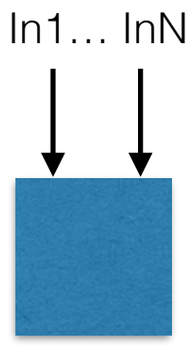
Gaudi::Functional

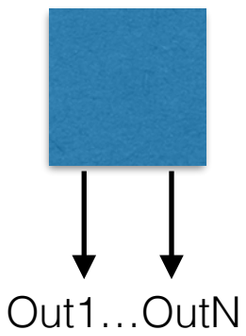
- Many algorithms are of the type ‘some data in’ → ‘some other data out’
- Standardize this pattern, and factor out ‘getting’ and ‘putting’ the *event* data

- establish a vocabulary
- less code to write,
- more uniform code — easier to understand code somebody else wrote;
- encourage (enforce!) ‘best practice’
- remove ‘boiler plate’ (aka. magic wand waving) from ‘client’ code — so future changes will not affect it! (at least, less likely they will)

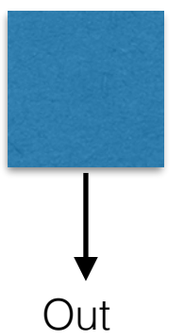
Taxonomy




```
template <typename Signature, typename Traits_ = Traits::useDefaults>
class Consumer;
.
template <typename... In, typename Traits_>
class Consumer<void(const In&…),Traits_>
{
   virtual void operator()(const In&...) const = 0;
};
```



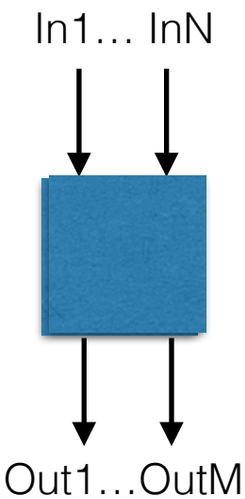

```
template <typename... Out, typename Traits_>
class Producer<std::tuple<Out...>(),Traits_>
{
   virtual std::tuple<Out...> operator()() const = 0;
}
```



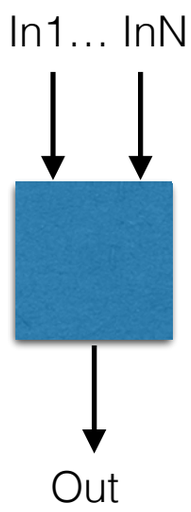

```
template <typename Out, typename Traits_>
struct Producer<Out(),Traits_>
{
    virtual Out operator()() const = 0;
};
```

Each In…, Out… corresponds to a data handle

Taxonomy




```
template <typename ... Out, typename... In, typename Traits_>
class MultiTransformer<std::tuple<Out...>(const In&...),Traits_>
{
   virtual std::tuple<Out...> operator()(const In&...) const = 0;
}
```




```
template <typename Out, typename... In, typename Traits_>
class Transformer<Out(const In&…),Traits_>
{
   virtual Out operator()(const In&...) const = 0;
};
```

Each In…, Out… corresponds to a data handle

Taxonomy

vector<“const In&”> (*)

Out

```
// Many of ‘In’ -> ‘Out’ (#In is known at initialization time)
template <typename Out, typename In, typename Traits_>
class MergingTransformer<Out(const vector_of_const_<In>&),Traits_>
{
   virtual Out operator()(const vector_of_const_<In>& inputs) const = 0;
}
```

In1… InN

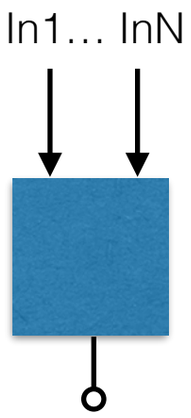
vector<Out>

```
// In1…InN -> Many of ‘Out’ (#Out is known at initialization time)
template <typename Out, typename... In, typename Traits_>
class SplittingTransformer<vector_of_<Out>(const In&…),Traits_>
{
      virtual std::vector<Out> operator()(const In&... ) const = 0
}
```

Each In…, Out… corresponds to a data handle

(*) behaves as either vector<std::cref<In>> or vector<const In*>, depending on std::is_pointer_v<In> — to (dis)allow absent inputs

Taxonomy



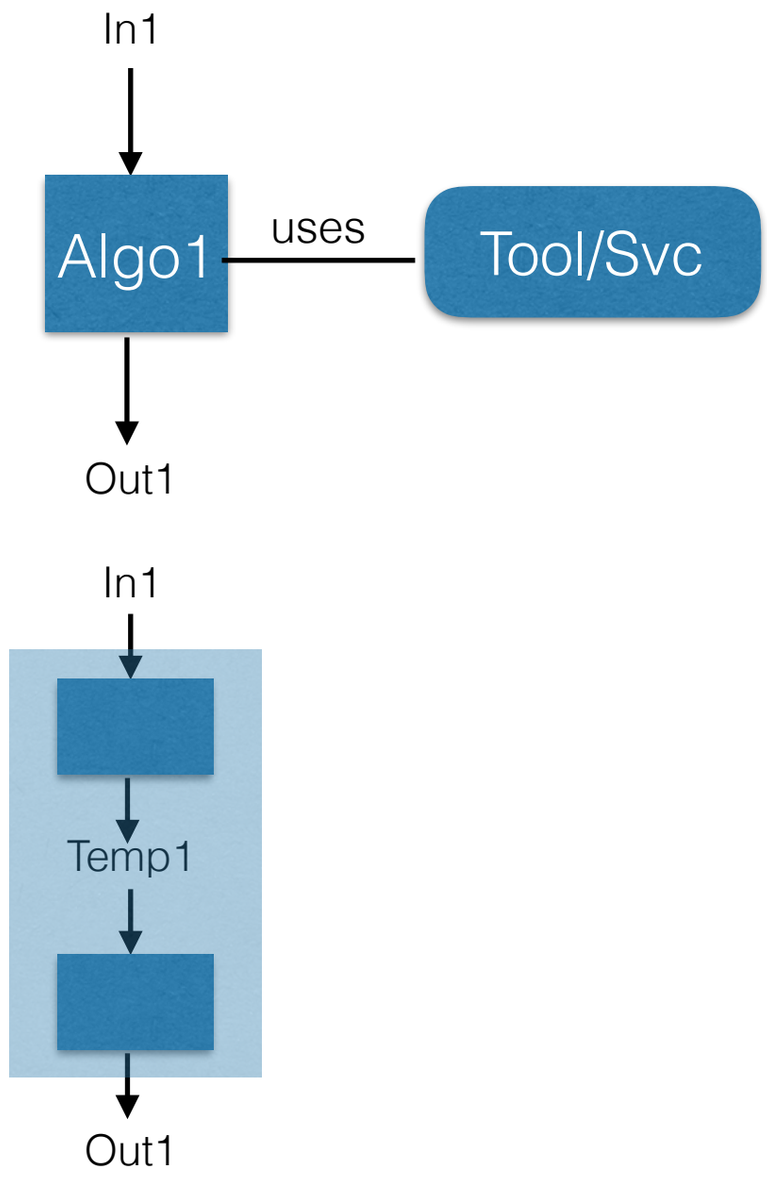

```
template <typename... In, typename Traits>
class FilterPredicate<bool(const In&...),Traits>
{
   virtual bool operator()(const In&...) const = 0;
};
```

