Advanced Template Meta-programming for Portable Performance

Matt Might (Chris Earl's Ph.D.)

Advanced Template Meta-meta-programming for Portable Performance

Matt Might (Chris Earl's Ph.D.)

Portable

Performant

Portable

Performant

Expressive

Tolerant

Correct

Problem

James

James says

```
c <<= a + sin(b);
```

“to act exactly like”

```
Field::const iterator ia = a.begin();
Field::const iterator ib = b.begin();
for(Field::iterator ic = c.begin();
      ic != c.end(); ++ic, ++ia, ++ib) {
  *ic = *ia + sin(*ib);
};
```

and

“run multicore or GPU...”

“...or the Xeon Phi.”

How?

New language!

You can use any language...

– James

...as long as it's C++.

– James

C++ Template Metaprogramming

Different?

Multibackend

Meta-meta-programming

Idea

$$\vec{c} = \vec{a} + \vec{sin}(\vec{b})$$

```
c <<= a + sin(b);
```

a
b
c
+

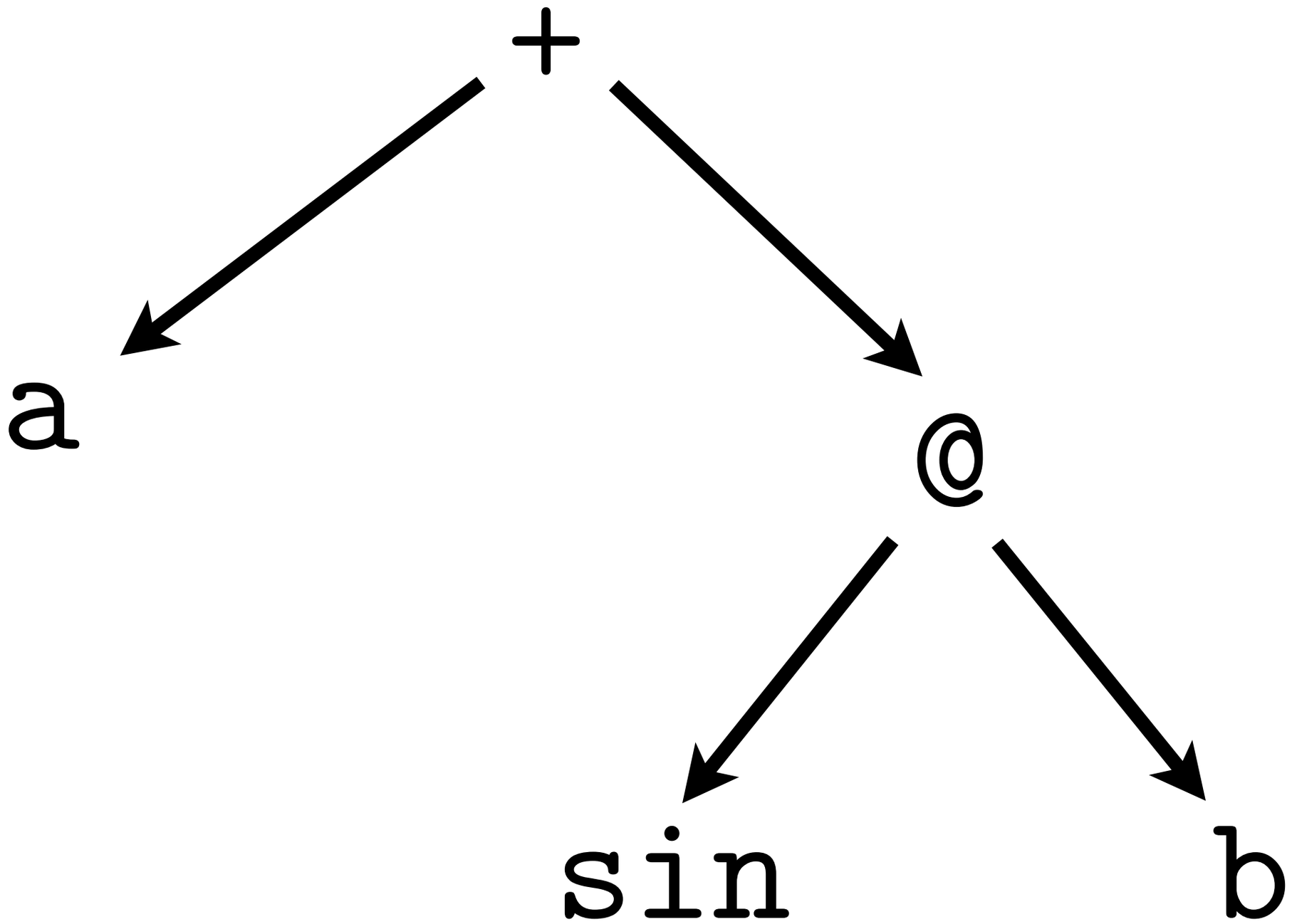
```
Field::const iterator ia = a.begin();
Field::const iterator ib = b.begin();
for(Field::iterator ic = c.begin();
      ic != c.end(); ++ic, ++ia, ++ib) {
  *ic = *ia + sin(*ib);
};
```

