Aspects of the Class Structure in Chroma

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

given at

HackLatt'08

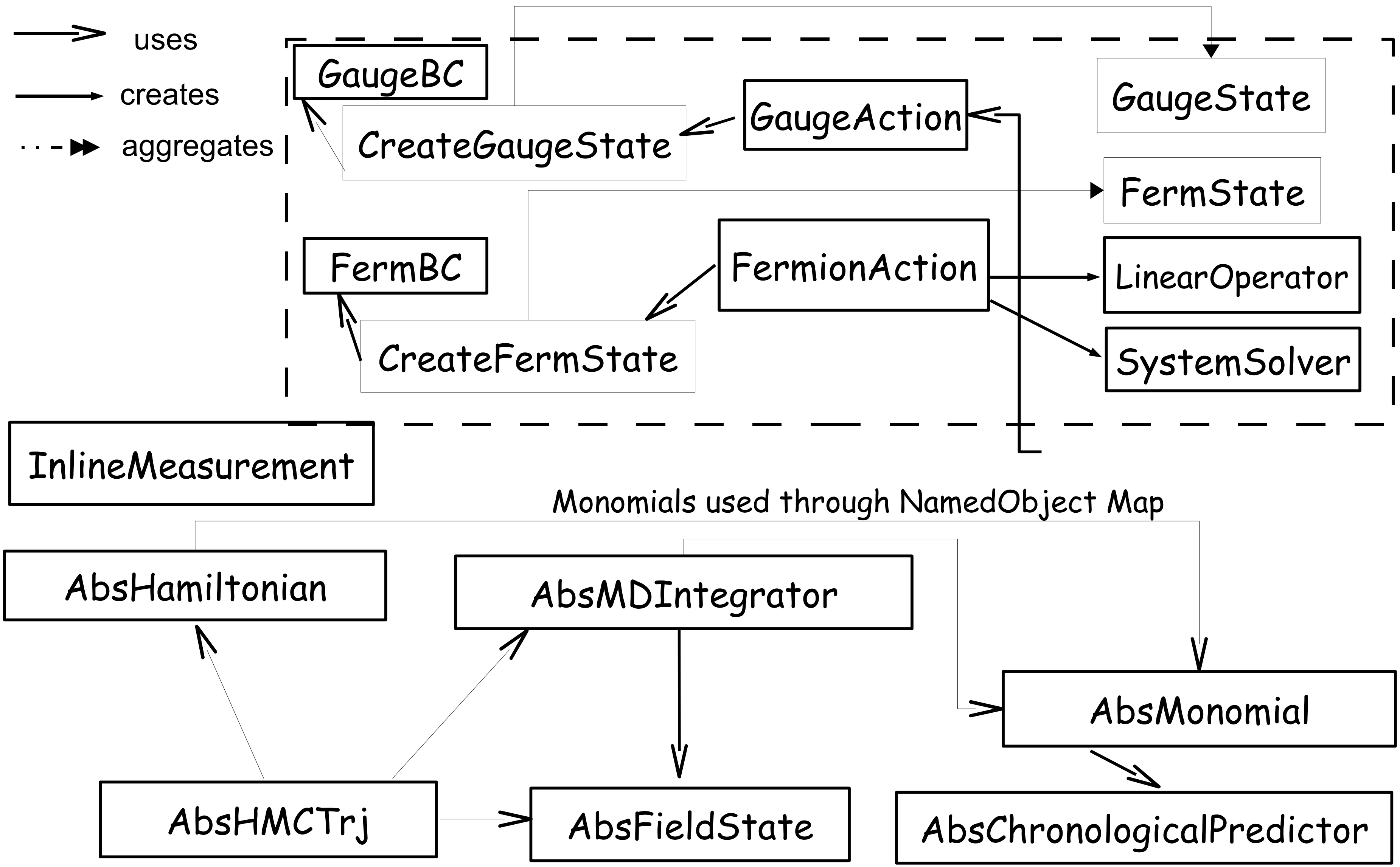
NeSC, Edinburgh

April 2-4, 2008

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
- Object Factories: run time binding to classes
  - You want an object that implements class "X"
  - You give the string "X" and an XML snippet containing the parameters to a factory
  - The factory returns a pointer to a newly created "X"