Revisit {I,,Gaudi}Algorithm

- Do we ever really use IAlgorithm (other than casting to Algorithm?)
- GaudiAlgorithm is bloated
  - contains (backwards compatible) functionality that we *do not want to be used anymore* eg. “get<EDMClass>(event_store_path)”
- Even Algorithm is rather large
- As algorithm implementations get smaller / more focussed, the overhead becomes worse…

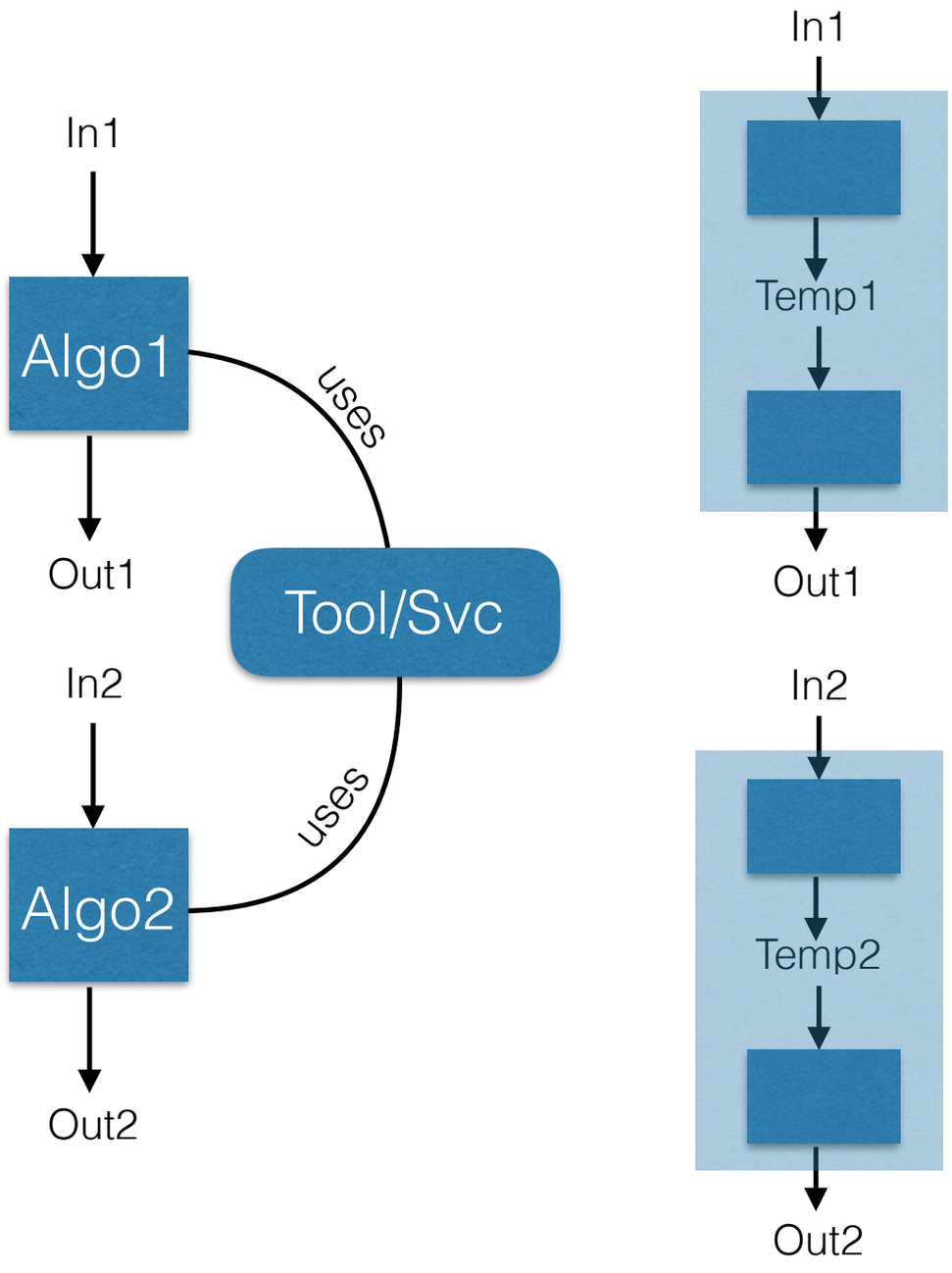
Algorithms vs. Tools

- Typical example: track fitting
  - Algorithm does ‘outer loop’ and calls a tool to do the actual fitting — one track at a time. Maybe does a bit of selection upfront, and cleanup (eg. clone killing) and monitoring afterwards.
    - Why: allows ‘easy’ change of fitting implementation — the algorithm ‘anchors’ the input and output. Tool just implements a ‘ITrackFitter’ interface.
- Coalesce into a single algorithm, remove the `ITrackFitter’ interface — a (Scalar)Transformer.
  - *Any (tool) interface which takes an event model class as input, and produces an event model class can be implemented as Transformer.*



