Streamulus

A language for real-time event stream processing

Irit Katriel

MADALGO Seminar
Århus, 14 June 2012

Event Stream

An infinite, ordered sequence of discrete elements

Event Stream Processing

A stream arrives as a sequence of calls to a **HandleEvent** function

HandleEvent( **)**

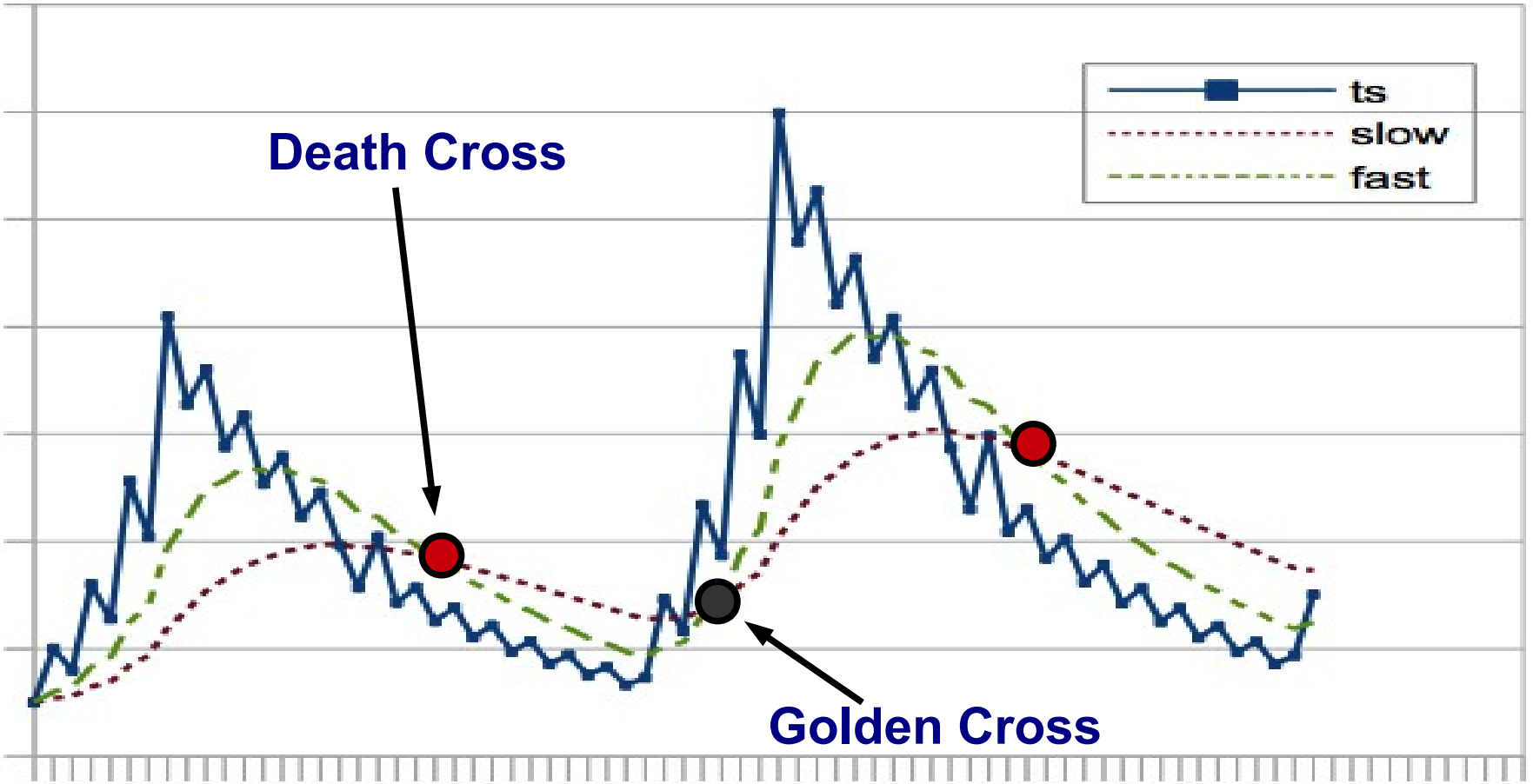
HandleEvent( **)**

HandleEvent( **)**

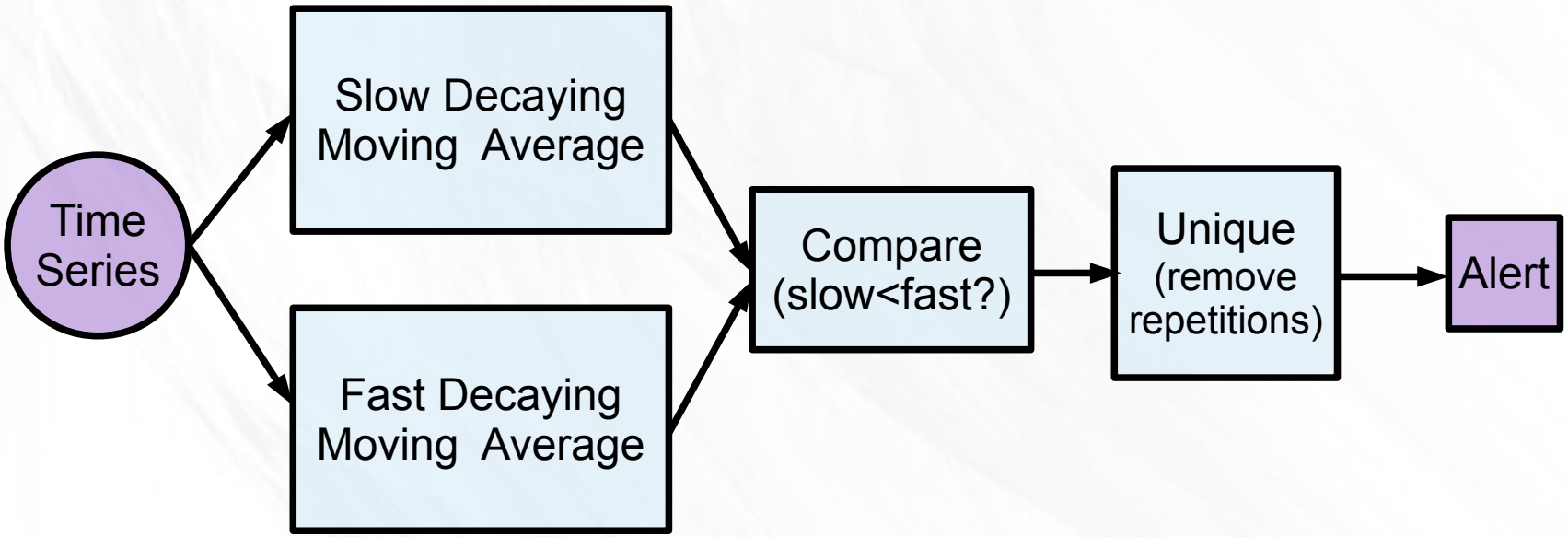
We need to reason about the forest

*We focus on **how to write programs** (not on algorithms)*

Motivating Example: Crossings of Moving Averages



Cross Detection



Let's implement this object-orientedly...

Moving Average

```
template<int DecayFactor>
class Mavg {
    ...
    double Tick(value) {
        double alpha = 1-exp(-DecayFactor*(now-prev_time));
        prev_time = now;
        return mavg = alpha*value + (1-alpha)*mavg;
    }
    double Get()  {
        return mavg;
    }

    double mavg;
    clock_t prev_time;
};
```

Cross Detection Class

```
class CrossDetection {

    ....

    void Tick(value) {
        bool comp = (slow.Tick(value) < fast.Tick(value));
        if (comp != prev_comp)
                IssueCrossingAlert(comp);
        prev_comp = comp;
    }

    Mavg<1>   slow;
    Mavg<10> fast;
    bool prev_comp;

};
```

Cross Detection Class

```
class CrossDetection {

    ….

    void Tick(value) {
        bool comp = (slow.Tick(value) < fast.Tick(value));
        if (comp != prev_comp)
                IssueCrossingAlert(comp);
        prev_comp = comp;
    }

    Mavg<1>   slow;
    Mavg<10> fast;
    bool prev_comp;

};
```

What if the moving averages are also needed elsewhere?

Refactored Cross Detection Class

```
class CrossDetection {

   CrossDetection(Mavg<1>& slow_, Mavg<10>& fast_)
      : slow(slow_), fast(fast_)   { }

   void UpdateValue() {
         bool comp = (slow.Get() < fast.Get());
         if (comp != prev_comp)
               IssueCrossingAlert(comp);
         prev_comp = comp;
   }

   Mavg<1>&   slow;
   Mavg<10>& fast;
   bool prev_comp;
};
```

Construct mavgs elsewhere and pass in references.

Update mavgs elsewhere. Here only probe.

Using the Refactored Class

```
Mavg<10>          fast_mavg;
Mavg<1>           slow_mavg;
CrossDetection    cross_detection(slow_mavg, fast_mavg);
SomethingElse     something_else(slow_mavg, fast_mavg);
```

setup

Using the Refactored Class

```
Mavg<10>          fast_mavg;
Mavg<1>           slow_mavg;
CrossDetection    cross_detection(slow_mavg, fast_mavg);
SomethingElse     something_else(slow_mavg, fast_mavg);
```

setup

```
HandleEvent(double value) {
    slow_mavg.Tick(value);
    fast_mavg.Tick(value);
    cross_detection.UpdateValue(); // implicit data
    something_else.UpdateValue(); // dependencies
}
```

process an event

This was noticed before

From “The 8 requirements of real-time stream processing”, Stonebraker, Çetintemel, Zdonik. SIGMOD Record, 2005:

> *"Historically, for streaming applications, general purpose languages such as C++ or Java have been used as the workhorse development and programming tools. Unfortunately, relying on low-level programming schemes results in long development cycles and high maintenance costs."*

And they conclude with the requirement:

> *"Query using StreamSQL"*

This was noticed before

From “The 8 requirements of real-time stream processing”, Stonebraker, Çetintemel, Zdonik. SIGMOD Record, 2005:

> *"Historically, for streaming applications, general purpose languages such as C++ or Java have been used as the workhorse development and programming tools. Unfortunately, relying on low-level programming schemes results in long development cycles and high maintenance costs."*

And they conclude with the requirement, where they probably meant:

> *"Query using ~~StreamSQL~~"*

A Domain-Specific Langauge

StreamSQL

```
SELECT avg(some_column) as AvgValue
FROM input [rows 20]
WHERE some_condition
GROUP BY another_column
```

Sliding window. Last 20 entries.