A Broad Overview of the Base Classes

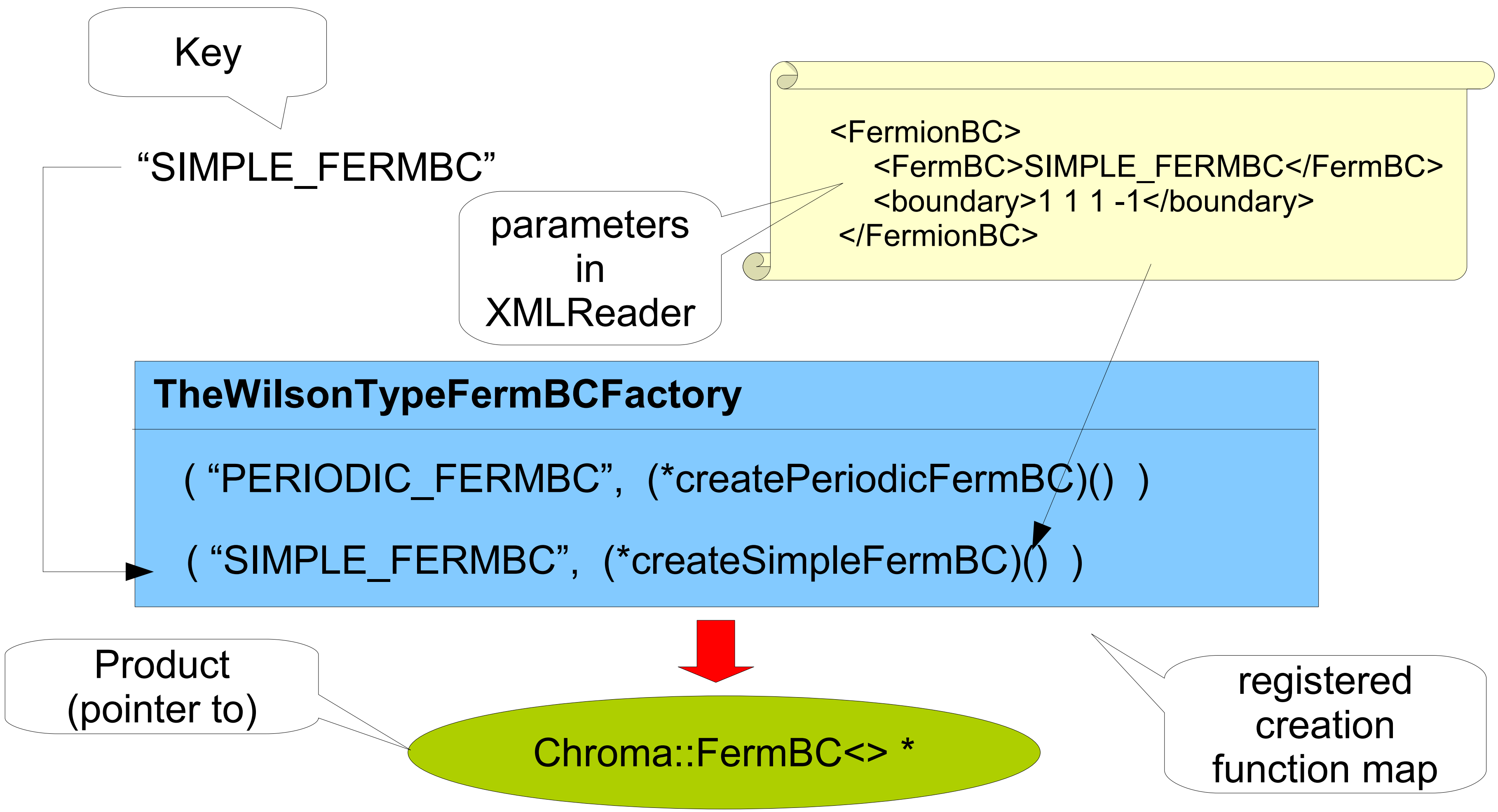


Base Classes/Implementations

- The base classes provide interfaces (primarily)
- Functionality is provided by derived classes (implementations)
- C++ does not allow you to create the base class
  - because it has virtual functions – its incomplete
- Different implementation can have different parameters
  - ie Wilson fermions need a mass parameter
  - DWF also need Ls and a domain wall height - M_5

Object Factories

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



Industrial Landscape

- We (ab)use the factory construction everywhere
  - FermStates (thin, stout, etc) & Boundaries
  - FermionActions (see later)
  - Selecting MD comonents (monomials, integrators)
  - Selecting the types of sources, sinks
  - Selecting inverters
  - Creating Measurement tasks (next talk)
    - selecting I/O tasks
    - managing the named object store etc

Aspects of the main classes

GaugeState/FermState

- In order to be useful raw gauge field states need extra info eg:
  - Boundary conditions
  - link smearing
  - eigenvectors/values
- GaugeState/FermState manages this
- Created by
  - CreateGaugeState / CreateFermState (directly)
  - GaugeAction / FermionAction (indirectly)
- Used by: LinearOps, Gauge/Fermion Monomials,etc

GaugeState/FermState

- Some Derivations of ConnectState
  - SimpleFermState / SimpleGaugeState
    - just u and BCs
  - StoutGaugeState/ StoutFermState
  - EigenConnectState
    - u, Bcs and Fermionic EIgenvalues/Vectors
- Base Class Member Functions:
  - getLinks() - return modified links
  - deriv() - force w.r.t thin (unsmeared links)
  - getBC(), getFermBC() - get boundary conditions

FermBCs

- Interface for applying fermionic BCs
- 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)

CreateState classes

- To make a state I need, the gauge field, boundaries and potentially other things (smearing etc)
  - f: (u, BCs, smearing, etc.) -> 'state'
- We'd like to have a functionality where we fix BCs, smearing etc, but not the gauge field
  - g(BCs, smearing etc.): u->'state'
- Note in g above, everything is frozen in except 'u'
  - aka. Currying / Partial Function Evaluation
- CreateState object acts as 'g'. For every kind of ConnectState we have an appropriate 'CreateState'

LinearOperator

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

```
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
};
```