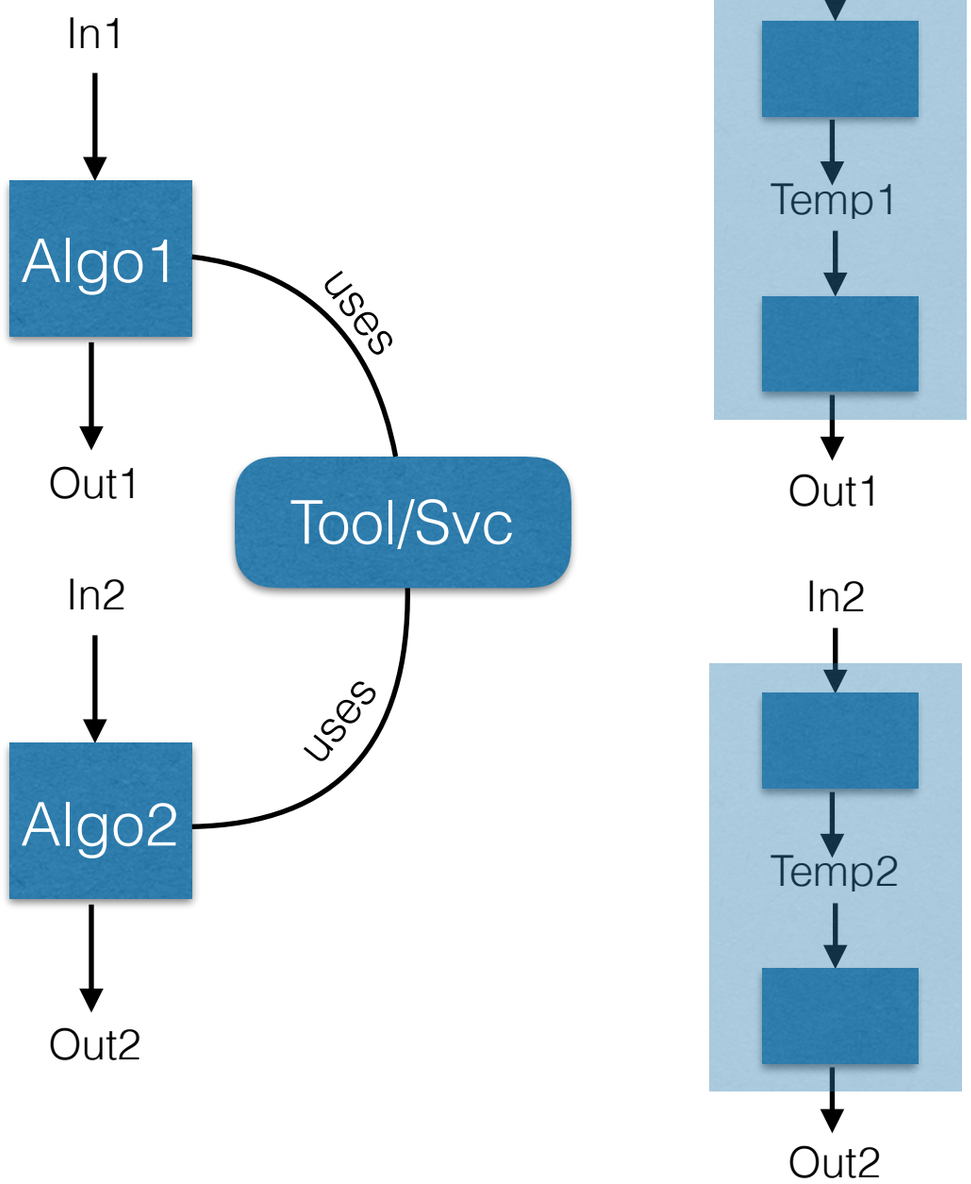
Algorithms vs. Tools

- Why was (something like) this not (already) done everywhere?
  - tool can be re-used from other places, with identical configuration
  - replacement of tool with algorithm typically requires splitting existing algorithms into (small) sequences
- may require multiple instances of algorithm, all with “almost the same” (ie. all but input/ output locations) configuration.
- may require additional (temporary!) information on event store which is only used to communicate between ‘neighbouring’ algorithms, but which now has a event-lifetime instead of algorithm scope



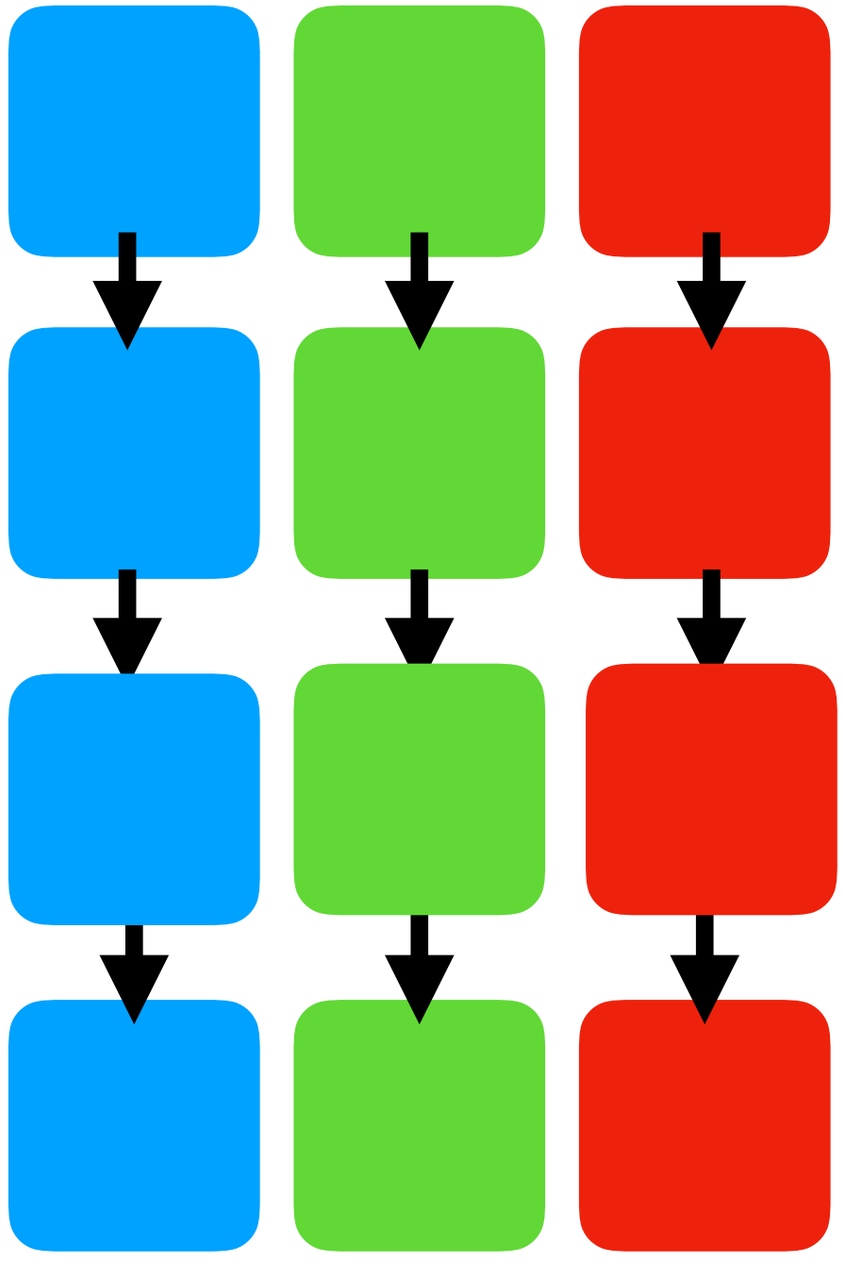
Algorithms vs. Tools

- Must have a representation of a (smallish) flow graph which replaces monolithic algorithms
  - Exists! “SuperAlgorithm”
  - Must be able to insert those flow graph “configurables” multiple times, automatically ‘wire up’ their in/output (and keep the rest of their config in sync — unless explicitly requested not to)
- `TempX` now in event store — lives (and uses memory) until ‘end of event’
  - if ‘shallow copy’ (eg. range) no real problem, otherwise …
  - …scheduler knows when it won’t be used (but that’s for next year ;-)