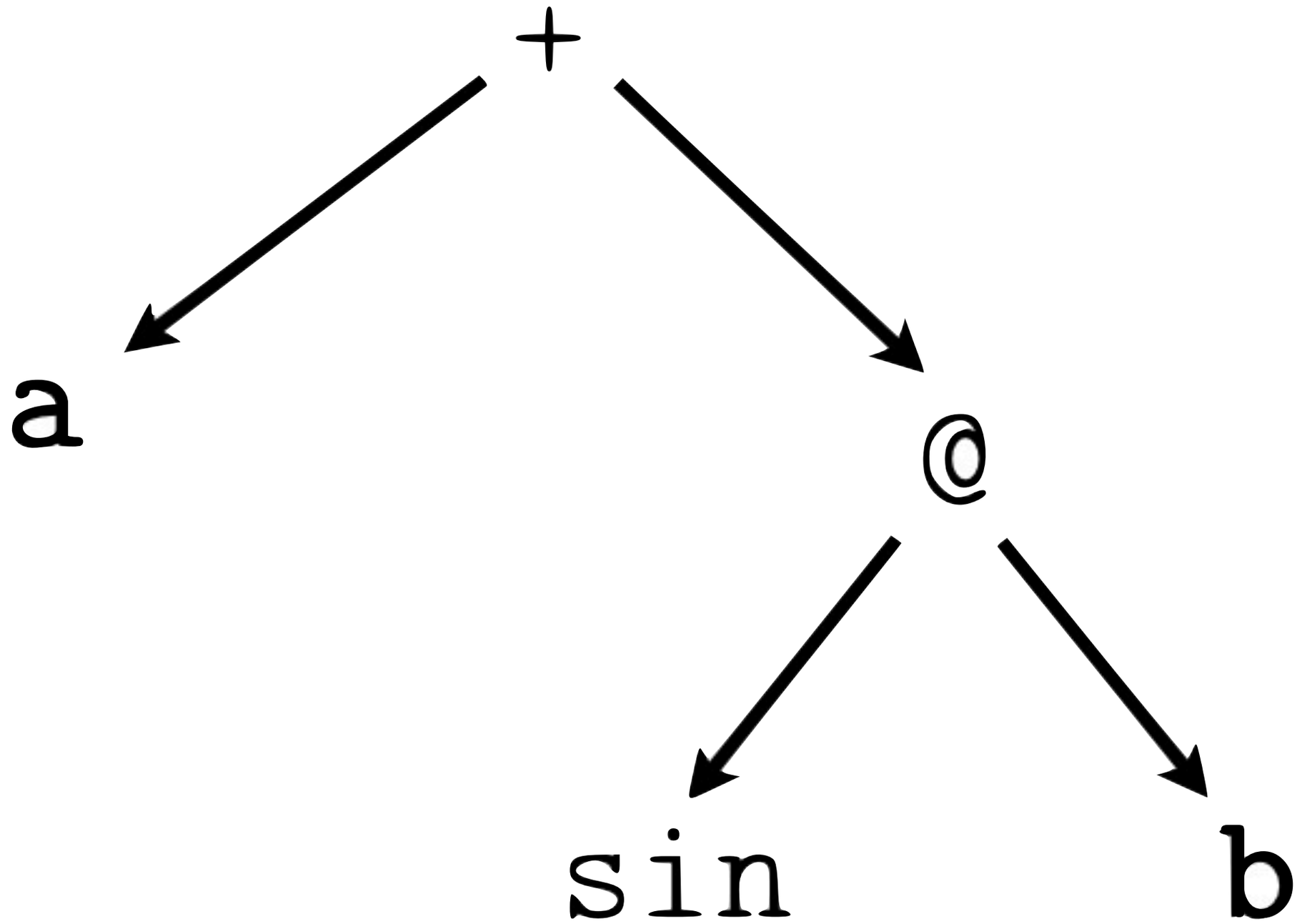
```
c <<= a + sin(b);
```

```
c <<= a + sin(b);
```

