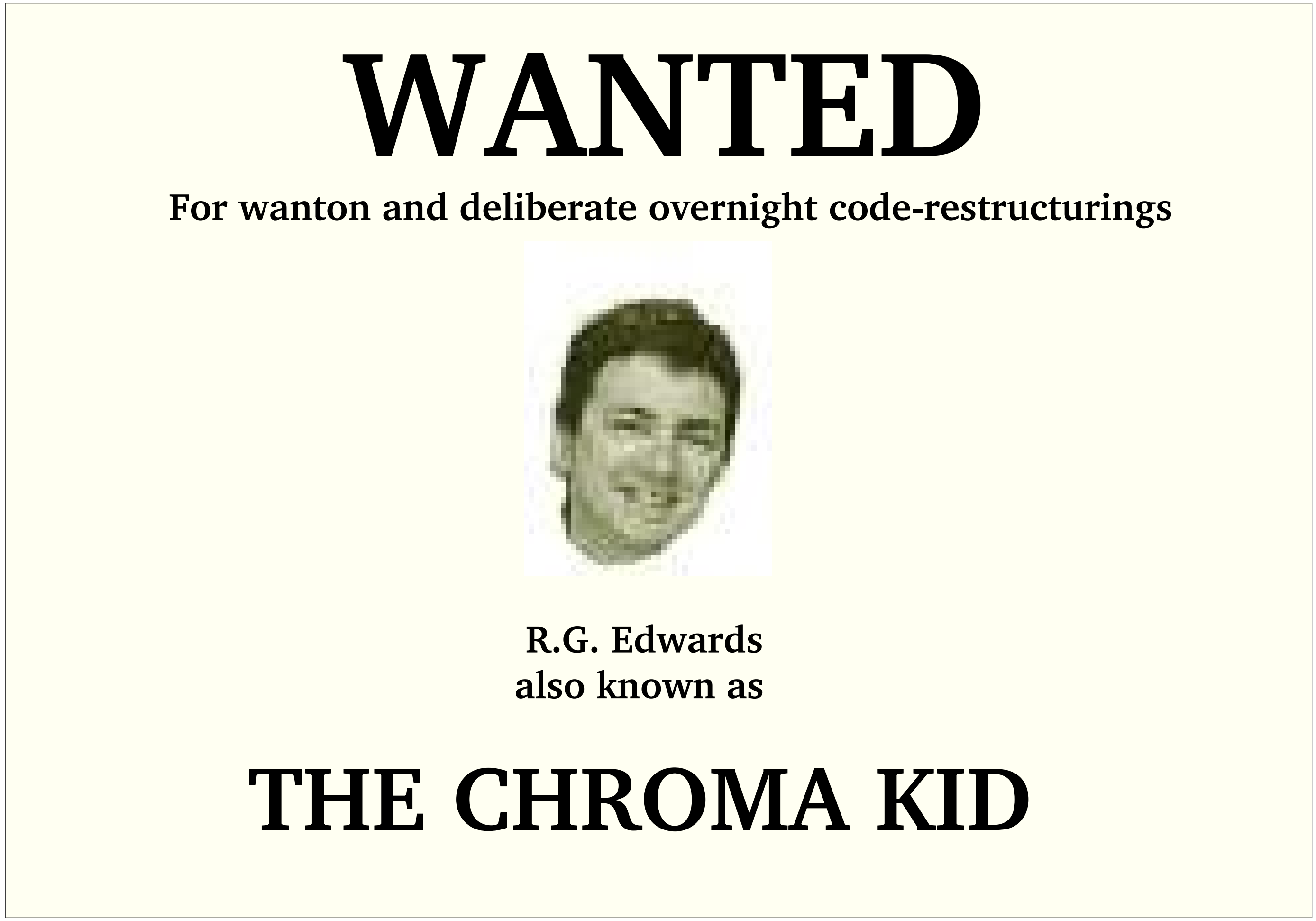
PROPAGATOR uses the source, creates prop

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>
```

Also: Tasks to read and erase objects to/from map

Changes in the wind

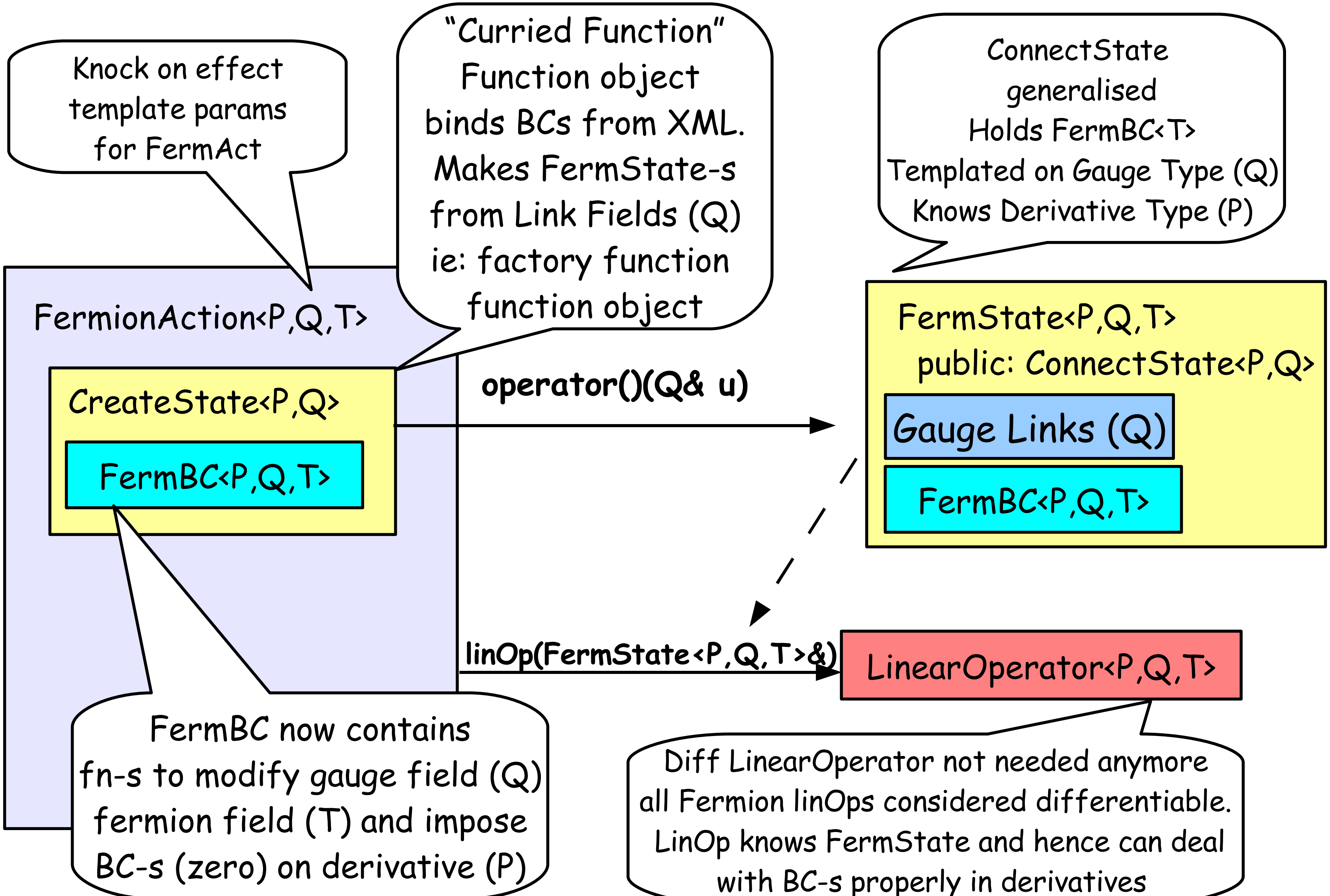


- Like all codes Chroma too must evolve with the times
- I have described chroma v2 here. v3 is next

Planned Changes to Class Structure in v3

- Push boundary conditions into ConnectStates
  - so we can deal correctly with non-trivial boundary conditions in the LinearOperator derivatives
    - eg: Schrodinger Functional BCs
  - so we can factor ConnectStates away from FermionActions
    - eg: to do link smearing independent of FermionAction
    - potentially need to impose BCs after every smearing iteration.
- Project driven – timing dictated by proposal deadline
  - SF BCs needed for JLab aniso clover project

How the changes will look:



Comments

- Fix current ConnectState inconsitencies
  - currently initialised with multi1d<LatticeColorMatrix>
  - will change to template type Q
  - consistent with HMC – evolution etc
- Inheritance with 5D FermActs wins nothing
  - doesn't reduce duplication
  - Uncouple – have FermActs and FermActArrays
- Maintain backward compatibility with XML structure and props etc.
  - But not in the API, sorry.

Summary and Conclusions

- Simple structure in terms of base classes and virtual functions
- Virtual functions not used for speed critical operations – no big inefficiency is introduced.
- "Mirrored" hierarchy of derivations:
  - Covariant Return Rule
- Nodes on class derivation tree supply default behaviour
- Detailed leaf-class object creation by factories.
  - Run time "binding"

Summary and Conclusions II

- Crucial Interfaces
  - LinearOperator
  - Boundary Conditions
  - ConnectState -s
  - FermionAction-s
  - Monomials
    - Two flavour
    - Rational
    - Hasenbusch
    - Gauge

Summary and Conclusion III

- Measurement Tasks
  - Data flow through Named Objects
  - Named Object I/O managed through special measurement tasks
  - Visual Grid based Chroma anyone?
- General
  - We have learned a lot about writing Object Oriented Lattice QCD software through writing Chroma
  - Hopefully useful tool to community (definitely to us)
  - We are continually working towards improvements