Target Vector

Source Vector

PLUS apply M
MINUS apply M^{+}

Know which subset to act on

LinearOperator

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

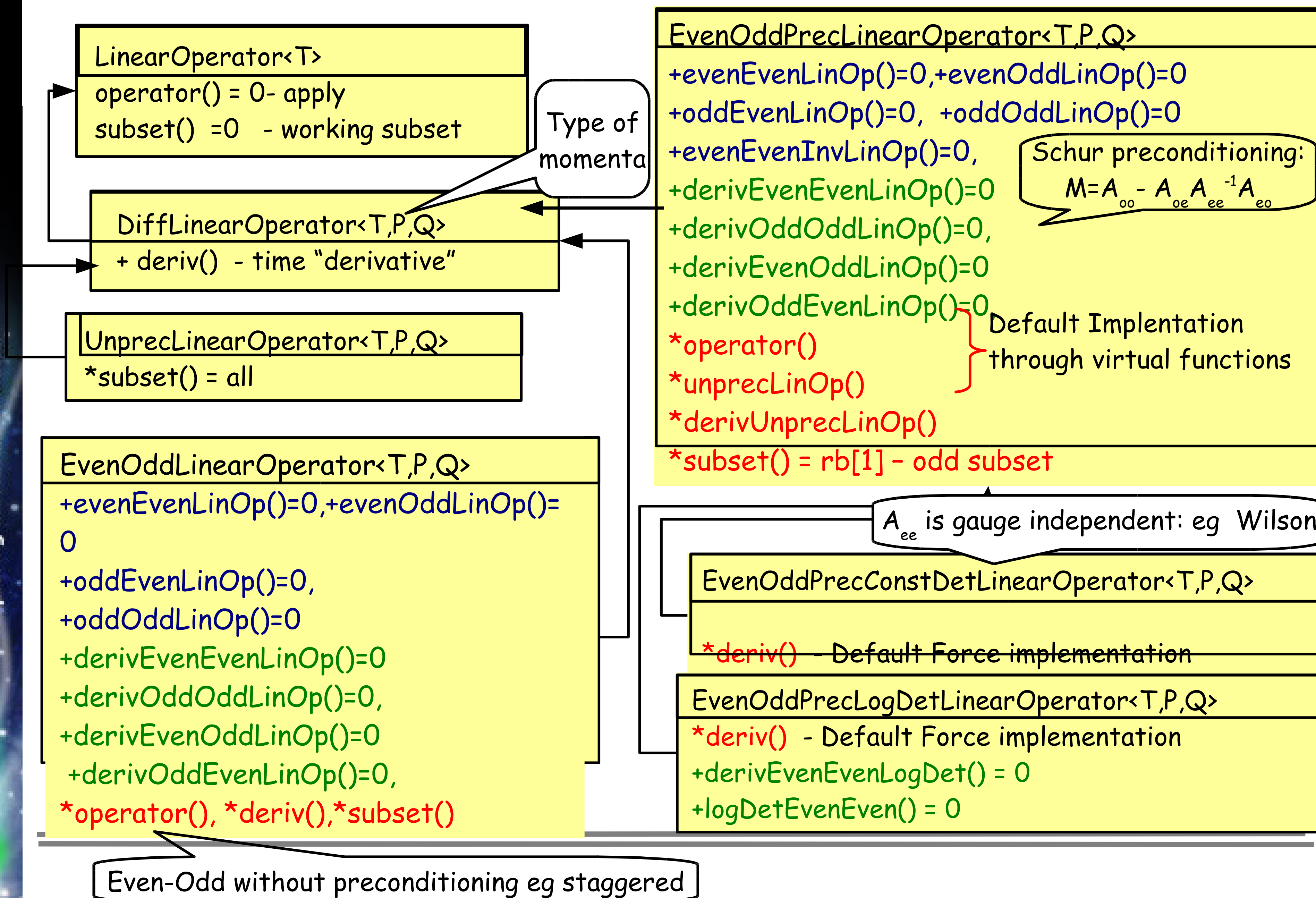
```
// Raw Gauge Field
multi1d<LatticeColorMatrix> u(Nd);
typedef QDP::LatticeFermion T;
typedef QDP::multi1d<LatticeColorMatrix> P;
typedef QDP::multi1d<LatticeColorMatrix> Q;
FermionAction<T,P,Q>& S = ...;

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

Some Derivations of LinearOperator

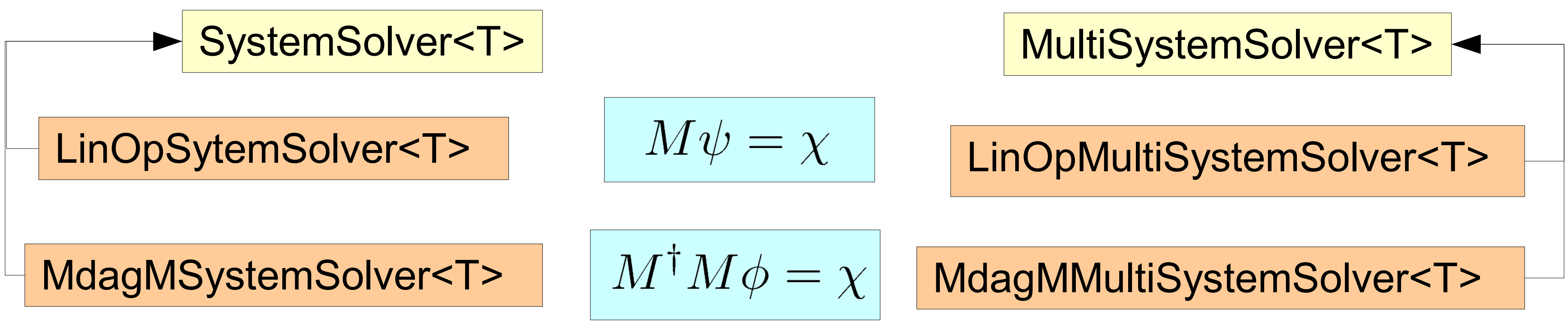


Linear Operators

- 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 FermState (deriv wrt thin links)
  - Wilsonesque Hierarchy follows (4D Schur like) Even Odd preconditioning (rather than Hermiticity etc)
  - Workhorse of the fermion sector.

System Solvers in 4D

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




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

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;
};
```