$$a + @$$

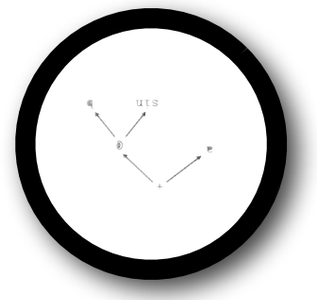
$$\sin b$$


+
a
@
sin
b

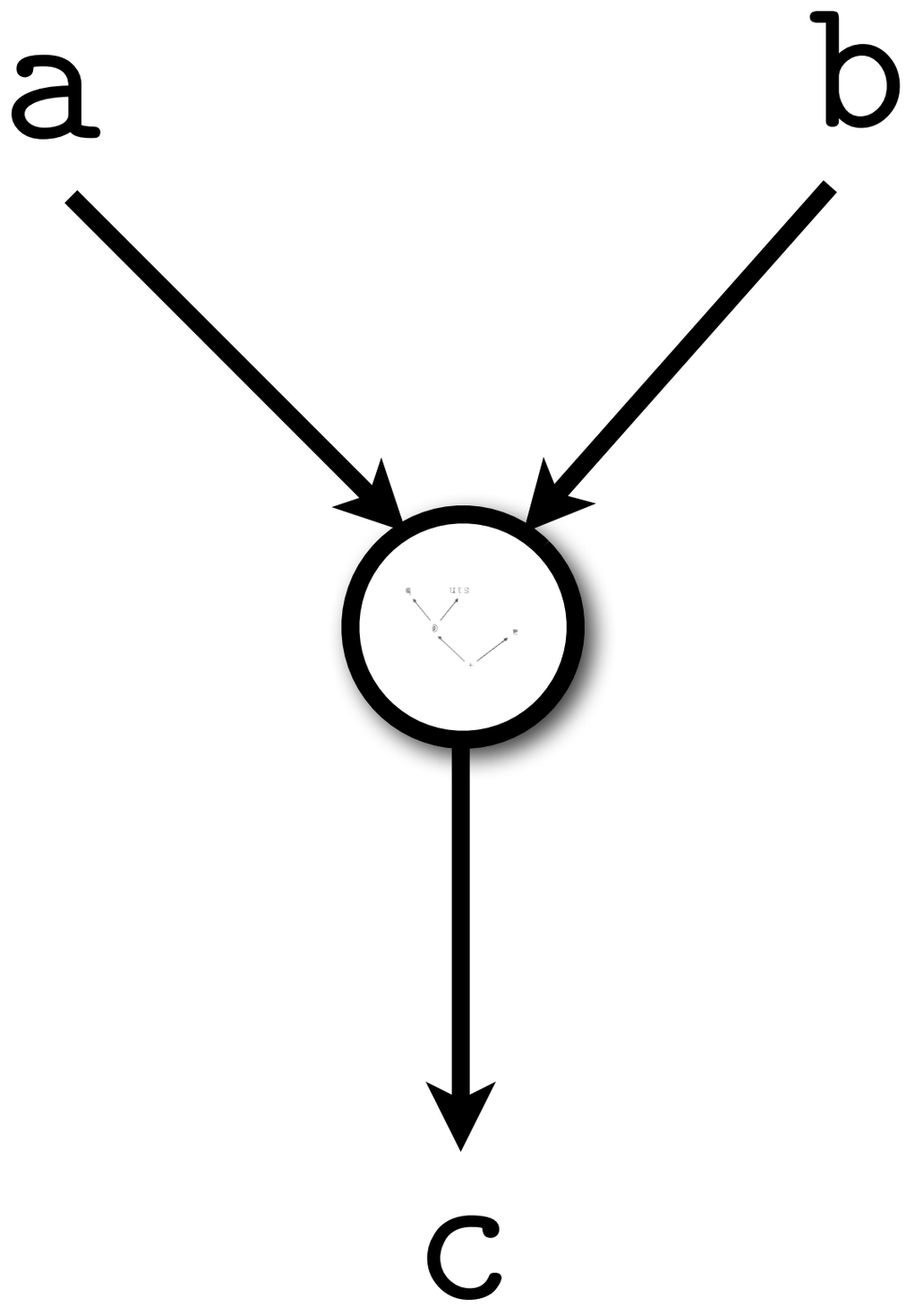

+
a