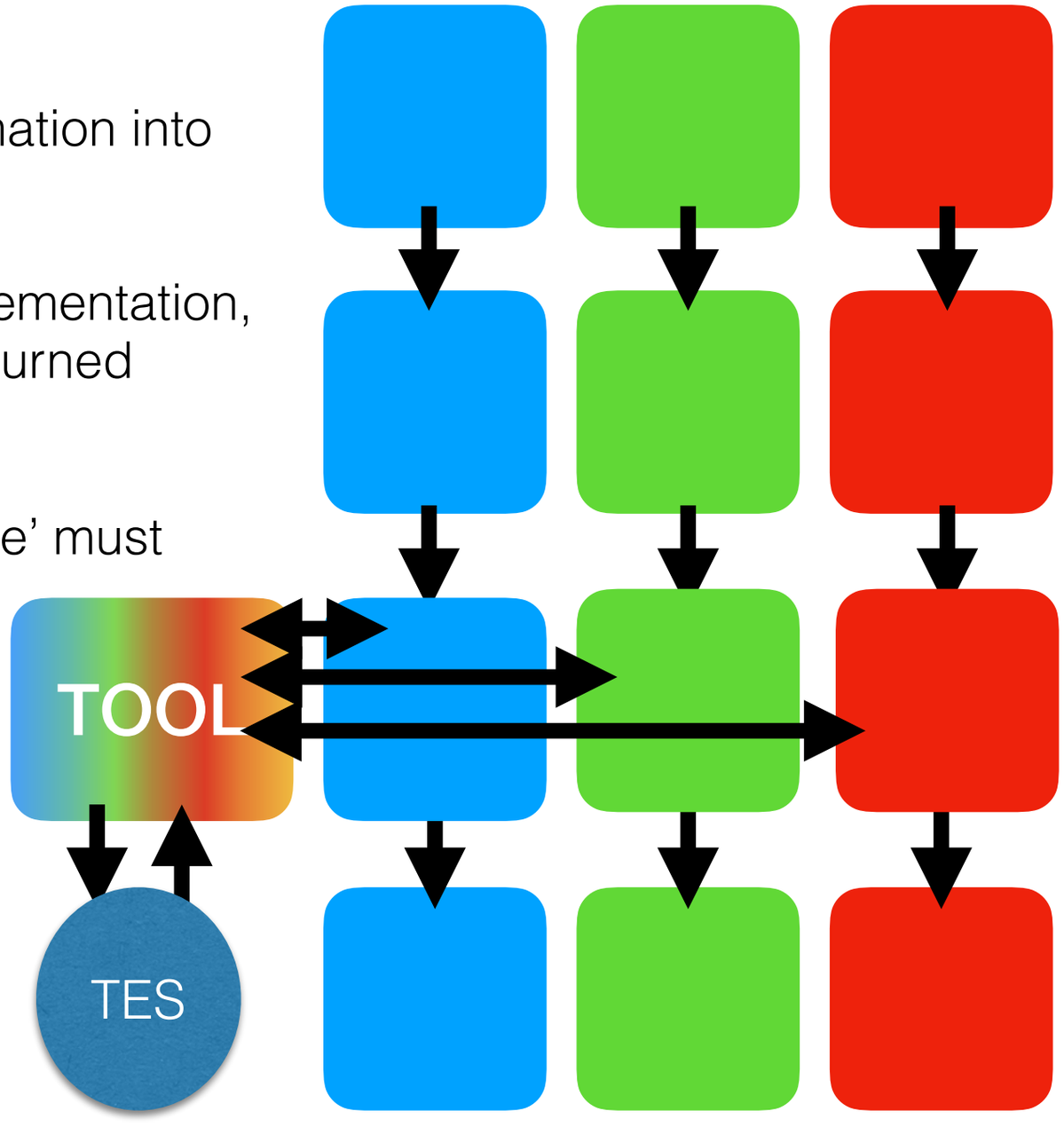
From control flow to data

- Must be able to convert ‘control flow’ information into data
- LHCb currently: dedicated sequencer implementation, HltLine, which records ‘filterPassed’ and returned ‘StatusCode’
  - Implicitly serial configuration! — one ‘line’ must finish before another starts…