Can include user-defined operators

- StreamBase
- Esper
- Sybase Aleri
- Microsoft StreamInsight
- ...

Returning to Our Problem

```
Mavg<10>          fast_mavg;
Mavg<1>           slow_mavg;
CrossDetection    cross_detection(slow_mavg, fast_mavg);
SomethingElse     something_else(slow_mavg, fast_mavg);
```

setup


```
HandleEvent(double value) {
    slow_mavg.Tick(value);
    fast_mavg.Tick(value);
    cross_detection.UpdateValue(); // implicit data
    something_else.UpdateValue(); // dependencies
}
```

process an event

The Streamulus Way

```
InputStreamT  ts = NewInputStream<double>("TS");
SubscriptionT slow = Subscribe<double>(Mavg<1>(ts));
SubscriptionT fast = Subscribe<double>(Mavg<10>(ts));
Subscribe( cross_alert( unique( slow < fast ) ) );
Subscribe( something_else(slow,fast) );
```

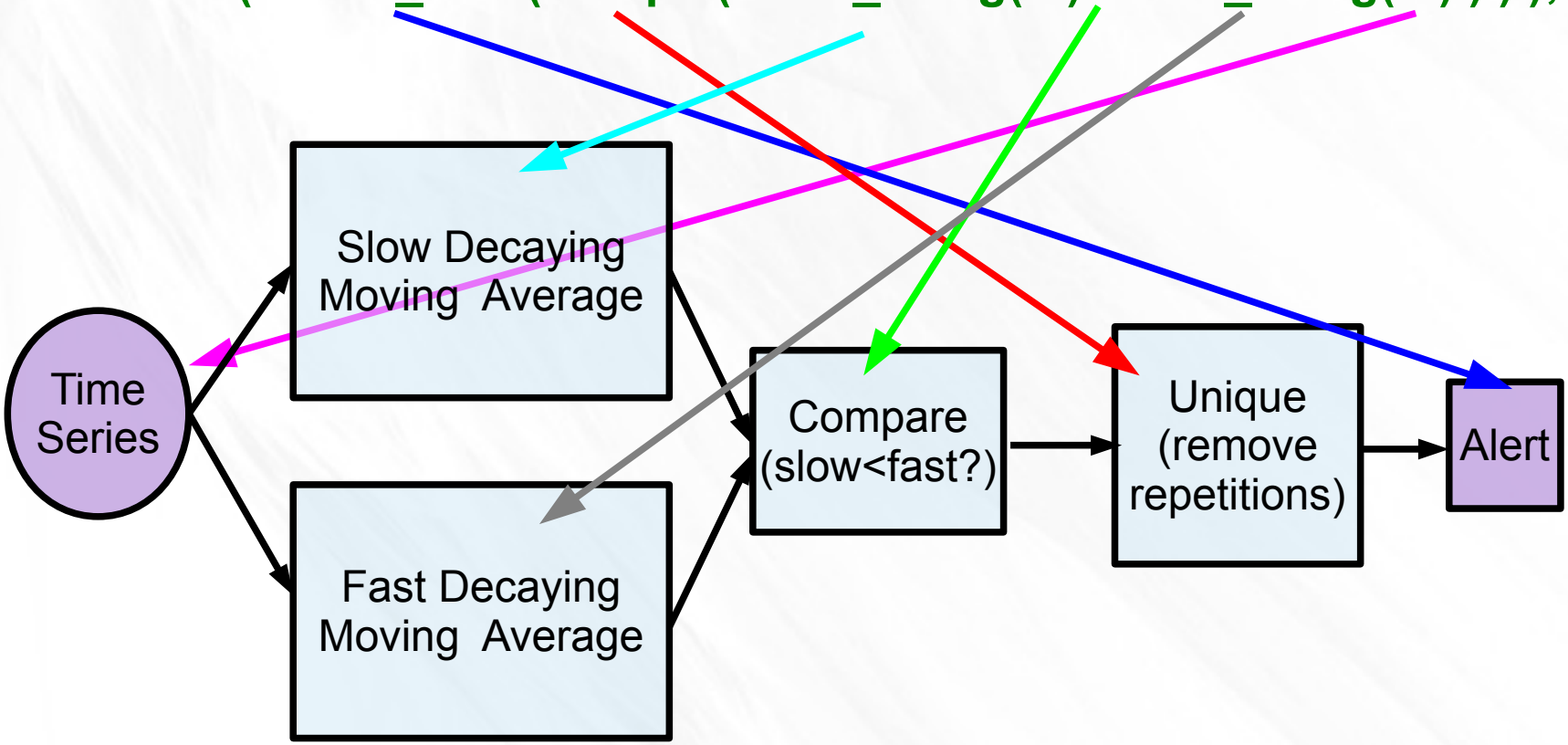
setup

```
HandleEvent(double value) {
    InputStreamPut(ts, value);
}
```

process an event

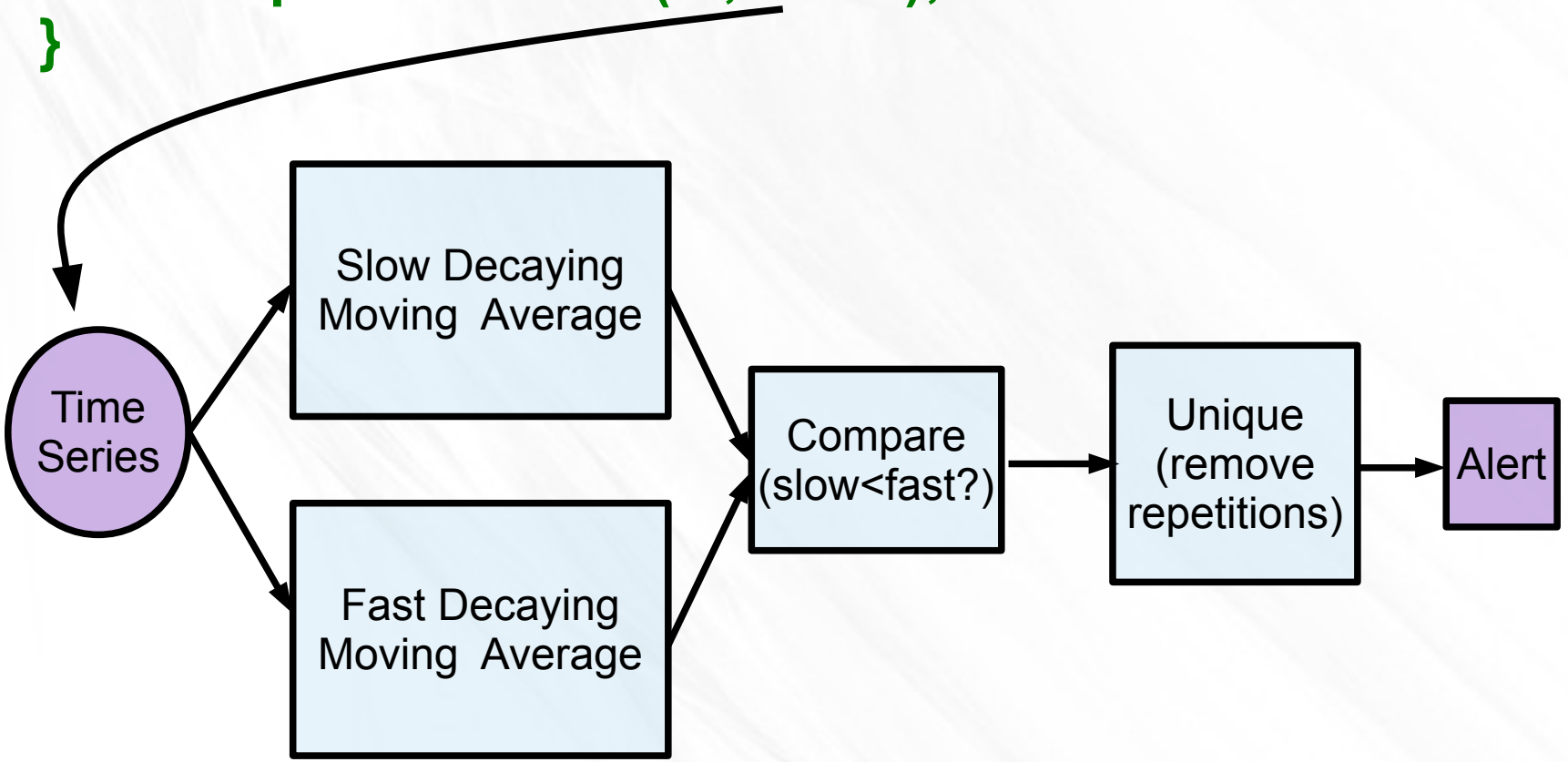
Setup Constructs the Graph

Subscribe(cross_alert(unique(slow_mavg(ts) < fast_mavg(ts))));



Inputs Propagate Automatically Through the Graph

```
HandleEvent(double value) {
        InputStreamPut(ts, value);
}
```



User-Defined Functions

What are **Mavg**, **unique** and **cross_alert**?

- Write a functor **F** that handles a single event
- **Streamify** it.

cross_alert is Streamify<cross>

```
struct cross {
    template<class Sig>
    struct result {
        typedef bool type;
    };
```

Boost result_of protocol (not needed in C++11)

```
    bool operator()(bool golden)
    {
        std::cout << (golden ? “Golden” : “Death”);
        std::cout << “ Cross” << std::endl;
        return golden;
    }
};
```

Process event

unique is Streamify<unique_func>

```
struct unique_func {
    unique()  : mFirst(true) {}

    template<class Sig>
    struct result {
        typedef bool type;
    };

    bool Filter(bool value) const {
        return mFirst || (value != mPrev);
    }

    bool operator()(bool value) {
        mFirst = false;
        return mPrev = value;
    }
private:
    bool mFirst; bool mPrev;

};
```

Boost result_of protocol (not needed in C++11)

Will there be an output? (optional)

Value of the next output

How does it work?