@
sin
b

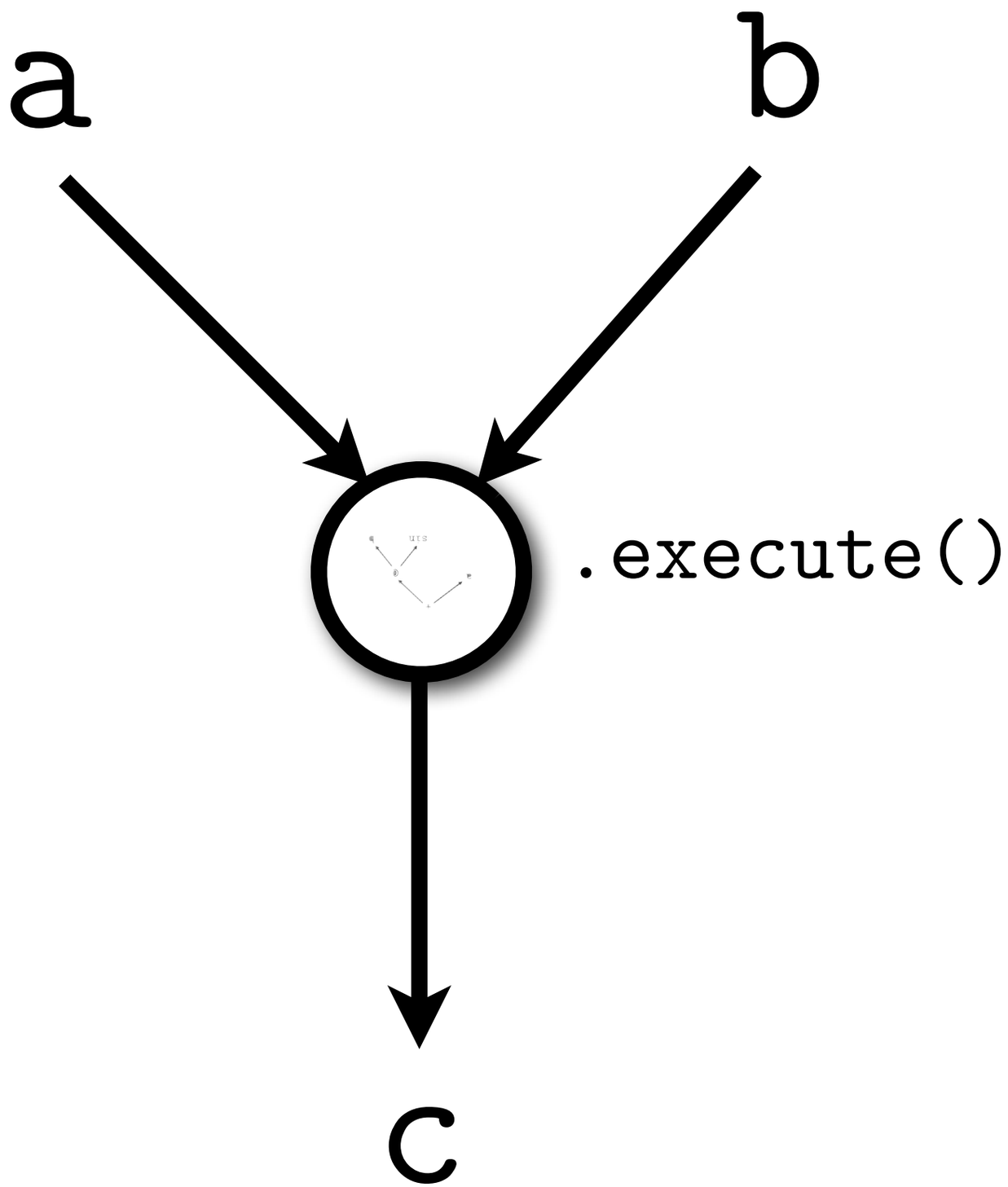
a