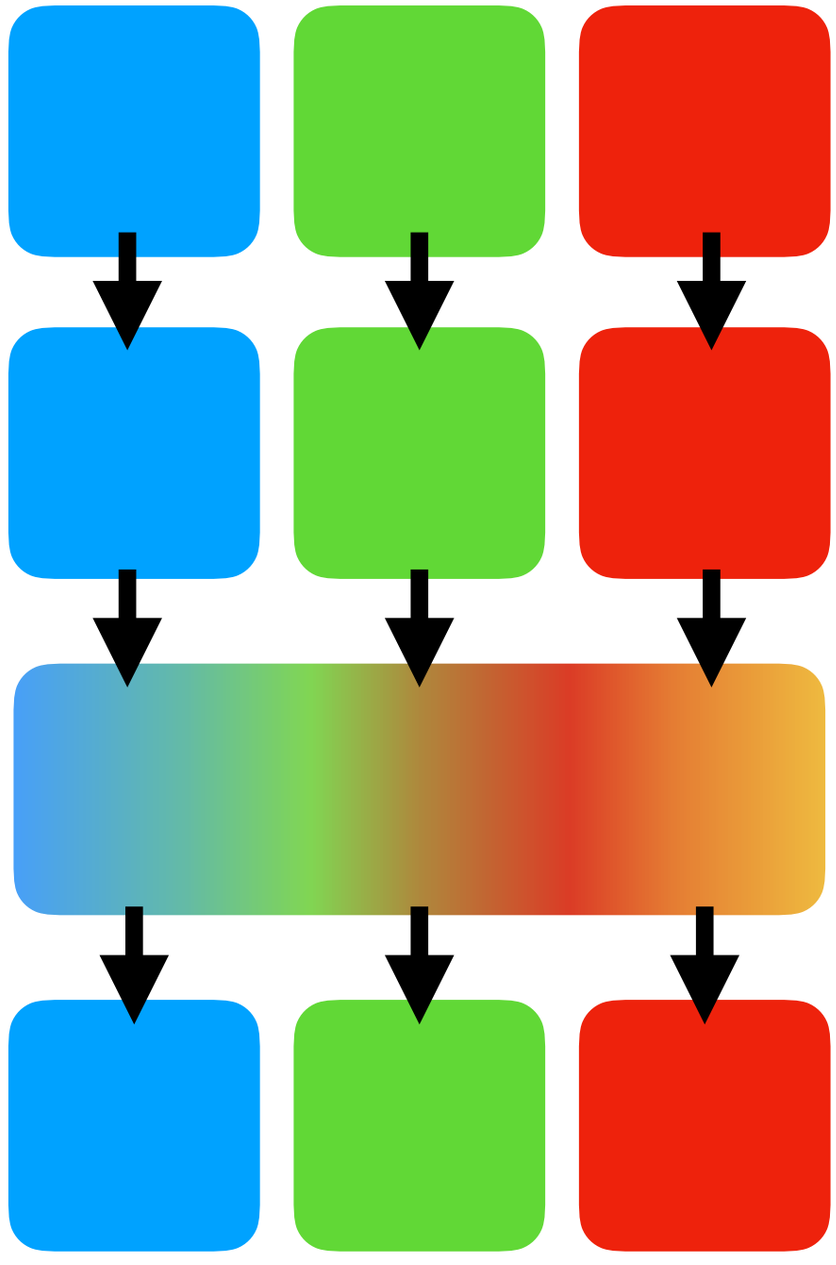
From control flow to data

- Must be able to convert ‘control flow’ information into data
- LHCb currently: dedicated sequencer implementation, HltLine, which records ‘filterPassed’ and returned ‘StatusCode’
  - Implicitly serial configuration! — one ‘line’ must finish before another starts…



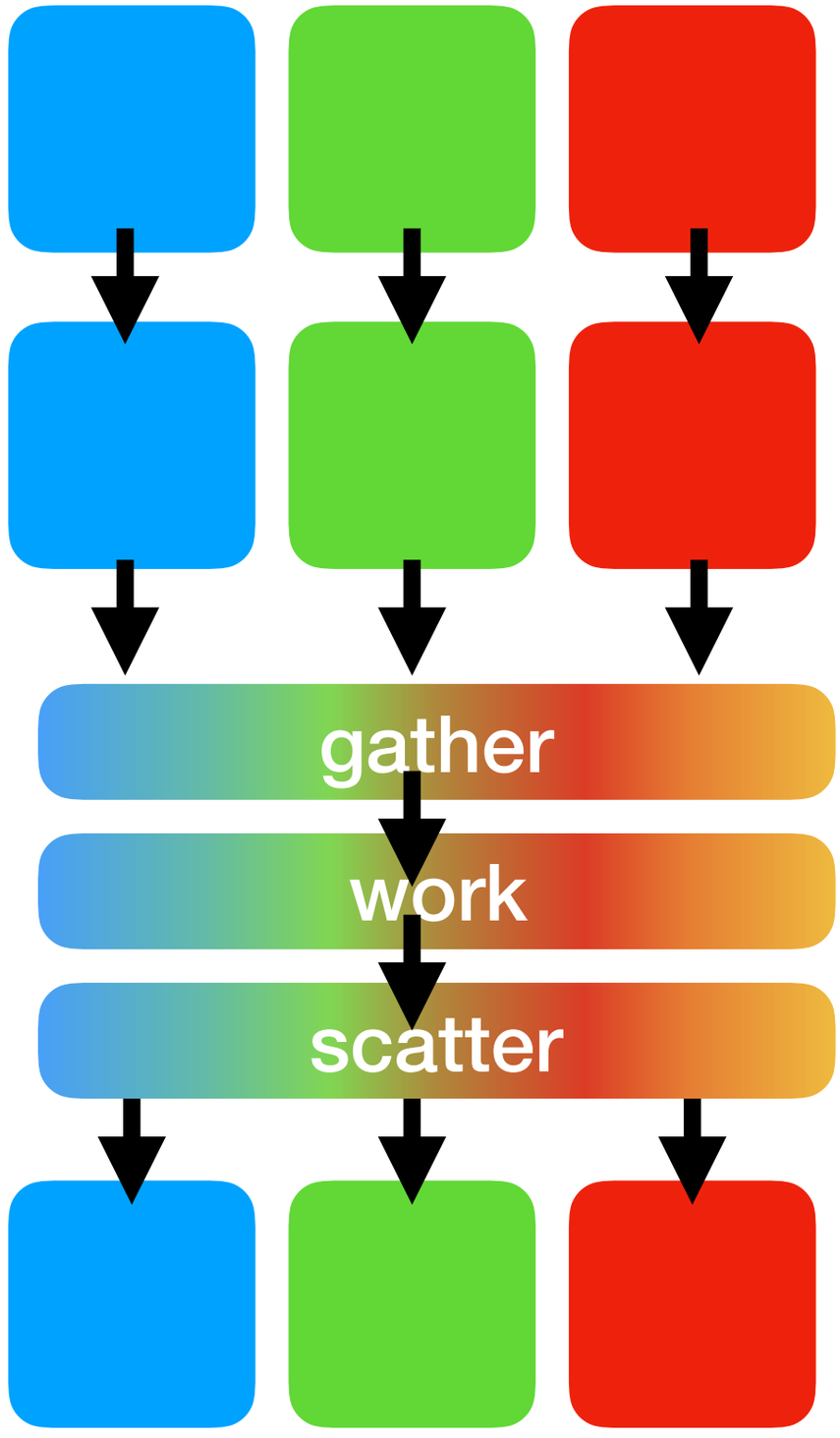
From control flow to data

- Must be able to convert ‘control flow’ information into data
- LHCb currently: dedicated sequencer implementation, HltLine, which records ‘filterPassed’ and returned ‘StatusCode’
  - Implicitly serial configuration! — one ‘line’ must finish before another starts…
- Would like to ‘suspend’ a trigger line, and allow other lines to ‘catch up’ prior to doing ‘common work’ — should be trivial with the scheduler (just data dependencies!)
  - but no (python) interface to (easily) configure this (one trigger line should not explicitly know about other trigger lines!)