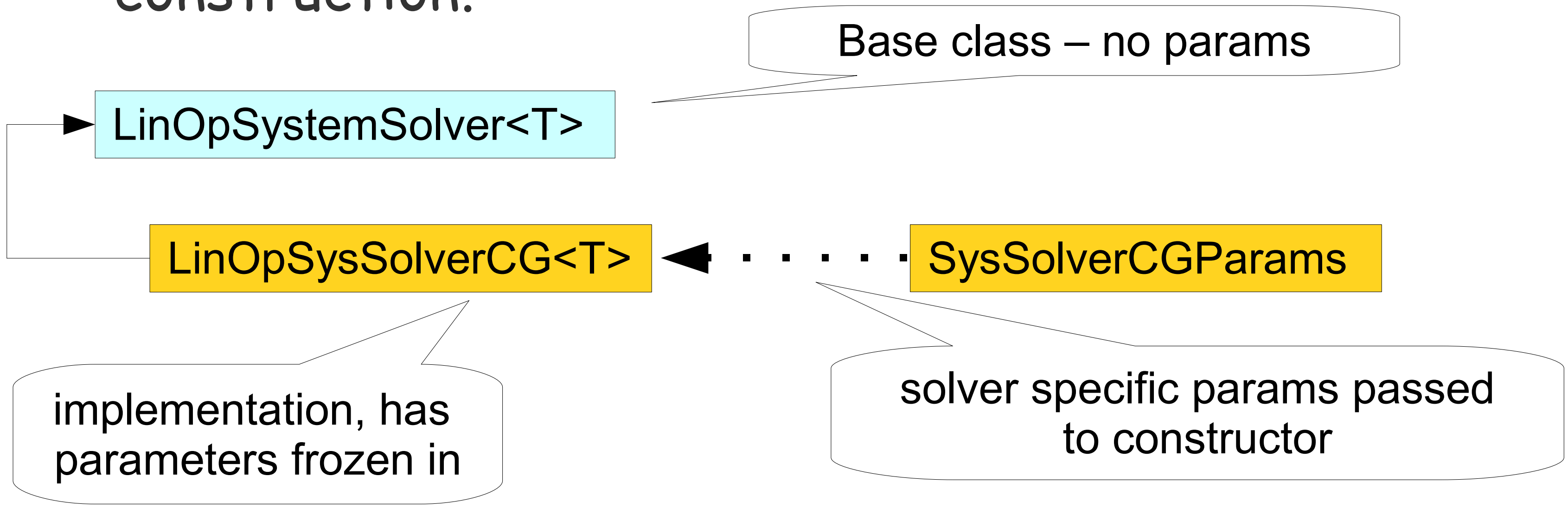
operator() - performs solve

System Solvers in 5D

- Similar situation/inheritance tree as 4D but classes now have "Array" on the end to indicate they work with arrays of type T e.g.:
  - LinOpSystemSolverArray<T> to solve with M
    - works with multi1d<T> for 5D
  - MdagMSystemSolverArray<T> to solve with $M^{+}M$
  - MdagMMultiSystemSolverArray<T> to solve a multi-shift system (eg for forces)

More on SystemSolvers

- Note absence of parameters, residua etc from the interfaces.
  - These have different meanings to each solver
  - Dealt with in the derived classes (implementations)
  - Typically 'frozen' into the derived classes on construction.