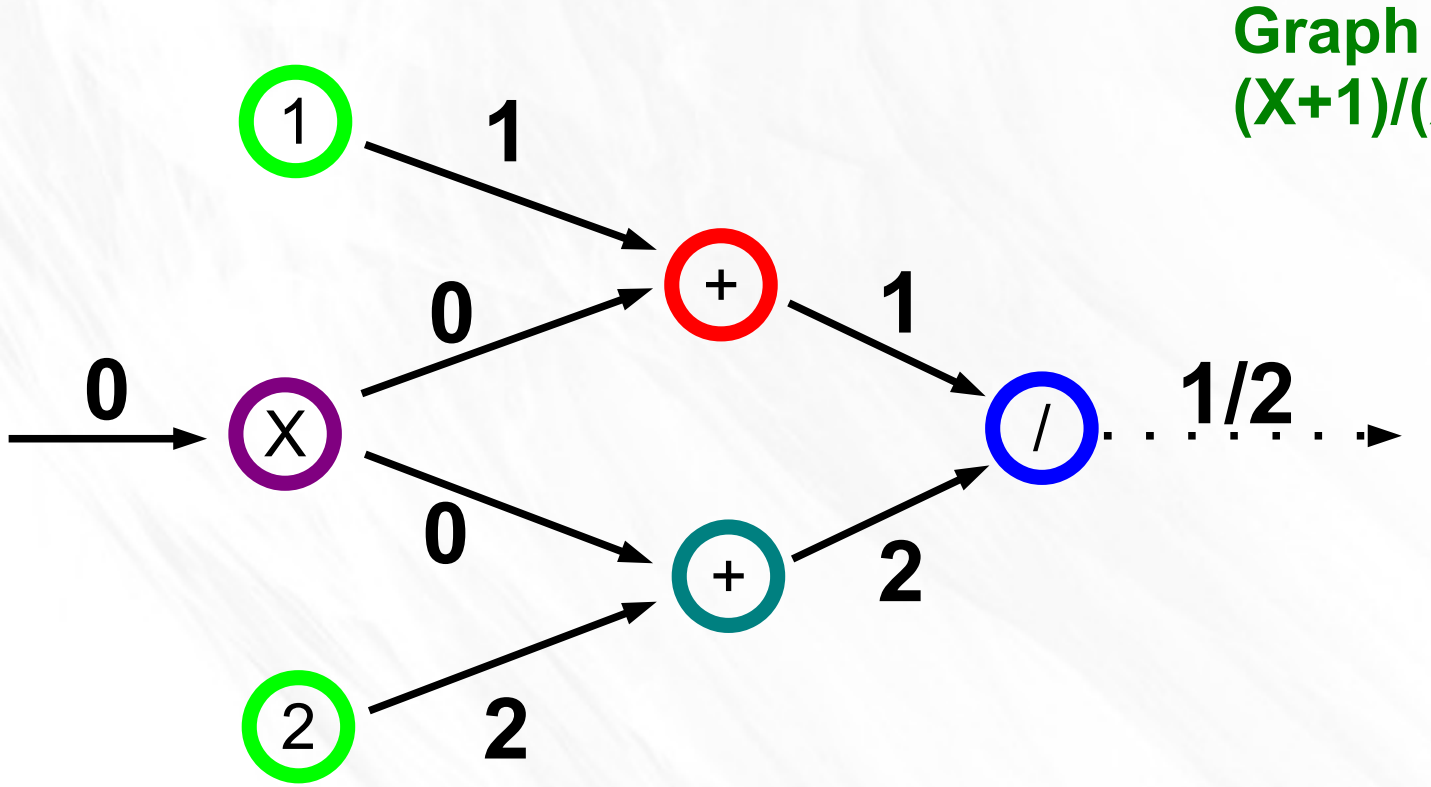
There are two things to talk about:

- The graph data structure
  - How the data propagates through it
- The Subscribe() function
  - How it turns expressions into a graph

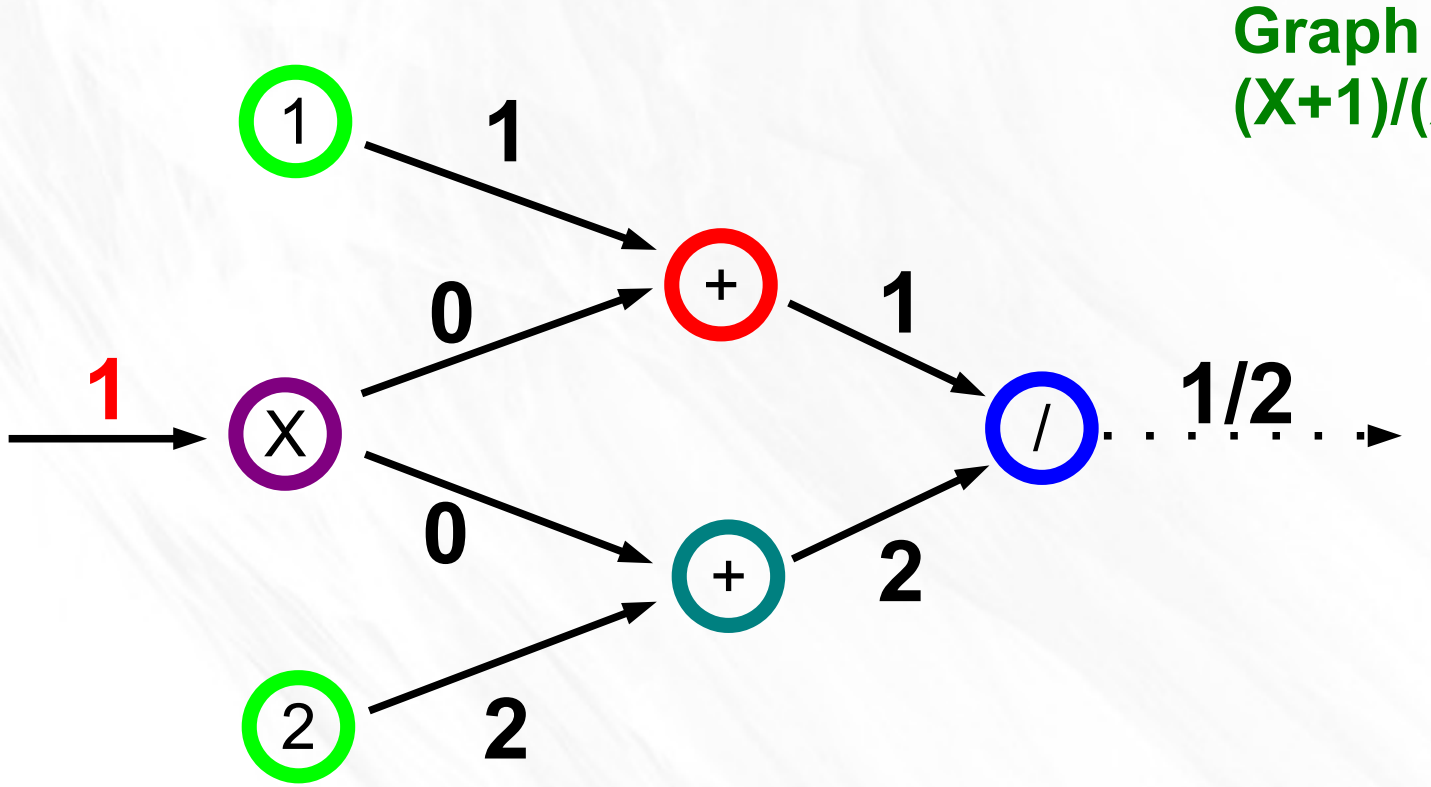
The Streamulus Engine

- Maintains the Graph
  - Nodes have operators
  - Edges have buffers
- Propagates inputs by activating nodes in a *safe* order

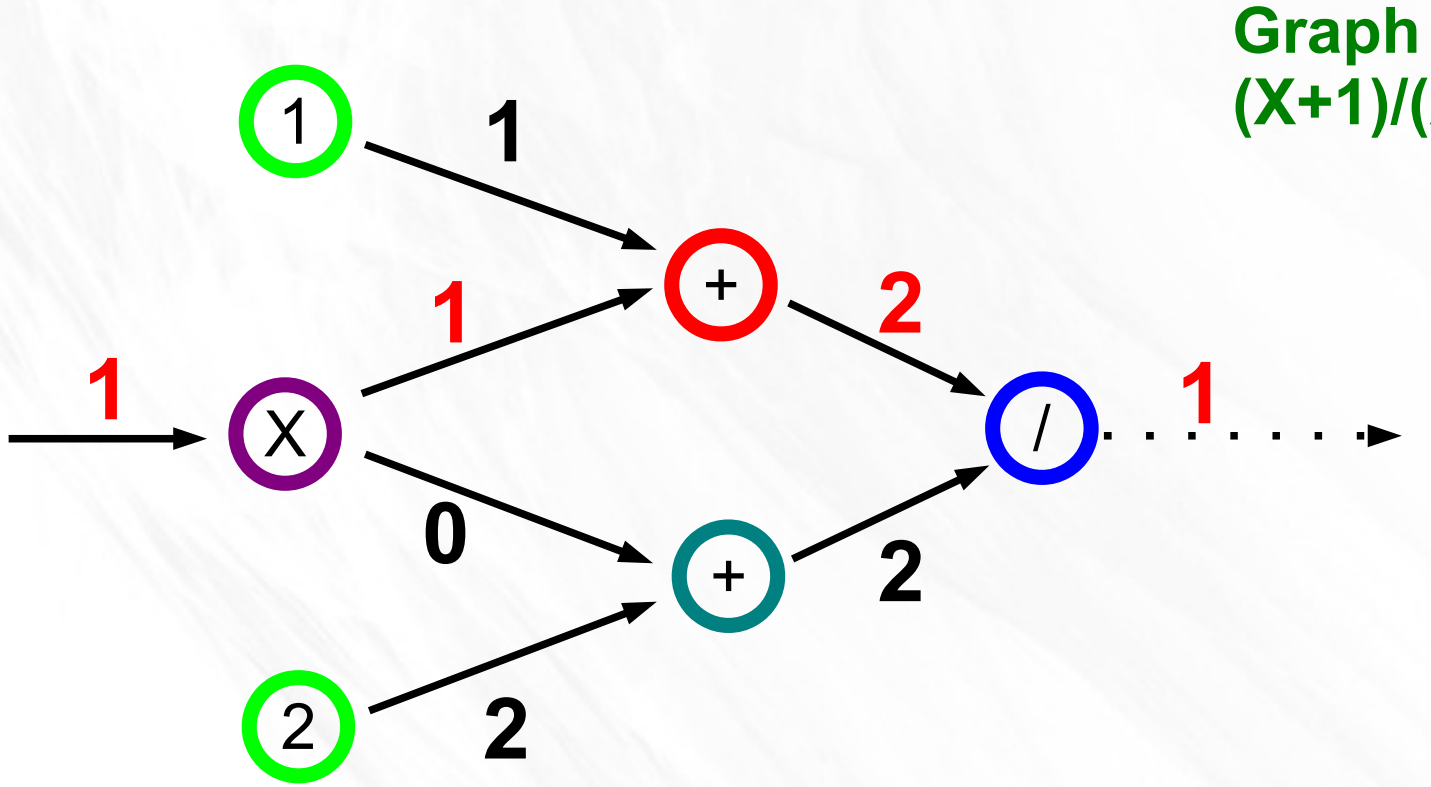
What is a safe order?