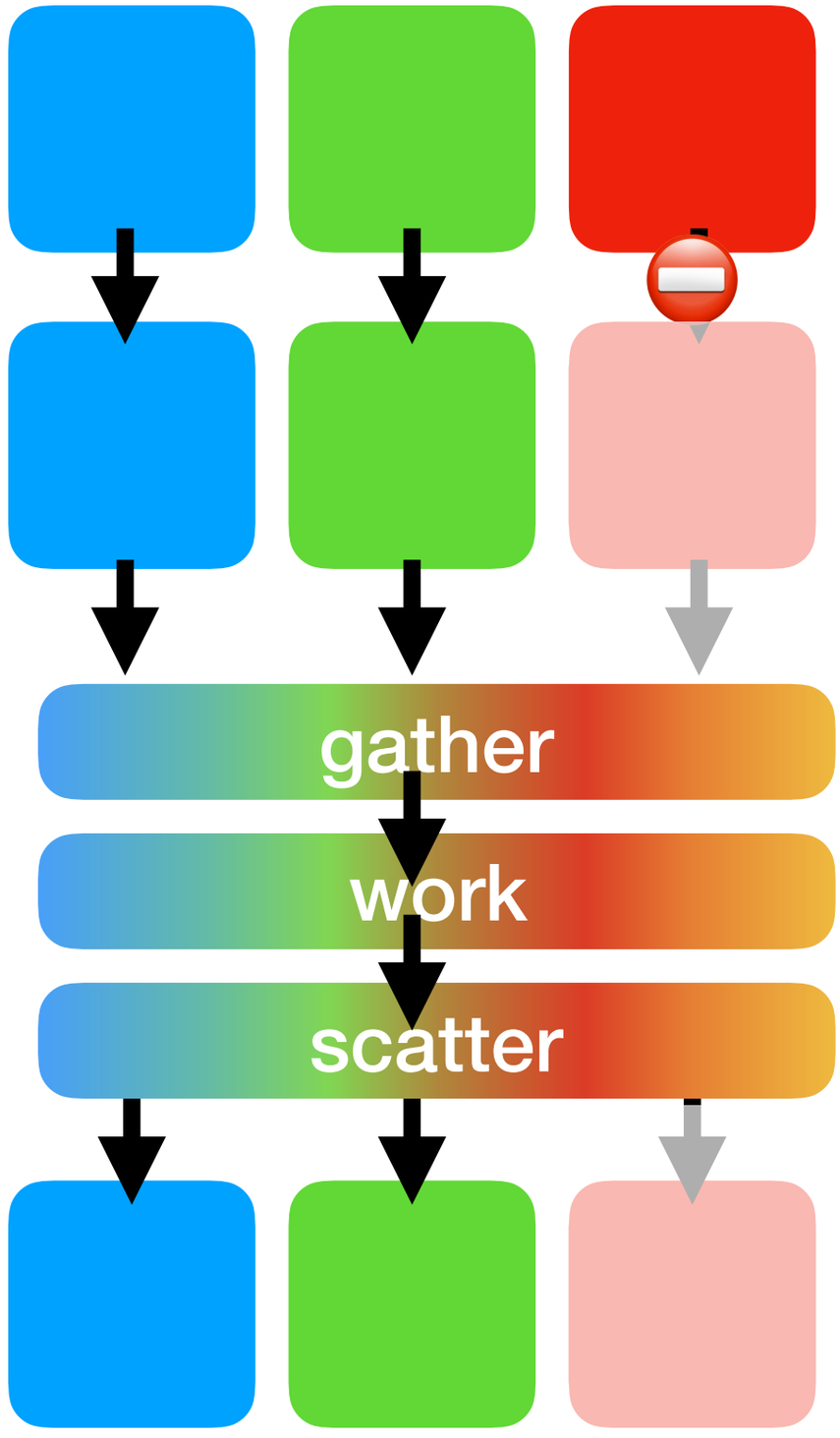
From control flow to data

- Must be able to convert ‘control flow’ information into data
- LHCb currently: dedicated sequencer implementation, HltLine, which records ‘filterPassed’ and returned ‘StatusCode’
  - Implicitly serial configuration! — one ‘line’ must finish before another starts…
- Would like to ‘suspend’ a trigger line, and allow other lines to ‘catch up’ prior to doing ‘common work’ — should be trivial with the scheduler (just data dependencies!)
  - but no (python) interface to (easily) configure this (one trigger line should not explicitly know about other trigger lines!)



From control flow to data

- Must be able to convert ‘control flow’ information into data
- LHCb currently: dedicated sequencer implementation, HltLine, which records ‘filterPassed’ and returned ‘StatusCode’
    - Implicitly serial configuration! — one ‘line’ must finish before another starts…
- Would like to ‘suspend’ a trigger line, and allow other lines to ‘catch up’ prior to doing ‘common work’ — should be trivial with the scheduler (just data dependencies!)
    - but no (python) interface to (easily) configure this (one trigger line should not explicitly know about other trigger lines!)
    - and control flow may interfere…



Tangent— if time and interest

- Wikipedia: “In computer science, **functional programming** is a programming paradigm—a style of building the structure and elements of computer programs—that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data”

- Key is ‘immutable state’ — if ‘something’ doesn’t change, it cannot cause a ‘data race’ (and thus doesn’t require locking)

- But also: using the type system — help the compiler help you

Using types for documentation

```
struct ITrajPoca : extend_interfaces<IAlgTool> {


  /// Find points along trajectories at which the distance between the
  /// trajectories is at its minimum. The precision parameter is the desired
  /// numerical accuracy of mu1 and mu2. If the restrictrange flag is true, mu
  /// is restricted to the range of the trajectory.

  virtual StatusCode minimize( const LHCb::Trajectory& traj1, double& mu1,
                               bool  restrictRange1,
                               const LHCb::Trajectory& traj2, double& mu2,
                               bool  restrictRange2,
                               Gaudi::XYZVector& distance,
                               double precision ) const = 0 ;
```

```
ITrajPoca* pocaTool = ...;
Trajectory& traj1 = ...;
Trajectory& traj2 = ...;
Gaudi::XYZVector distance;
auto sc = pocaTool->minimize( traj1, mu1, true,
                              traj2, mu2, true,
                              distance, 0.001*Gaudi::Units::mm)
```

Using types for documentation

```
struct ITrajPoca : extend_interfaces<IAlgTool> {

  /// Find points along trajectories at which the distance between the
  /// trajectories is at its minimum. The precision parameter is the desired
  /// numerical accuracy of mu1 and mu2. If the restrictrange flag is true, mu
  /// is restricted to the range of the trajectory.

  virtual StatusCode minimize( const LHCb::Trajectory& traj1, double& mu1,
                               bool  restrictRange1,
                               const LHCb::Trajectory& traj2, double& mu2,
                               bool  restrictRange2,
                               Gaudi::XYZVector& distance,
                               double precision ) const = 0 ;
```

```
ITrajPoca* pocaTool = ...;
Trajectory& traj1 = ...;
Trajectory& traj2 = ...;
Gaudi::XYZVector distance;
auto sc = pocaTool->minimize( traj1, mu1, true,
                              traj2, mu2, true,
                              distance, 0.001*Gaudi::Units::mm)
```

```
struct ITrajPoca : extend_interfaces<IAlgTool> {

  // tagged_bool is (a slightly modified version of) this code
  using RestrictRange = xplicit::tagged_bool<struct RestrictRange_tag>;

  /// Find points along trajectories at which the distance between the
  /// trajectories is at its minimum. The precision parameter is the desired
  /// numerical accuracy of mu1 and mu2. If the restrictrange flag is
  /// RestrictToRange{true} , mu is restricted to the range of the trajectory.

  virtual StatusCode minimize( const LHCb::Trajectory& traj1, double& mu1,
                               RestrictRange  restrictRange1,
                               const LHCb::Trajectory& traj2, double& mu2,
                               RestrictRange  restrictRange2,
                               Gaudi::XYZVector& distance,
                               double precision ) const = 0 ;
```

```
ITrajPoca* pocaTool = ...;
Trajectory& traj1 = ...;
Trajectory& traj2 = ...;
Gaudi::XYZVector distance;
auto sc = pocaTool->minimize( traj1, mu1, ITrajPoca::RestrictToRange{true},
                              traj2, mu2, ITrajPoca::RestrictToRange{true},
                              distance, 0.001*Gaudi::Units::mm)
```