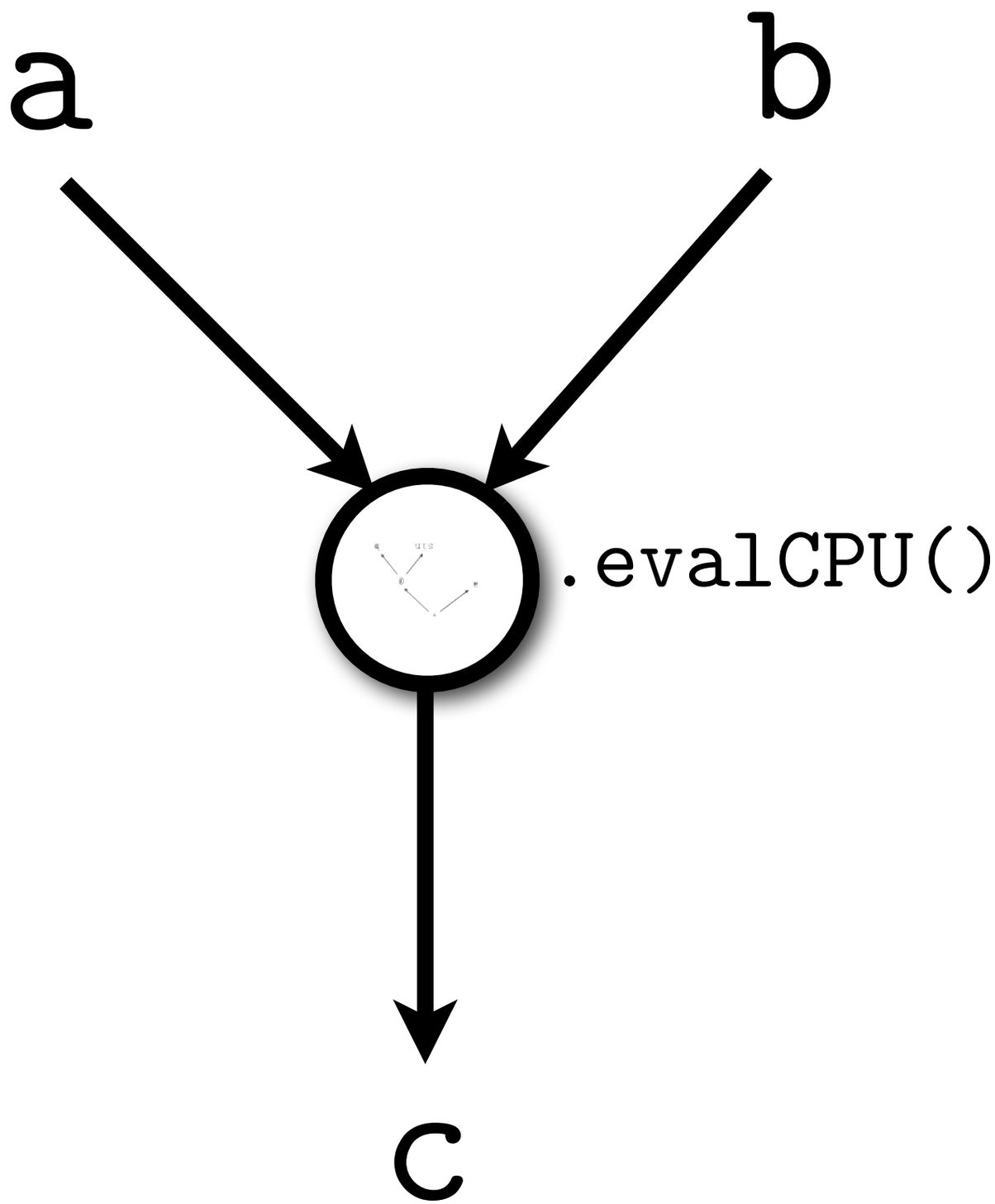
b

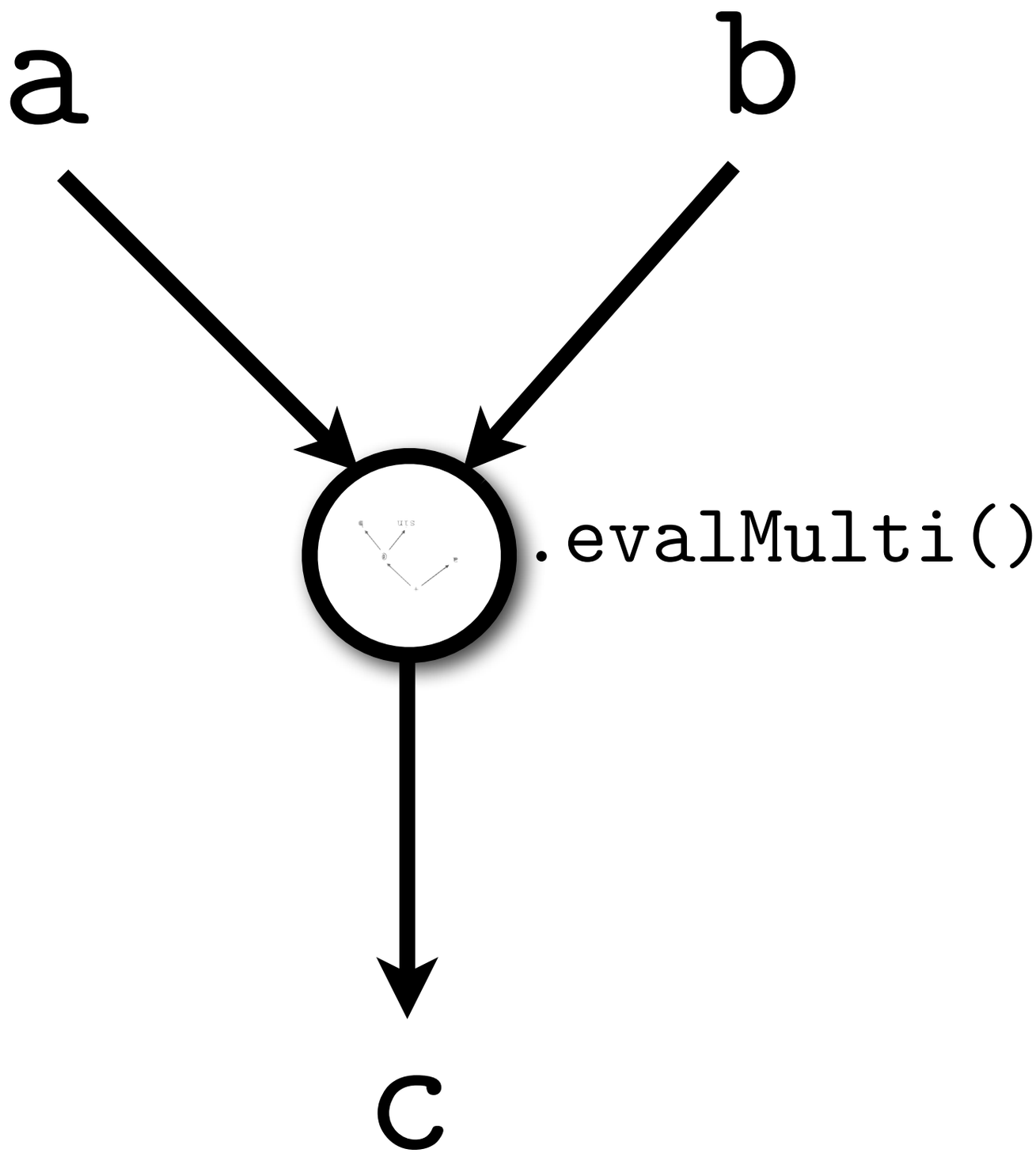