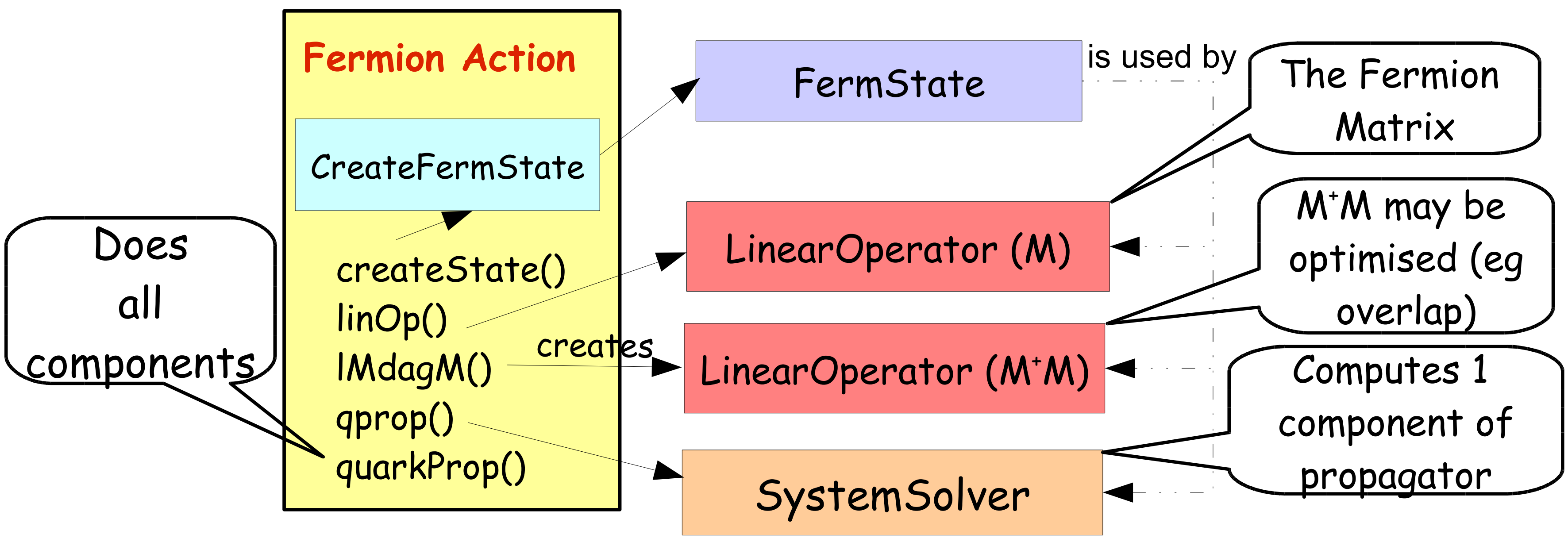


Qprop classes

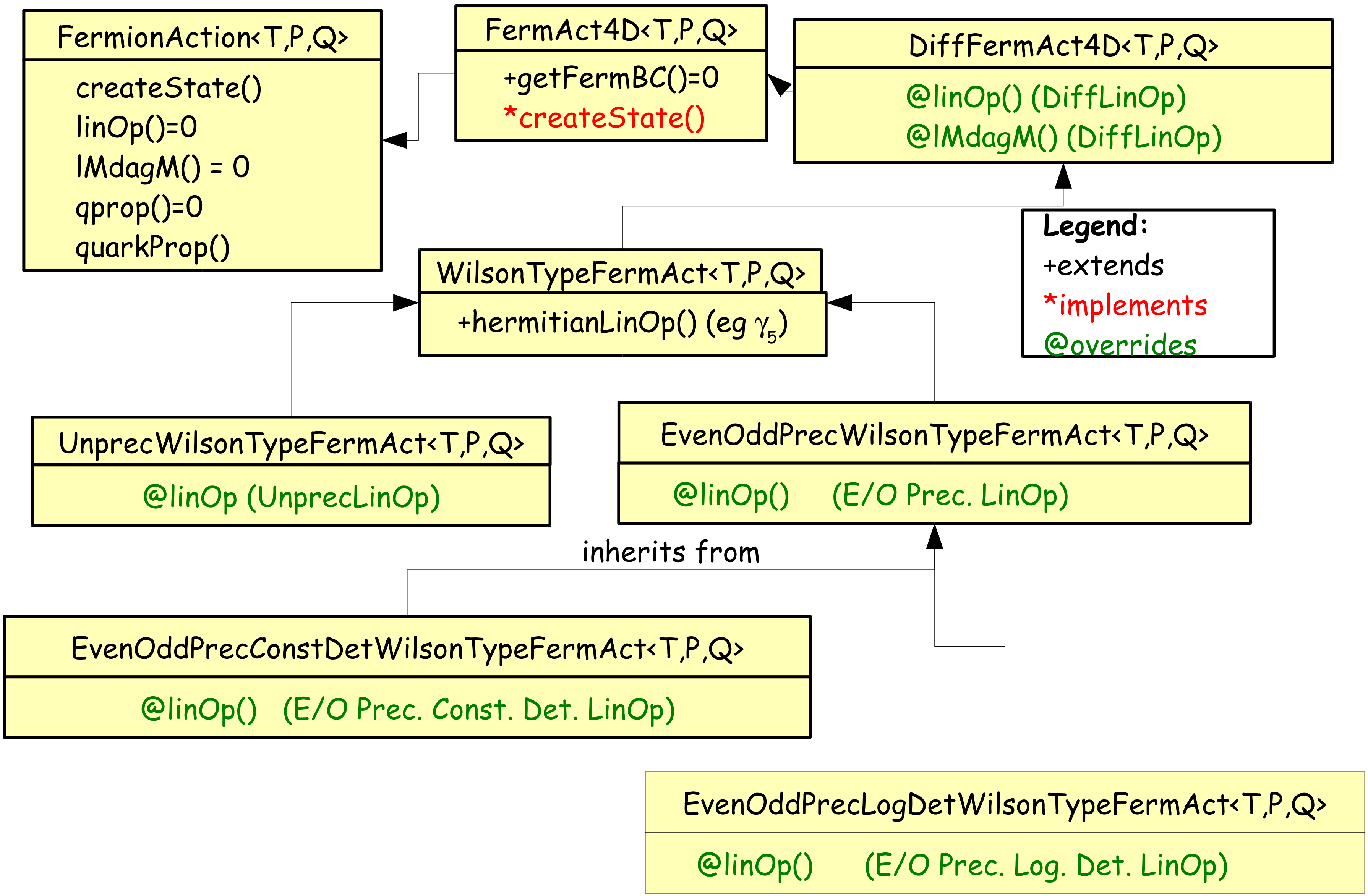
- Qprop-s are a special kind of system solver
  - solve for 1 component of a 4d quark propagator
    - For 5D actions deal with 5D source construction and 4D projection post solve
    - eg: DWFQprop, FermActQprop, ContFrac5DQprop
- QpropT-s are a 5D construction
  - solve for 1 component of a 5D quark prop, but don't project down
    - really this is just the same as LinOpSysSolverArray?
    - eg: FermAct5DQprop<T>, PrecFermAct5DQprop<T>