Using types to constrain

```
struct Partition {
   SmartIF<IDataProviderSvc> dataProvider;
   SmartIF<IDataManagerSvc> dataManager;
   wbMutex storeMutex;
   DataObjIDColl newDataObjects;
   int eventNumber;

   Partition();
   Partition( IDataProviderSvc* dp, IDataManagerSvc* dm )
   Partition( const Partition& entry )
   Partition& operator=( const Partition& entry )
};

std::vector<Partition>  m_partitions;

TTHREAD_TLS( Partition* ) s_current( 0 );
```

```
// load all preload items of the list
  StatusCode preLoad()
  {
    wbMutex::scoped_lock lock;
    lock.acquire( s_current->storeMutex );
    StatusCode sc = s_current->dataProvider->preLoad();
    DataAgent da( s_current->newDataObjects );
    s_current->dataManager->traverseTree( &da );
    return sc;
  }
```

```
/// Add an item to the preload list
StatusCode addPreLoadItem( const DataStoreItem& item )
{
  for ( auto& p : m_partitions ) {
        p.dataProvider->addPreLoadItem( item );
  }
  return StatusCode::SUCCESS;
}
```

Using types to constrain

```
//trivial version of facebooks' Synchronized
template <typename T, typename Mutex = tbb::mutex>
class Synced {
   T m_obj;
   mutable Mutex m_mtx;
public:
   template <typename F> auto with_lock(F&& f) -> decltype(auto)
   { typename Mutex::scoped_lock lock; lock.acquire( m_mtx ); return f(m_obj); }
   template <typename F> auto with_lock(F&& f) const -> decltype(auto)
   { typename Mutex::scoped_lock lock; lock.acquire( m_mtx ); return f(m_obj); }
};
```

```
struct Partition final {
   SmartIF<IDataProviderSvc> dataProvider;
   SmartIF<IDataManagerSvc> dataManager;
   DataObjIDColl newDataObjects;
   int eventNumber = -1;
};
/// Datastore partitions
std::vector<Synced<Partition>> m_partitions;

TTHREAD_TLS( Synced<Partition>* ) s_current = nullptr;
```

```
/// load all preload items of the list
  StatusCode preLoad() override
  {
    return s_current->with_lock( [](Partition& p) {
       StatusCode sc = p.dataProvider->preLoad();
       DataAgent da( p.newDataObjects );
       p.dataManager->traverseTree( &da );
       return sc;
    } );
  }
```

Using types to constrain

//trivial version of facebooks' Synchronized

```
template <typename T, typename Mutex = tbb::mutex>
class Synced {
   T m_obj;
   mutable Mutex m_mtx;
public:
   template <typename F> auto with_lock(F&& f) -> decltype(auto)
   { typename Mutex::scoped_lock lock; lock.acquire( m_mtx ); return f(m_obj); }
   template <typename F> auto with_lock(F&& f) const -> decltype(auto)
   { typename Mutex::scoped_lock lock; lock.acquire( m_mtx ); return f(m_obj); }
};
```

```
struct Partition final {
   SmartIF<IDataProviderSvc> dataProvider;
   SmartIF<IDataManagerSvc> dataManager;
   DataObjIDColl newDataObjects;
   int eventNumber = -1;
};
/// Datastore partitions
std::vector<Synced<Partition>> m_partitions;

TTHREAD_TLS( Synced<Partition>* ) s_current = nullptr;
```

```
// transform an f(T) into an f(Synced<T>)
template <typename Fun> auto with_lock(Fun&& f) {
   return [f=std::forward<Fun>(f)]
         (auto& p) -> decltype(auto)
         { return p.with_lock( f ); };
}
// call f(T) for each element in a container of Synced<T>
template <typename ContainerOfSynced, typename Fun>
void for_(ContainerOfSynced& c, Fun&& f) {
  std::for_each(begin(c),end(c),with_lock(std::forward<Fun>(f)));
}
```

```
/// load all preload items of the list
  StatusCode preLoad() override
  {
    return s_current->with_lock( [](Partition& p) {
       StatusCode sc = p.dataProvider->preLoad();
       DataAgent da( p.newDataObjects );
       p.dataManager->traverseTree( &da );
       return sc;
    } );
  }
```

```
StatusCode addPreLoadItem( const DataStoreItem& item )
  {
    for_(m_partitions, [&](Partition& p) {
          p.dataProvider->addPreLoadItem( item );
    });
    return StatusCode::SUCCESS;
  }
```

Example: RawEvent→FTLiteClusters

templated on ‘signature’: Out(const In&…)

```
struct FTRawBankDecoder
: Gaudi::Functional::Transformer<FTLiteClusters(const LHCb::RawEvent& )>
{

  FTRawBankDecoder( const std::string& name, ISvcLocator* pSvcLocator )
  : Transformer(name , pSvcLocator,
                KeyValue{ “RawEventLocations",
                          concat_alternatives( LHCb::RawEventLocation::Other,
                                               LHCb::RawEventLocation::Default )},
                KeyValue{ "OutputLocation", LHCb::FTLiteClusterLocation::Default } )
  { }
```

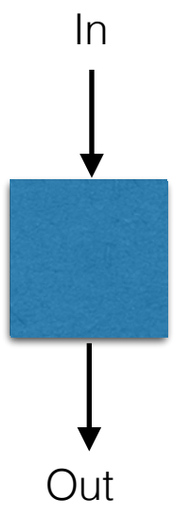
Transformer generates DataObjectHandle<RawEvent>, DataObjectHandle<FTLiteClusters>, and properties “RawEventLocation” and “OutputLocation”

```
  FTLiteClusters operator()(const LHCb::RawEvent& rawEvent) const override
  {
     FTLiteClusters clusters;
     // create clusters from RawEvent
     return clusters;
  }
};
```

Transformer::execute ‘gets’ RawEvent using data handle, and passes it as ‘const RawEvent&’ to this ‘*const*’ method

and ‘moves’ FTLiteClusters into DataHandle (actually, RVO kicks in, and the move is elided)