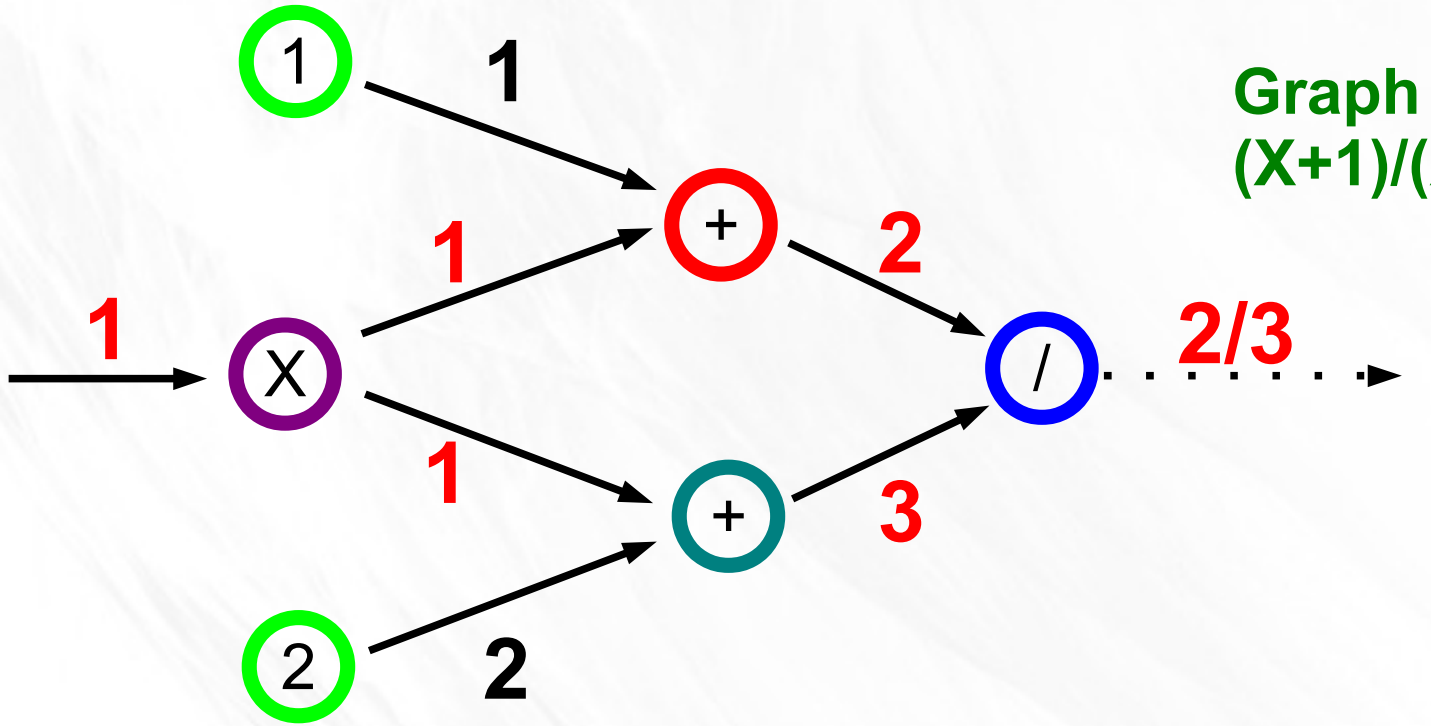
What is a safe order?



What is a safe order?



What is a safe order?

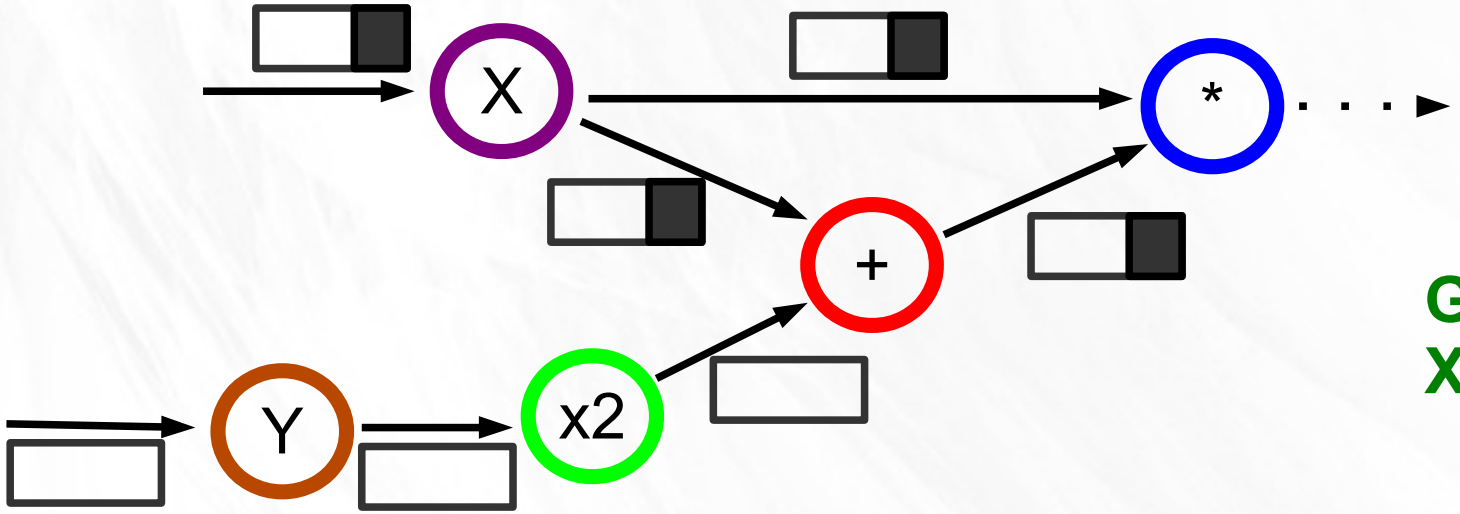


Graph for
(X+1)/(X+2)

both + and + should be activated before /

in other words: topological order

The Streamulus Data Structure



Graph for X*(X+2Y)

Priority Queue of Active Nodes

Priority = { TimeStamp, Index }

TimeStamp of oldest incoming data,
Index of the node in topological order

What's in a Node?

```
class Strop  // for STReam Operator
{
   ...
   virtual bool Work()=0; // return true if emitted output
}
```

Also has *context* data members:

- Pointer to the engine
- Identifier of its node in the graph
- It's topological order index

Streamify<f>

We had:

unique is Streamify<unique_func>

Streamify takes a single-event functor (or object) and creates a strop that *does the right thing*.

You can create your own strops directly

- but for most purposes Streamify should suffice.

InputStream

- A special kind of Strop.
- Has a Tick(value) function
  - Called from outside of Streamulus
  - Causes the node to emit *value* to its output

```
InputStream<double>::type ts=NewInputStream<double>(“TS”);
InputStreamPut(ts, value);
```

Calls ts's Tick function

How Data Propagates

- Engine's Main Loop (single threaded):

  While ActiveNodes is not empty:

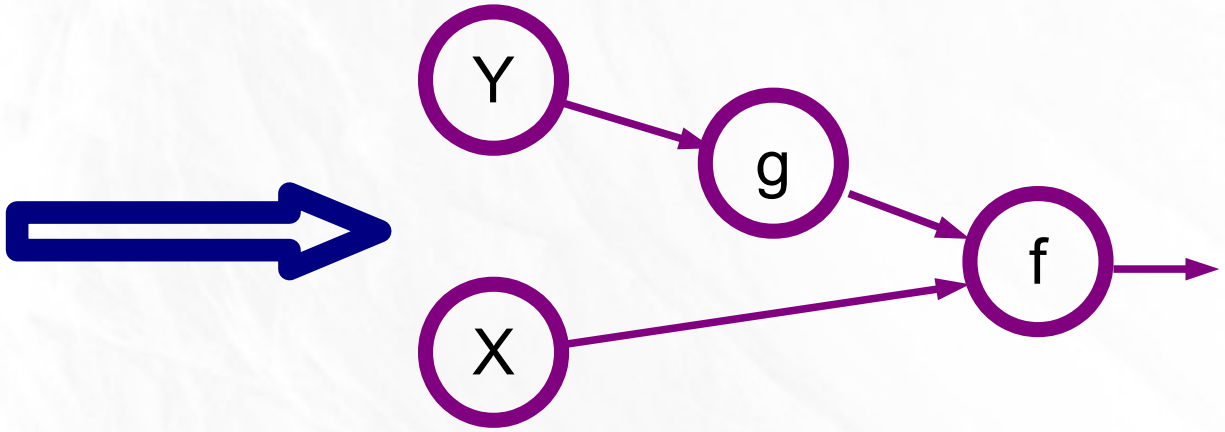
  v = ActiveNodes.Pop()

  v.Work()

- When the queue is empty, the engine idles
  - No busy waiting
- When an input Tick()s, the engine is activated
  - Resumes its main loop

Subscribe()

“f(x,g(y))”

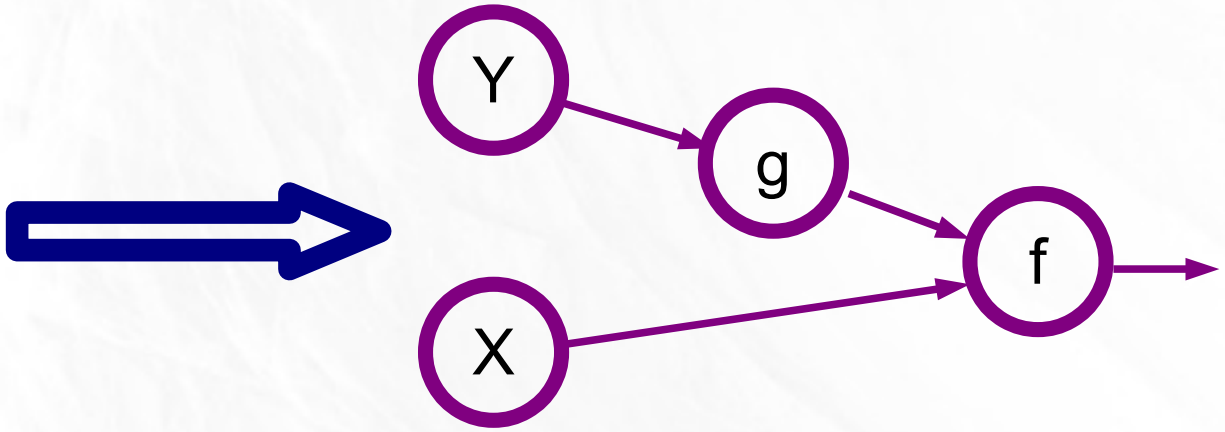


Easy. Anyone with a parser and a stack can do it.