Fermion Actions

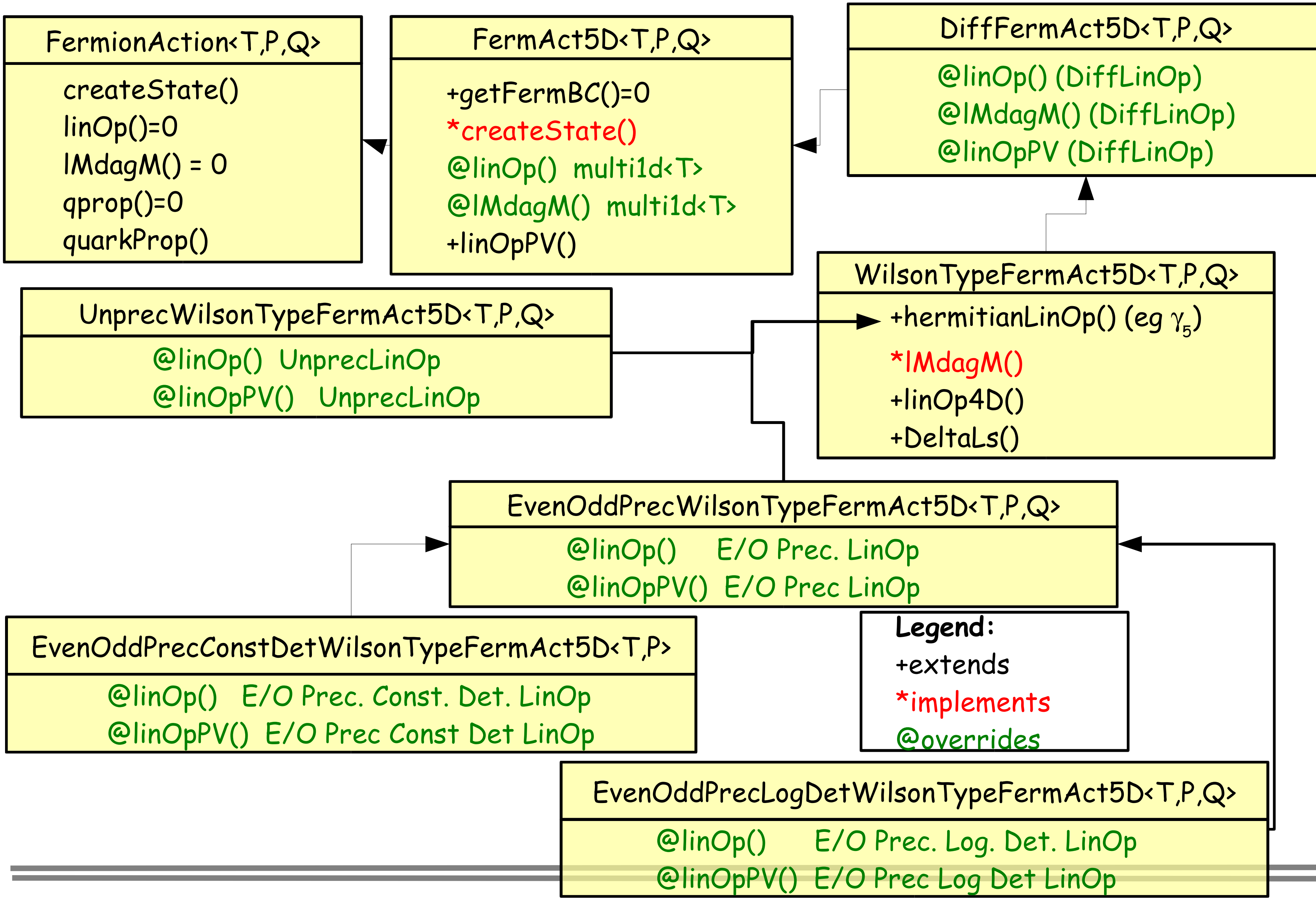
- Manages related Linear Operators, States and propagator Inverters
- Created by Factory pattern
- Not "action" in the true sense, does not know about flavour structure (see monomials later)