ScalarTransformer




```
#include “GaudiAlg/Transformer.h"

namespace Gaudi { namespace Functional {

// Scalar->Vector adapted 1->1 algorithm
template <typename ScalarOp,
          typename TransformerSignature,
          typename Traits_ = Traits::useDefaults> class ScalarTransformer;

template <typename ScalarOp, typename Out, typename In, typename Traits_>
class ScalarTransformer<ScalarOp,Out(const In&),Traits_> : public Transformer<Out(const In&),Traits_>
{
    const ScalarOp& scalarOp() const { return static_cast<const ScalarOp&>(*this); }

public:

    using Transformer<Out(const In&)>::Transformer;
    Out operator()(const In& in) const final {
        Out out; out.reserve(in.size());
        auto& scalar = scalarOp();
        for (const auto& i : in) details::insert( out, scalar( details::deref(i)) );
        details::apply( scalar, out );
        return out;
    }
};

}}
```

- Why: less ‘boilerplate’ code, *framework* does the loop for you, and (eventually) ‘chunk’ the loop data, and dispatch ‘chunks’ to tasks. (think eg “tbb::parallel_for”)

ScalarTransformer

‘curious recurring template pattern’

```
struct TTrackFromLong
: Gaudi::Functional::ScalarTransformer<TTrackFromLong, LHCb::Tracks(const LHCb::Tracks&)>
{
  TTrackFromLong(const std::string& name, ISvcLocator* pSvcLocator):
    ScalarTransformer( name, pSvcLocator,
                       KeyValue( "inputLocation", LHCb::TrackLocation::Forward),
                       KeyValue( "outputLocation", LHCb::TrackLocation::Seed)  )
  { }
  using ScalarTransformer::operator();
  boost::optional<LHCb::Track> operator()(const LHCb::Track& trk) const
  {
     auto count = std::count_if( begin(trk.lhcbIDs()), end(trk.lhcbIDs()), isT );
     if (count<5) return boost::none;
     LHCb::Track seed;
     // do stuff with trk to fill seed - not shown here
     return seed;
  };
};
```

Example

```
template <typename... In, typename Traits_>
class Consumer<void(const In&…),Traits_>
{
   virtual void operator()(const In&...) const = 0;
};
```

Consumer templated on
‘signature’ void(const In&…)

```
class EventTimeMonitor
: public Gaudi::Functional::Consumer<void(const LHCb::ODIN&),
                                     Gaudi::Functional::Traits::BaseClass_t<GaudiHistoAlg>>
{
  EventTimeMonitor( const std::string& name, ISvcLocator* pSvcLocator)
  : Consumer( name , pSvcLocator,
              KeyValue{"Input", LHCb::ODINLocation::Default } )
  {
    …
  }
  void EventTimeMonitor::operator()(const LHCb::ODIN& odin) const override
  {
    …   code using ‘odin’ goes here …
    …   yes, you can fill histograms here! …
  }
```

“Traits” allow customization,
eg. which base class to use.
Default: GaudiAlgorithm

baseclass generates
DataObjectHandle<ODIN>,
and matching property “Input”
with (initial) value as specified

baseclass’ execute ‘gets’ ODIN using data handle,
and passes it as ‘const ODIN&’ to this ‘const’ method

(Multi)Transformer

- Given N input (containers) (of varying type) produce 1 output (container)

```
template <typename Out, typename... In, typename Traits_>
class Transformer<Out(const In&…),Traits_>
{
      virtual Out operator()(const In&...) const = 0;
};
```

const In1&, …, const InN& —> Out

- Given N inputs (containers) (of varying type) produce (a tuple of) M output (containers)

```
template <typename ... Out, typename... In, typename Traits_>
class MultiTransformer<std::tuple<Out...>(const In&...),Traits_>
{
       virtual std::tuple<Out...> operator()(const In&...) const = 0;
};
```

const In1&, …, const InN& —> Out1, …, OutM

Notes:

1. In1..InN, Out1…OutM can all be *different* types, but the signature must be known *at compile time*
2. Inputs are passed (again) by const&; outputs by value, and are *moved* (not copied) into event store; and the operator() is const

Multi(Transformer)

- HCRawBankDecoder: 2 inputs -> 2 outputs
- (RawEvent, ODIN) —> (HCDigits, HCDigits)

```
class HCRawBankDecoder
: public Gaudi::Functional::MultiTransformer<
             std::tuple<LHCb::HCDigits,LHCb::HCDigits>(const LHCb::RawEvent&, const LHCb::ODIN&),
             Gaudi::Functional::Traits::BaseClass_t<GaudiHistoAlg> >
{
    HCRawBankDecoder(const std::string& name, ISvcLocator* pSvcLocator)
    : MultiTransformer( name, pSvcLocator,
                { KeyValue{ “RawEventLocations",
                            Gaudi::Functional::concat_alternatives( LHCb::RawEventLocation::HC,
                                                                    LHCb::RawEventLocation::Default,
                                                                    LHCb::RawEventLocation::Other ) },
                  KeyValue{ “OdinLocation",     LHCb::ODINLocation::Default } },
                { KeyValue{ "DigitLocation",    LHCb::HCDigitLocation::Default },
                  KeyValue{ "L0DigitLocation",  LHCb::HCDigitLocation::L0 } } )
    {
       …  other properties declared here  …
    }
    std::tuple<LHCb::HCDigits,LHCb::HCDigits>
    operator()(const LHCb::RawEvent& raw, const LHCb::ODIN& odin) const override
    {
        std::tuple<LHCb::HCDigits,LHCb::HCDigits> output;
          … code to “fill” output from ‘raw’ and ‘odin’ goes here…
        return output;
    }
};
```

c’tor requires 2 ‘KeyValue’ arrays of 2 elements each so it can create the 4 properties needed to specify input/output locations

“search path” gets resolved on first access

{Splitting,Merging}Transformer

- Merging Transformer: N containers of the same -> 1 container
- Splitting Transformer: 1 container -> N containers of the same
- N unknown at compile time, set at configuration time (think “vector<Container>”)

- MergingTransformer: Calo/CaloPIDs: InCaloAcceptanceAlg, Tr/TrackUtils: TrackListMerger,
- SplittingTransformer: Hlt/HltDAQ: HltRawBankDecoderBase

- Note: input *containers* can either be ‘mandatory’ — require a vector of references to const container — or ‘optional’ — require a vector of pointers to const containers as ‘signature’

Anti-Pattern: “ping-pong” calls

- Ghost tool interface:

```
/** consider this the beginning of a new event **/
virtual StatusCode beginEvent() = 0;
/** Add the reference information **/
virtual StatusCode execute(LHCb::Track& aTrack) const = 0;
```

  - to use, *first* call ‘beginEvent’ — tool fetches some event data
  - *then* separate calls for each track of interest, and then ‘modifies’ track (i.e. adds extraInfo field)

- Suggested solution:

```
/** Compute ghost information for a list of tracks **/
virtual std::vector<float> execute(const Tracks& listOfTracks) const = 0;
```

  - call with a list (range) of tracks, return list (range) of ghost values, always fetch event data