If all the edges carry the same type.

Subscribe()

"f(x,g(y))"



But what is the type of g(y)

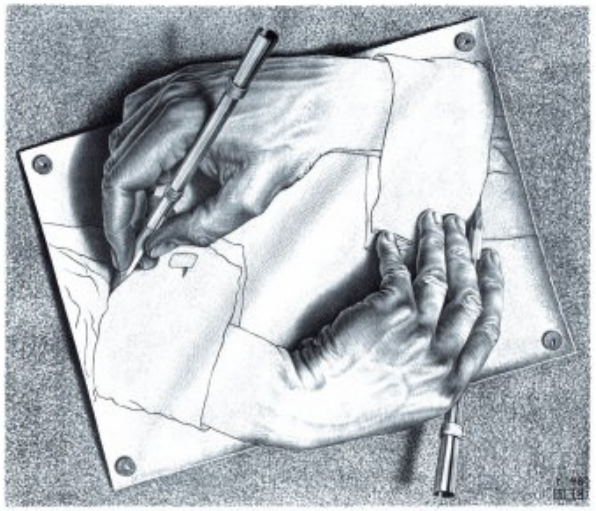
- When y is int?
- When y is a user-defined type?

Two Options

- Avoid the problem
  - Generic container (union, variant): waste space
  - Serialisation: waste time
  - void pointers: unsafe
- Solve the problem
  - Compute the data type of each edge
  - Allocate a buffer for that type
  - How? *C++ Template Metaprogramming*

Metaprogramming

Writing code that generates
or manipulates code



- Compilers
- Source code generators
- Self-modifying programs

C++ Templates

Designed for Generic Programming

```
template<typename T>
T max(T a, T b) {
    return a > b ? a : b;
}
```

Paradigm	Type resolution
Generic Programming	During compilation
Polymorphism	At runtime

C++ Template Metaprogramming

"Programming with types"

Metafunctions map types to types:

```
template<typename T>
struct VectorOfPairs {
    typedef  std::vector<std::pair<T>>  type;
};
```

typedef is an assignment:

```
typedef VectorOfPairs<double>::type my_vector;
```

C++ Template Metaprogramming

Recursive metafunctions make the compiler compute stuff:

```
template<int i>
struct Factorial {
    static const int value = i * Factorial<i-1>::value;
};

struct Factorial<1> {       // base case
   static const int value = 1;
};

int five_factorial = Factorial<5>::type;
```

C++ Template Metaprogramming

Control flow via template specialization:

```
template<typename T>
struct IntToDouble {
    typedef T type;
};

struct IntToDouble<int> {
    typedef double type;
};

IntToDouble<int>::type      // == double
IntToDouble<char>::type     // == char (unchanged)
```

C++ Template Metaprogramming

Compile-Time Data Structures

Linked List of Types

```
struct End {};

template<typename T, typename NEXT>
struct Node {
        typedef T type;
        typedef NEXT next;
};

typedef Node<int, Node<bool, Node<char, End>>> List;
```

C++ Template Metaprogramming

Insert types to a list:

```
template<typename LIST, typename T>
struct Push {
    typedef Node<T, LIST> type;
};


typedef Push<End, int>::type L1;
typedef Push<L1, bool>::type L2;
typedef Push<L2, char>::type L3;
```

C++ Template Metaprogramming

Insert types to a list:

```
template<typename LIST, typename T>
struct Push {
    typedef Node<T, LIST> type;
};


typedef Push<End, int>::type L1;
typedef Push<L1, bool>::type L2;
typedef Push<L2, char>::type L3;
```

C++ Template Metaprogramming

Compute the length of a list:

```
template <typename T>
struct Size;

template<typename T, typename NEXT>
struct Size<Node<T, NEXT> > {
    static const int value = 1 + Size<NEXT>::value;
};


Template <>
struct Size<End> {
    static const int value = 0;
};
```

Useful Boost Libraries

MPL (Aleksey Gurtovoy and David Abrahams)

- Utilities, Data Structures, Sequences, Iterators

Fusion (Joel de Guzman, Dan Marsden, Tobias Schwinger)

- Heterogenous containers

```
fusion::vector<int, char, bool>  my_vector;
```

Proto (Eric Niebler + Joel Falcou, Christophe Henry)

- A framework for building Domain-Specific Embedded Languages in C++

Using Proto

- Define a grammar
  - Which expression are valid?
- Define transformations
  - What should become of each sub-expression?
- Activate the grammar on an expression

Operator Overloading in C++

```
class MyType    { … };
class YourType { … };
class OurType    { … };

OurType operator+(MyType mine, YourType yours) {
    return …. ;  // Compute an OurType from the inputs
}


MyType mine;
YourType yours;
OurType ours = mine + yours;
```

Expression → Tree

Proto defines a static expression type

proto::expr

and overloads all operators for it.

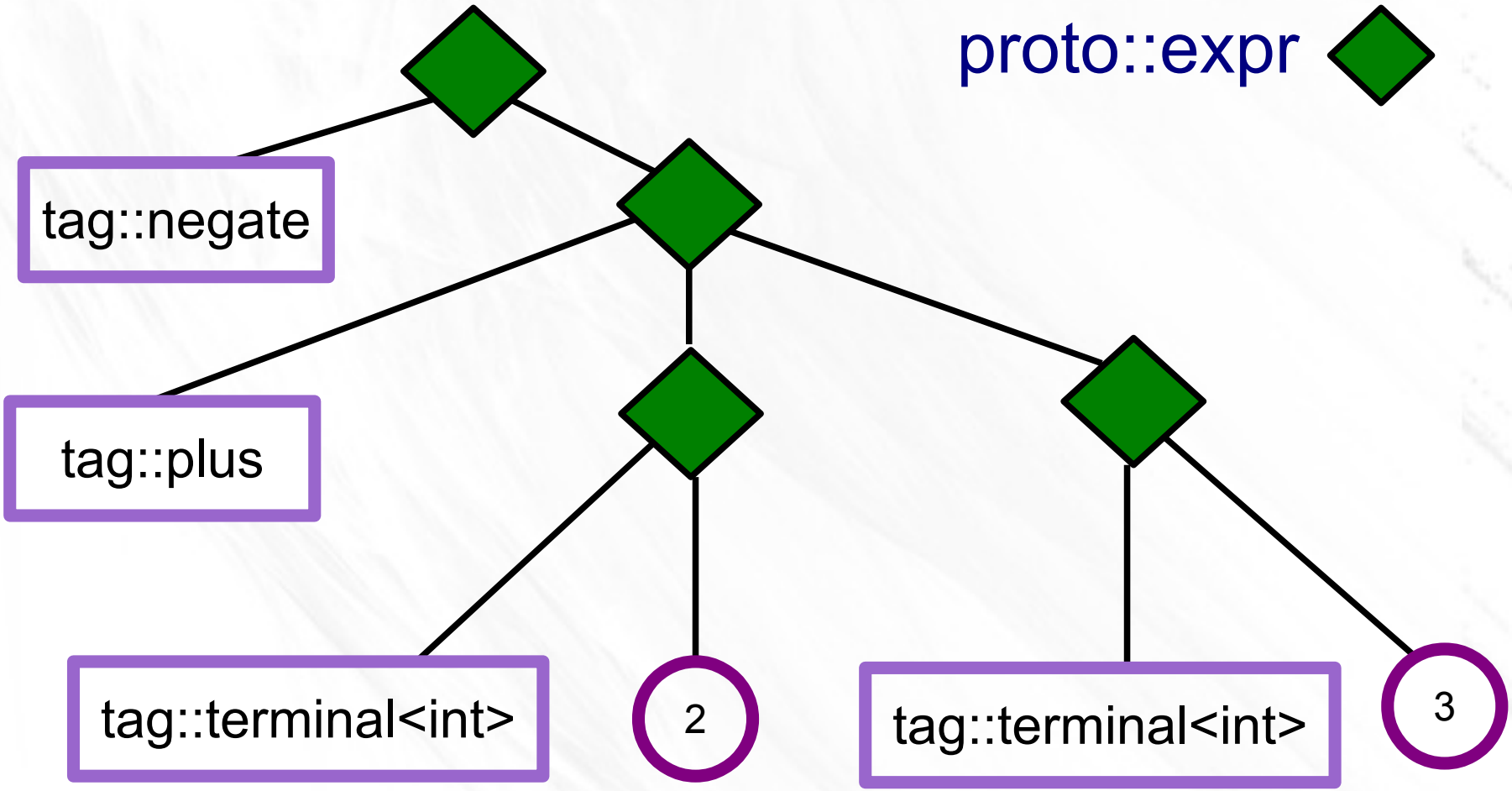
For example:

expr1 - expr2

returns something like

proto::expr<tag::minus, list<expr1,expr2>,2>

Expression Tree for -(2+3)



The proto::expr Type

```
template<typename Tag,              // what this node does
         typename Args,             // who it does it to
         long Arity = Args::arity >
struct expr;


template< typename Tag, typename Args >
struct expr< Tag, Args, 1 > {               // unary expression
    typedef typename Args::child0 proto_child0;
    proto_child0 child0;
    // ...
};

...                                 // specialisations for other arities
```

Creating Proto Expressions

- Define proto terminals

  proto::terminal<int>::type x = {12};

- x is a proto expression

  → So is any expression involving x

  ~((x+12)/x & 0xff)

Function call expressions

proto::expr<tag::function, Args,... >

- First arg is a proto::terminal<func>::type
  - Identifies the function
- Then the function's arguments
  - Arbitrary proto::expr's