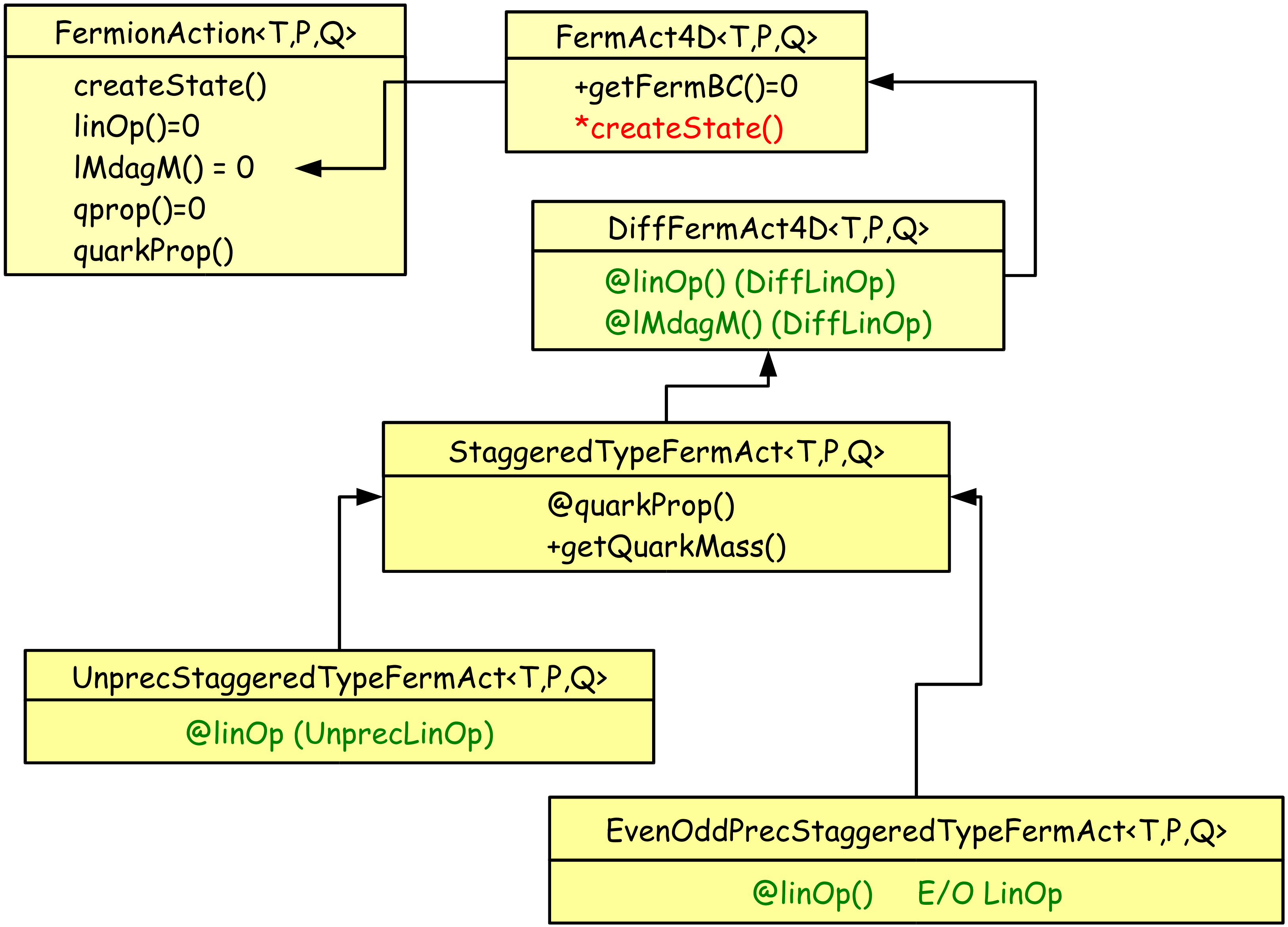
4D Derivations of Fermion Action



5D Derivations of Fermion Action



Staggered Derivations of FermionAction

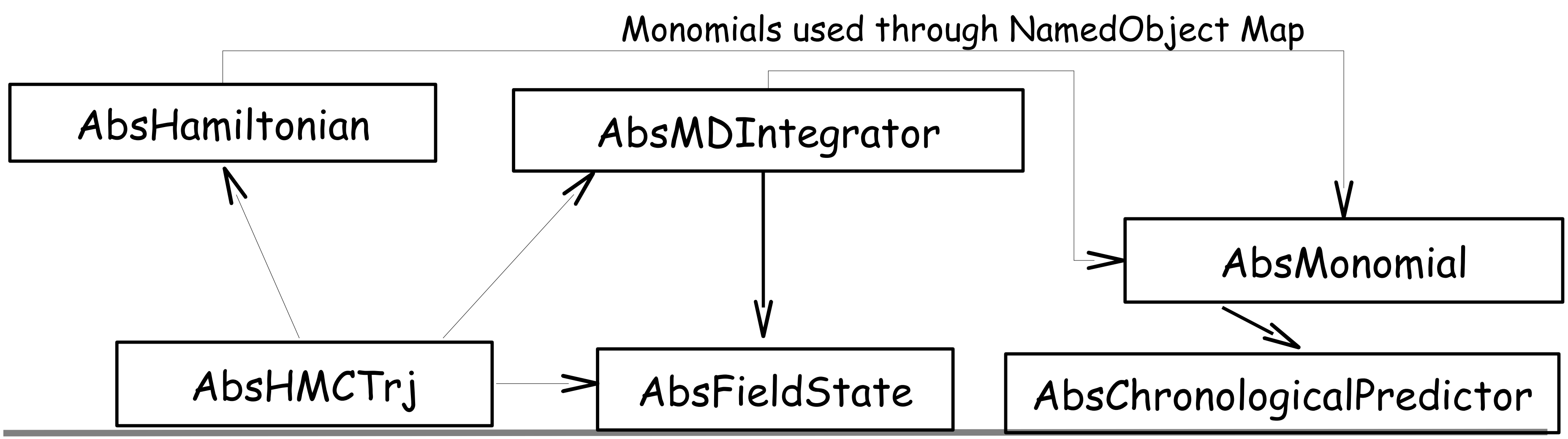


Notes on Fermion Action

- 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

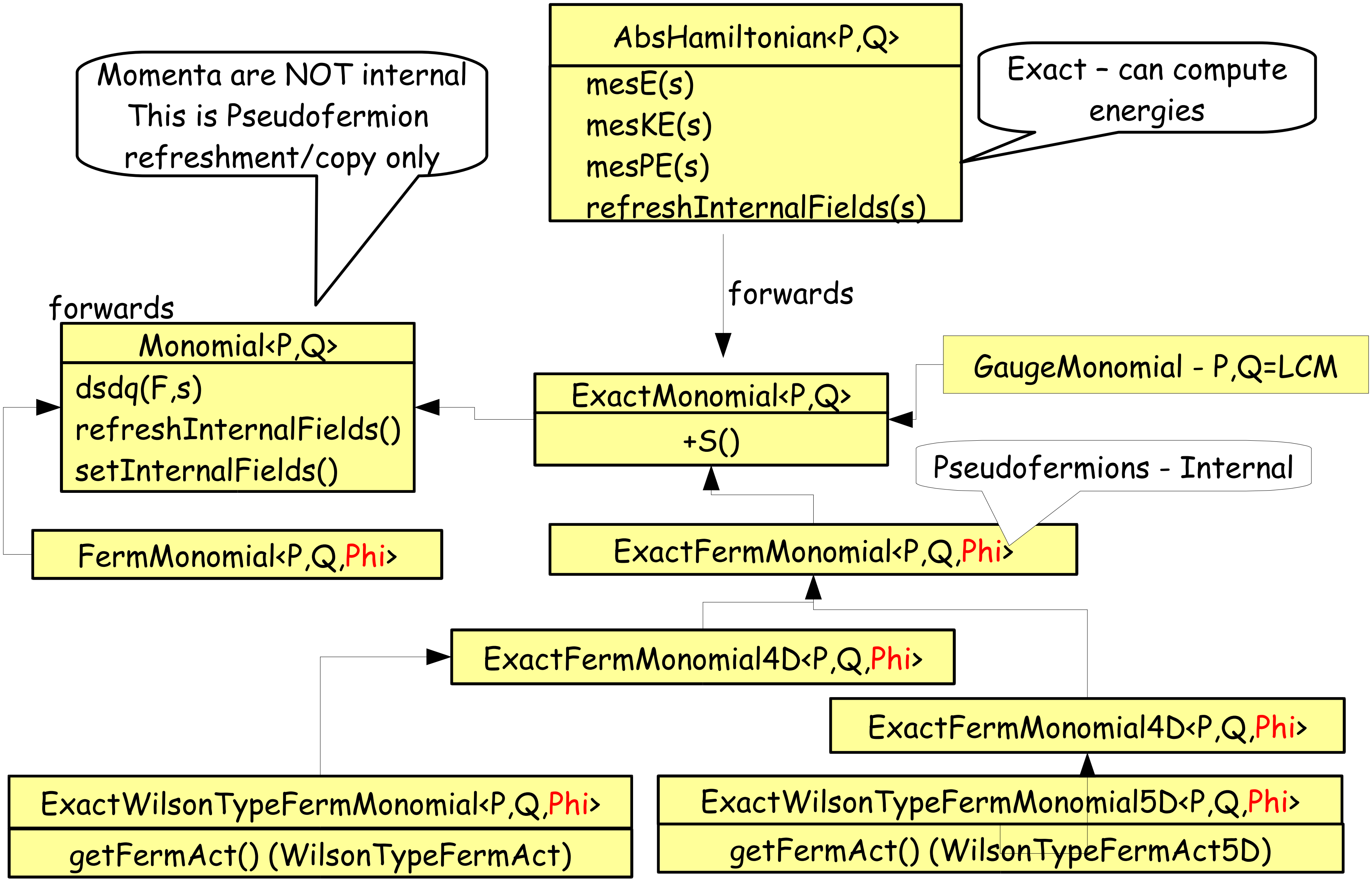
Hamiltonians and Monomials

- We evolve the Hamiltonian System

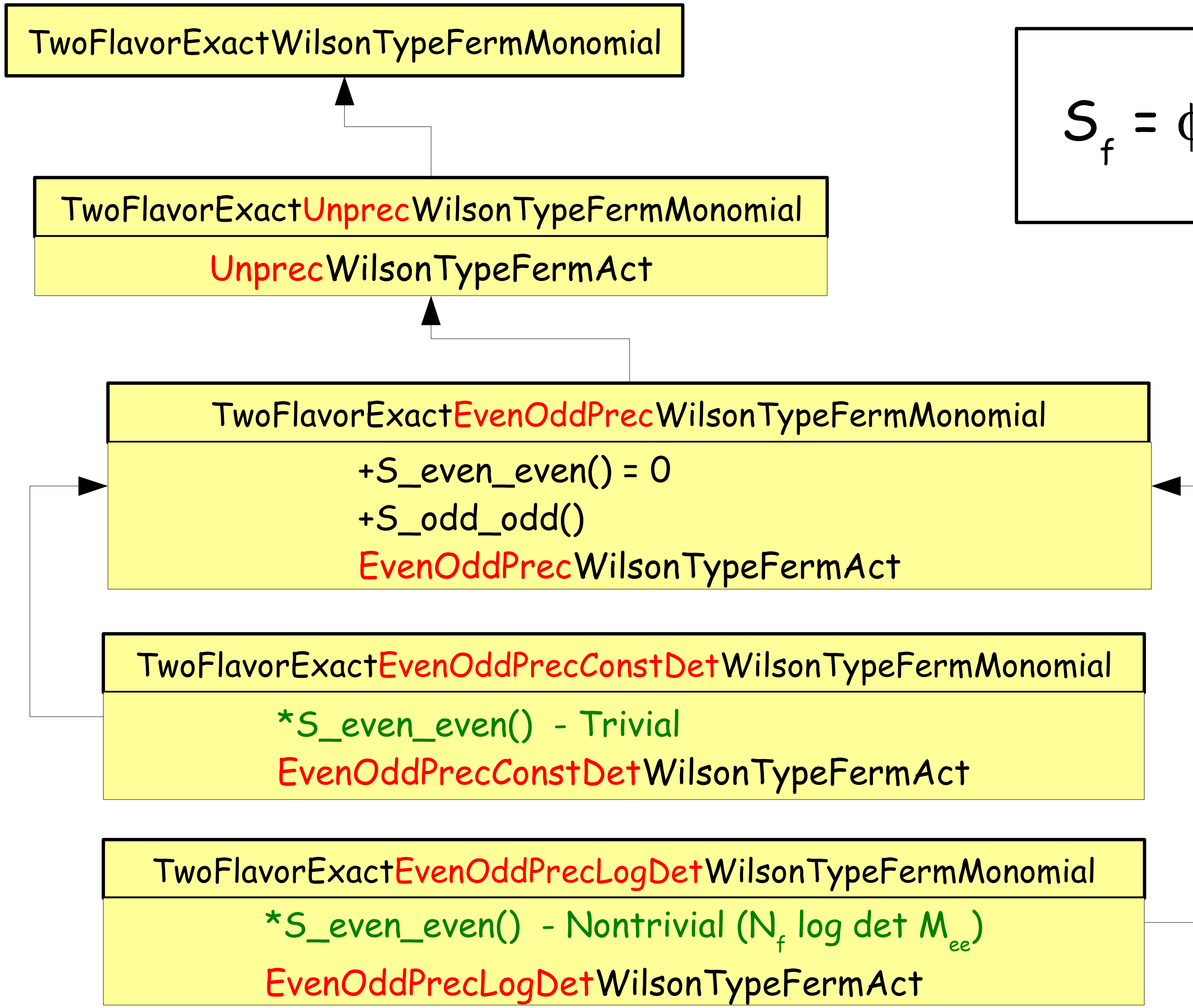
  $H(p,q) = (\frac{1}{2})\, p^2 + \sum_i S_i$

- We refer to S_i as Monomials (blame Tony!)
- In each Monomial can contribute
  - MD Force
  - Contribution to the Energy (if it is "exact")
- Monomials get created in the NamedObject store – this is referenced by Hamiltonians and MD Integrators. Hamiltonians compute energies.
- 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
- Recursive Integrators:
  - Replace leapQ() with subintegrator
  - base case: leave leapQ() in place
  - use factory to create subintegrator.

MD Integrators

- Top Level integrator:
  - knows trajectory length
  - can give back reference to the top level of recursion
- Component integrator
  - binds to list of monomials for that timescale
  - monomials live in named object store.
  - give back reference to next level integrator
    - or just a leapQ() update
- We have both 2nd and 4th order integrators of various kinds.

"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

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

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
  - SystemSolver
  - Boundary Conditions
  - ConnectState -s, CreateState-s
  - FermionAction-s
  - Monomials
    - Two flavour, Rational, Hasenbusch, Gauge
  - AbsIntegrator etc

Summary and Conclusion III

- Measurement Tasks
  - Data flow through Named Objects
  - Named Object I/O managed through special measurement tasks
- 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
- Stay tuned – for writing those pesky XML Files