A Proto Grammar

Recursive definition of valid expressions

```
struct arithmetic
  : proto::or_<
      proto::plus       <arithmetic, arithmetic>
    , proto::minus      <arithmetic, arithmetic>
    , proto::multiplies <arithmetic, arithmetic>
    , proto::divides    <arithmetic, arithmetic>
    , proto::terminal   <proto::_>  // anything

> {};
```

A Grammar With Transforms

```
struct arithmetic
  : proto::or_<
      proto::when<
        proto::plus<arithmetic, arithmetic>,
        Plus(arithmetic(proto::_left),
             arithmetic(proto::_right)>
    , proto::when<
        proto::minus<arithmetic, arithmetic>,
        Minus(arithmetic(proto::_left),
              arithmetic(proto::_right)>

      …
    , proto::when<
       proto::terminal  <proto::_>,
       proto::_value>

> {};
```

A Transform

A functor that publishes its return type as result_type

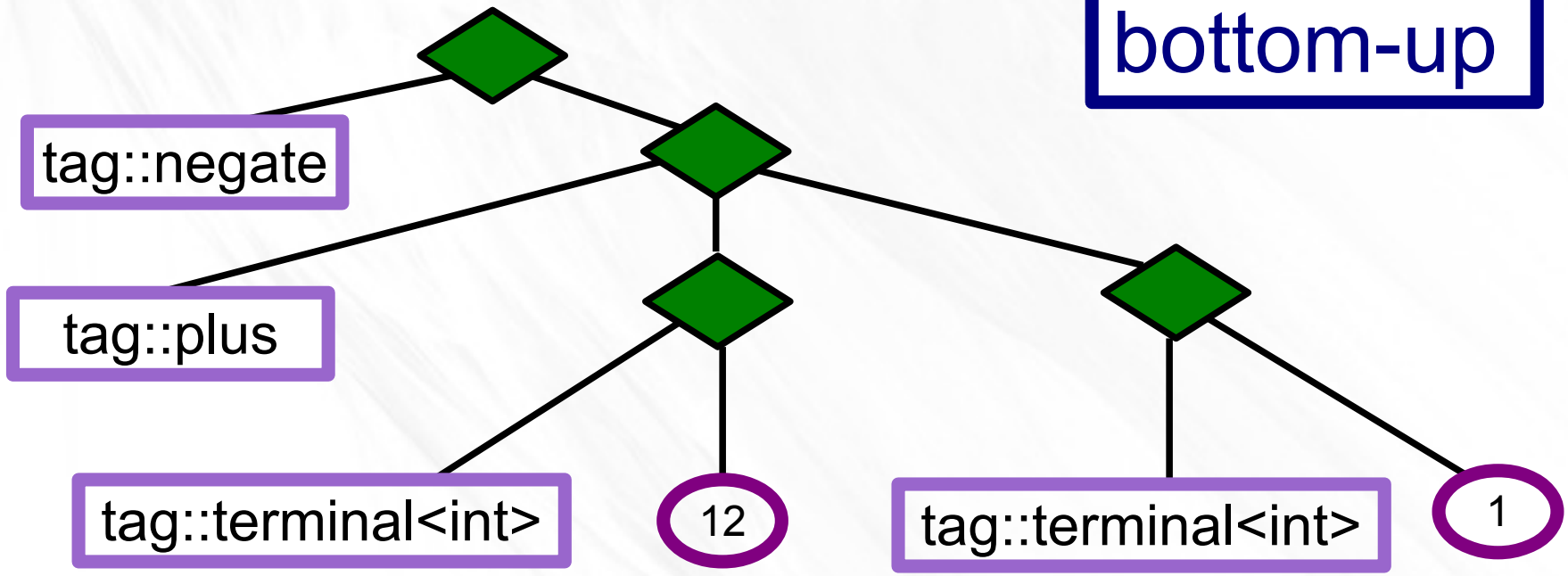
```
struct Plus : proto::callable
{
    typedef int result_type;

    int operator()(int left, int right) {
        return left+right;
    }
};
```

Invoking a Grammar

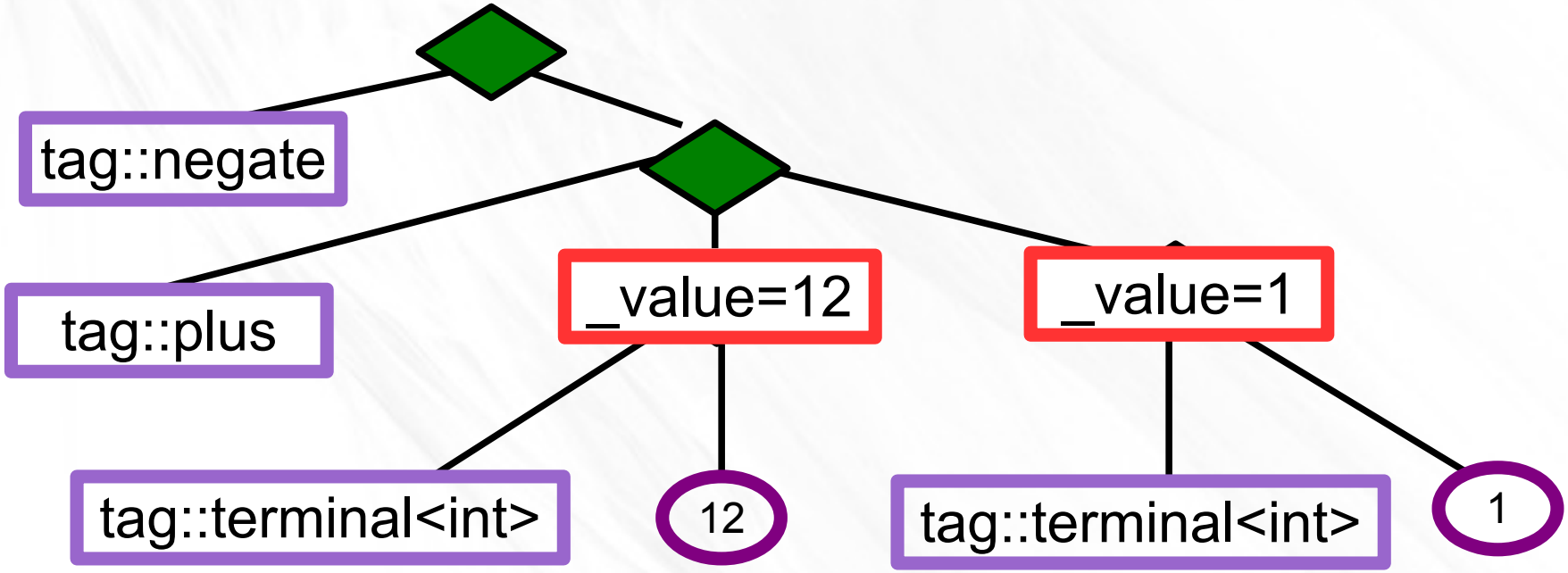
```
proto::terminal<int>  x = {12};
int result = arithmetic() ( -(x+1) );
```

Apply the transforms bottom-up



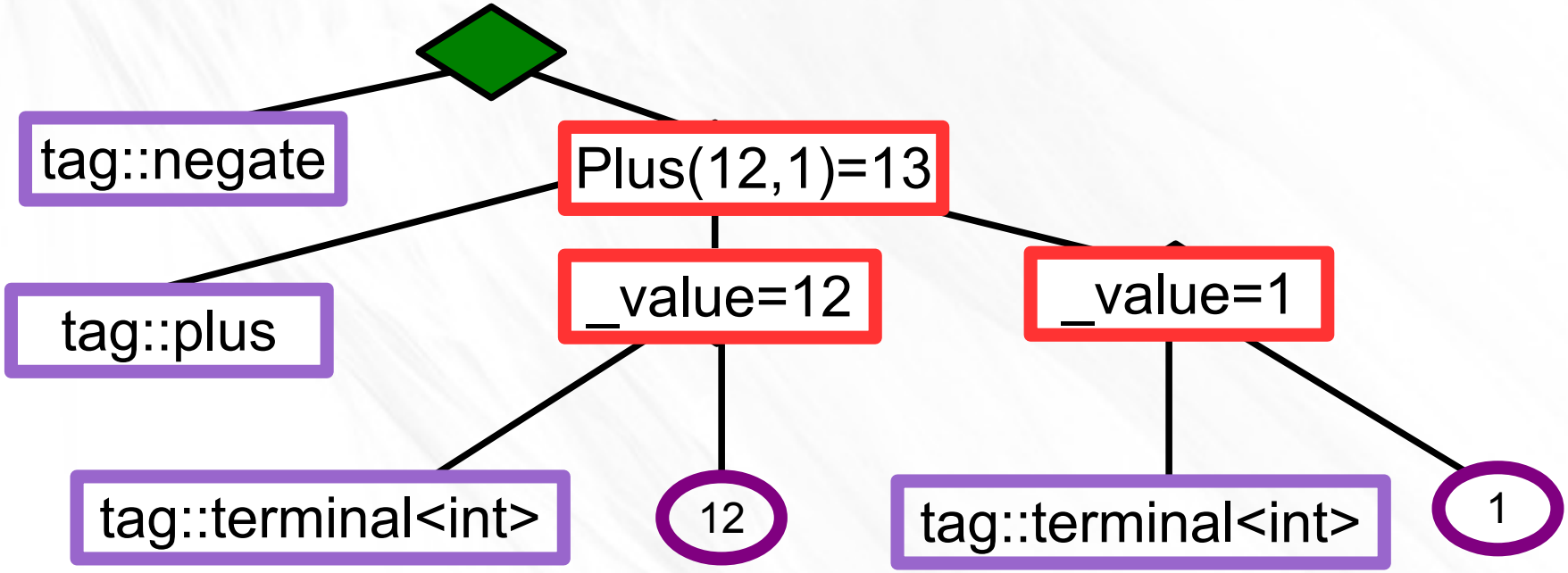
Invoking a Grammar

```
proto::terminal<int>  x = {12};
int result = arithmetic() ( -(x+1) );
```



Invoking a Grammar

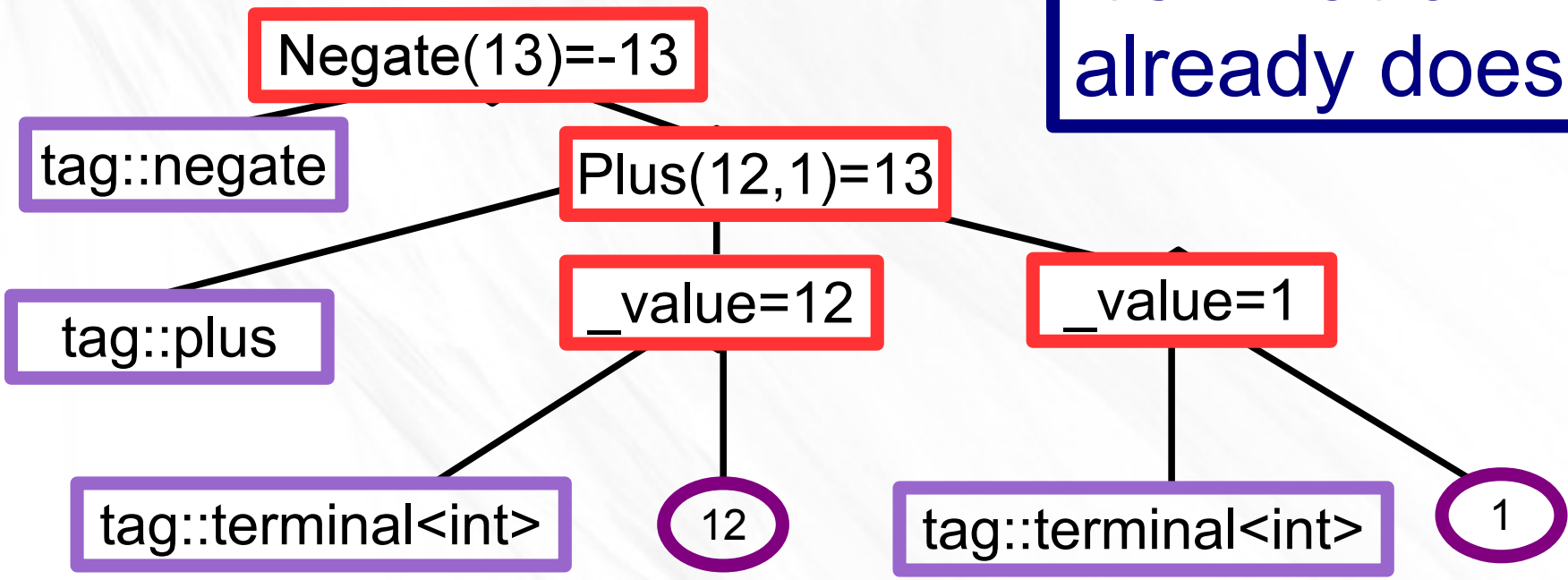
```
proto::terminal<int>  x = {12};
int result = arithmetic() ( -(x+1) );
```



Invoking a Grammar

```
proto::terminal<int>  x = {12};
int result = arithmetic() ( -(x+1) );
```

This example is not very useful. It's what c++ already does.

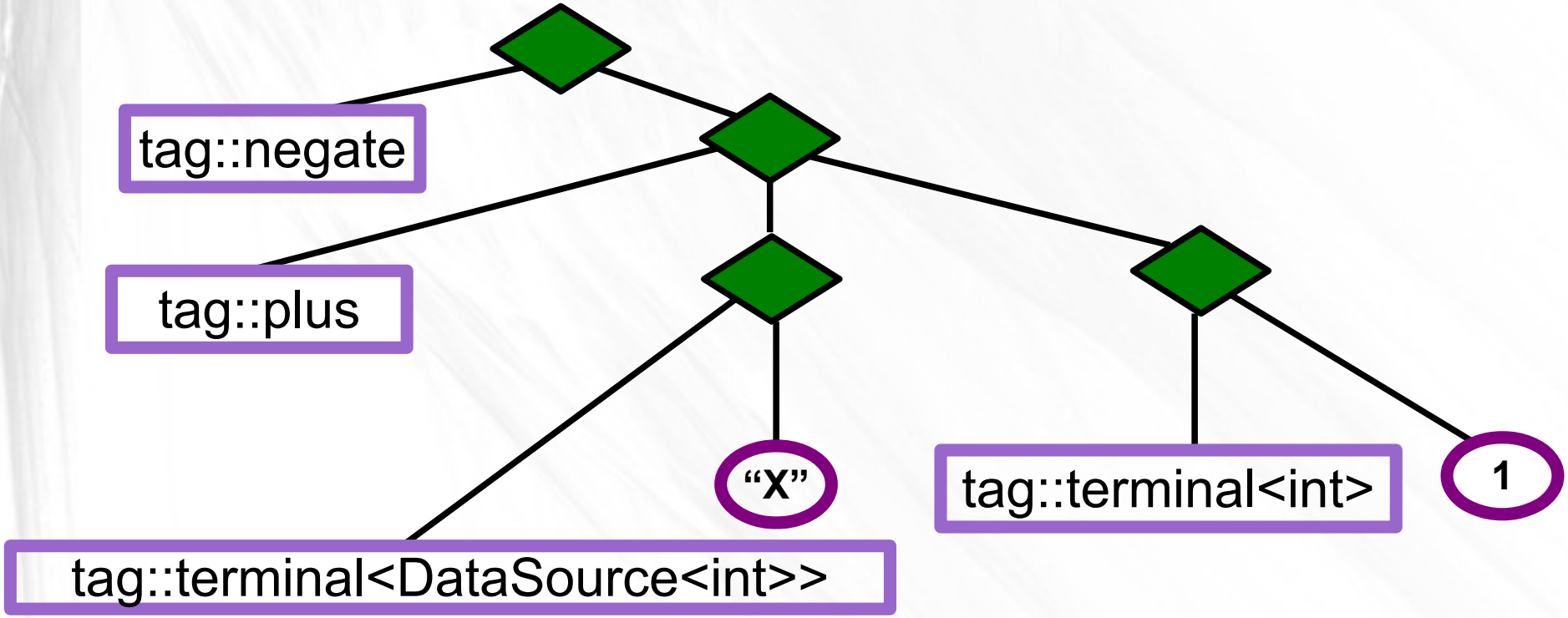


The Streamulus Grammar

- Identifies all operators, as well as user-defined functions.
- Each transform
  - Creates a strop for the node's operator/func
  - Inserts it to the graph
  - Connects it to child-nodes' strops
    - Which were created recursively
  - Returns a pointer to the new strop

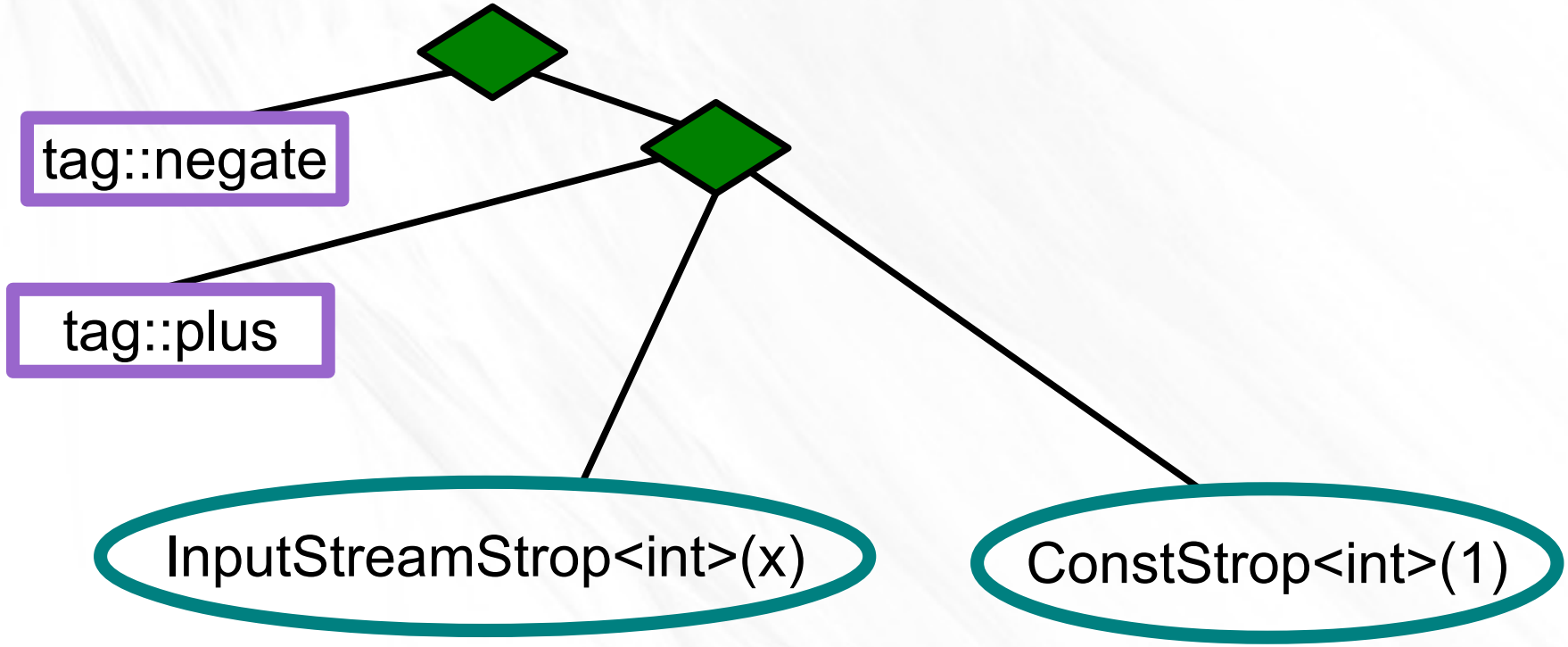
Subscribe()

InputStream<int>::type x = NewInputStream<int>("X");
Subscribe (-(x+1));



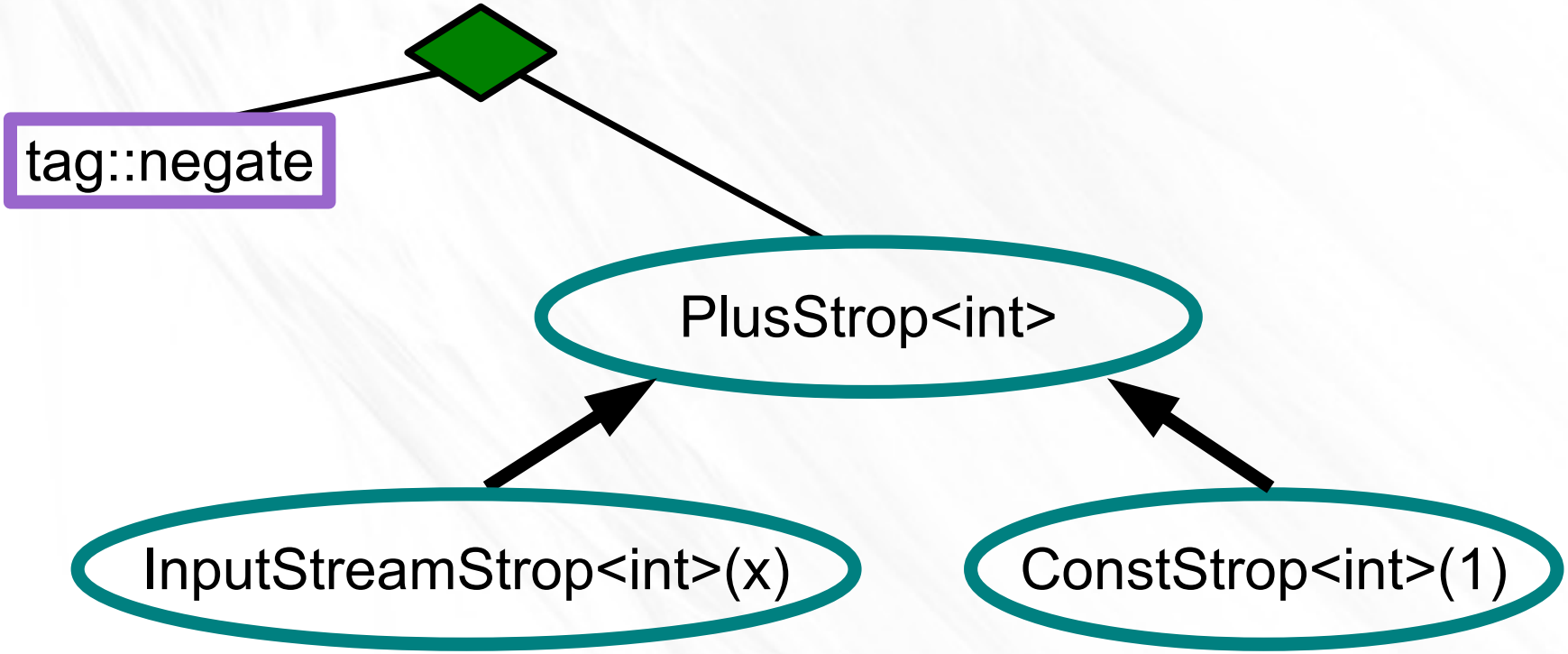
Subscribe()

InputStream<int>::type x = NewInputStream<int>("X");
Subscribe (-(x+1));



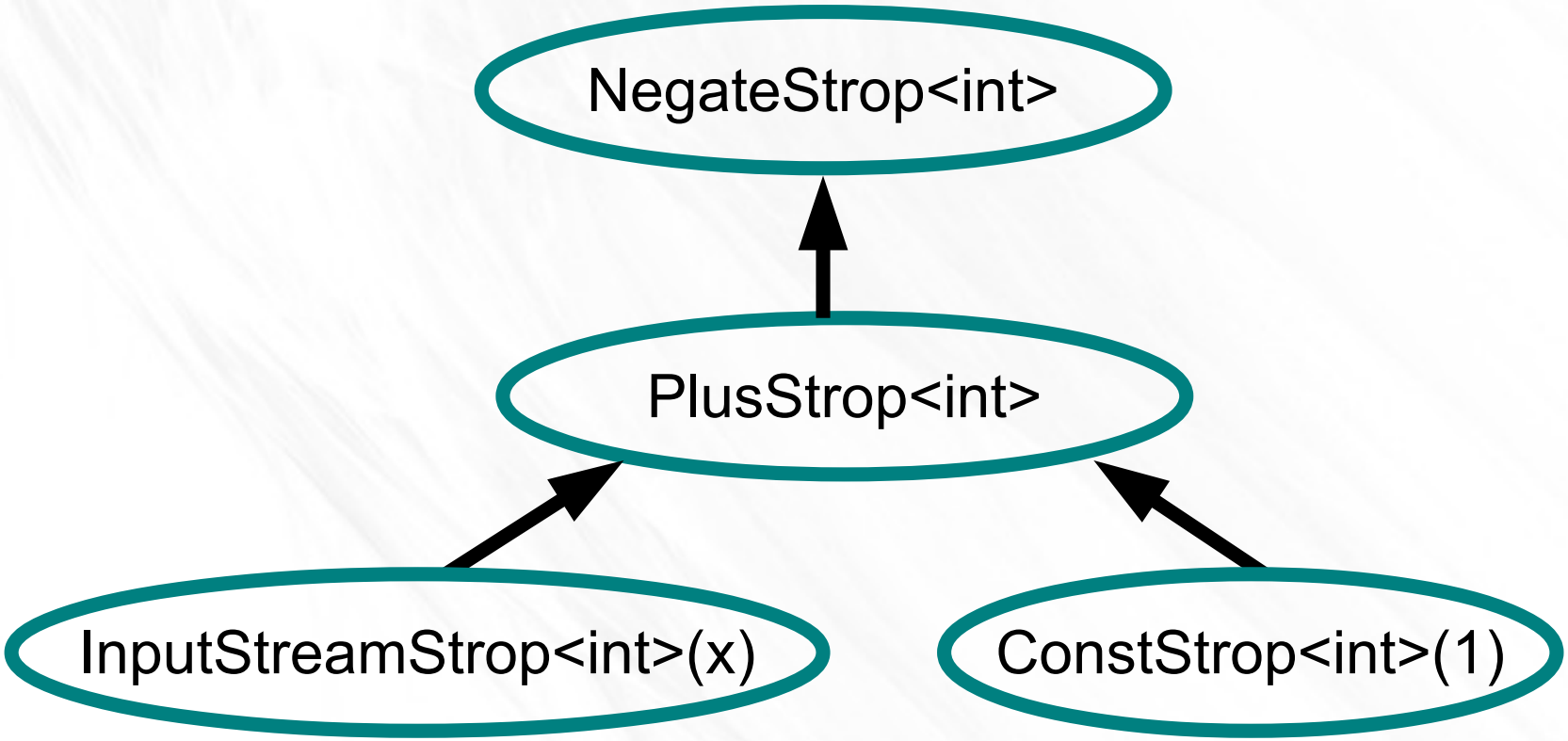
Subscribe()

InputStream<int>::type x = NewInputStream<int>("X");
Subscribe (-(x+1));



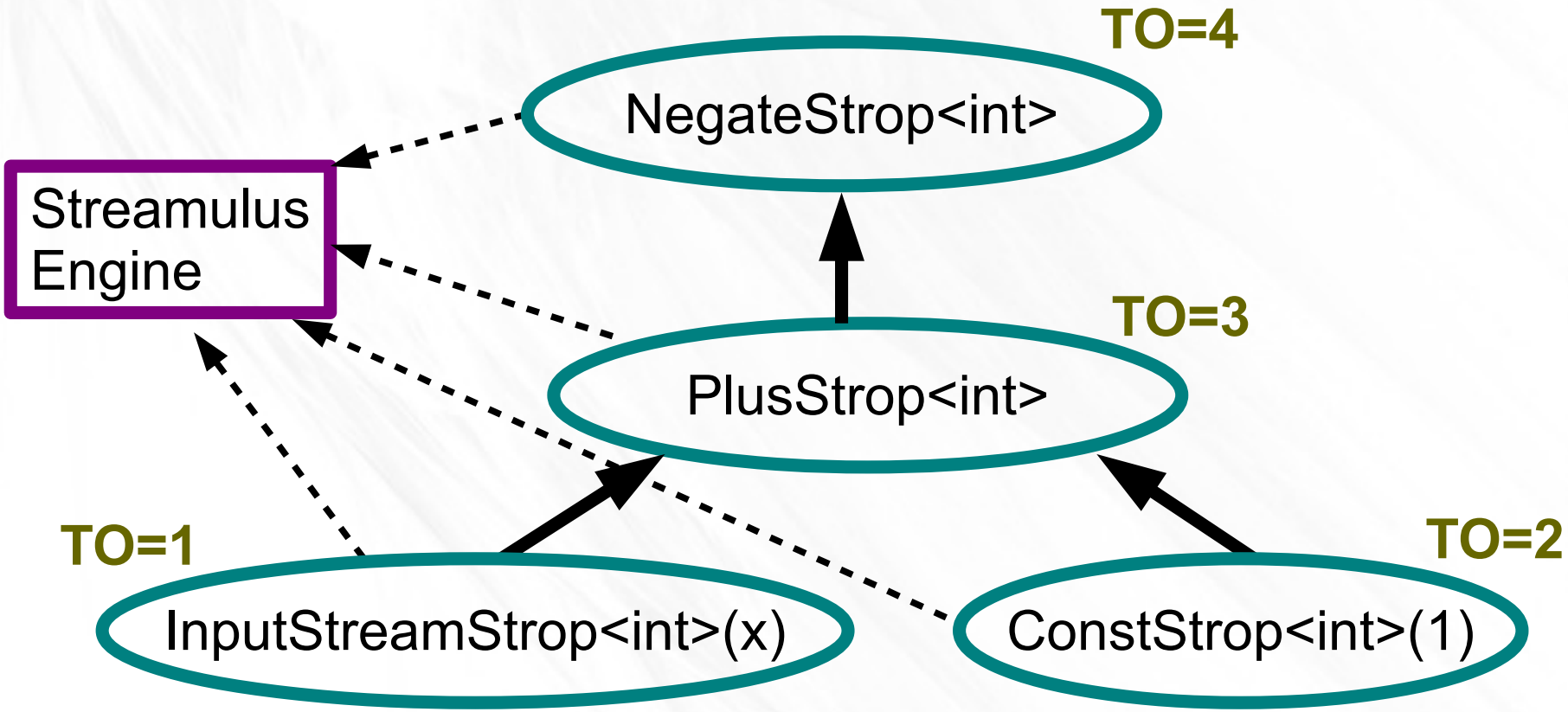
Subscribe()

InputStream<int>::type x = NewInputStream<int>(“X”);
Subscribe (-(x+1));



Subscribe()

Finally: Compute topological order
Link the nodes to the engine



Status

- First Release - soon
- User Manual – eventually
  - Nagging will help
- There's a lot to do
  - Improve it (e.g., multi-core version)
  - Apply it
- It's open-source, join in.

Links

- www.streamulus.com
  - Link to github from there
- Follow @streamulus on twitter
  - Infrequent notifications (releases, news, etc)