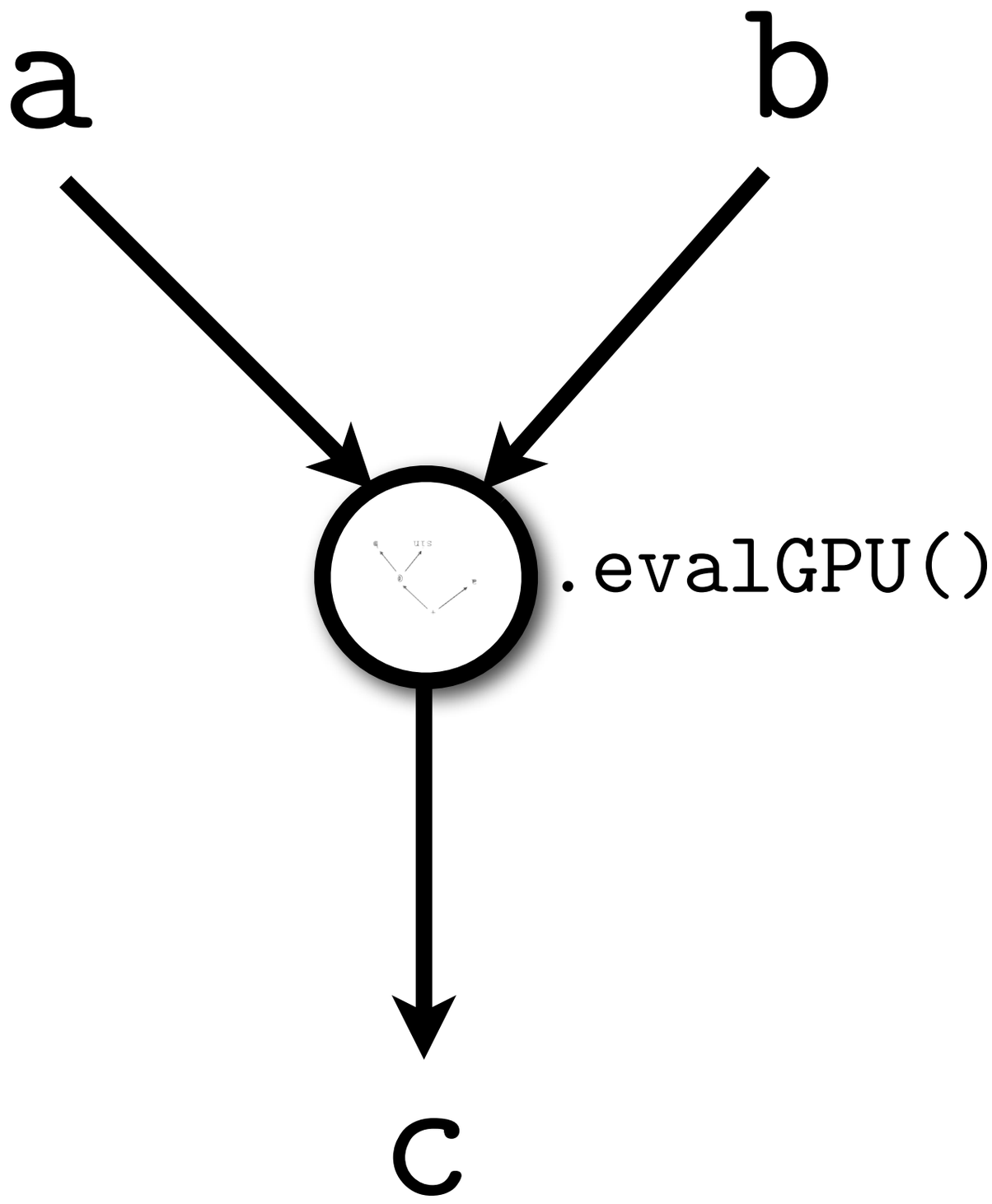
c

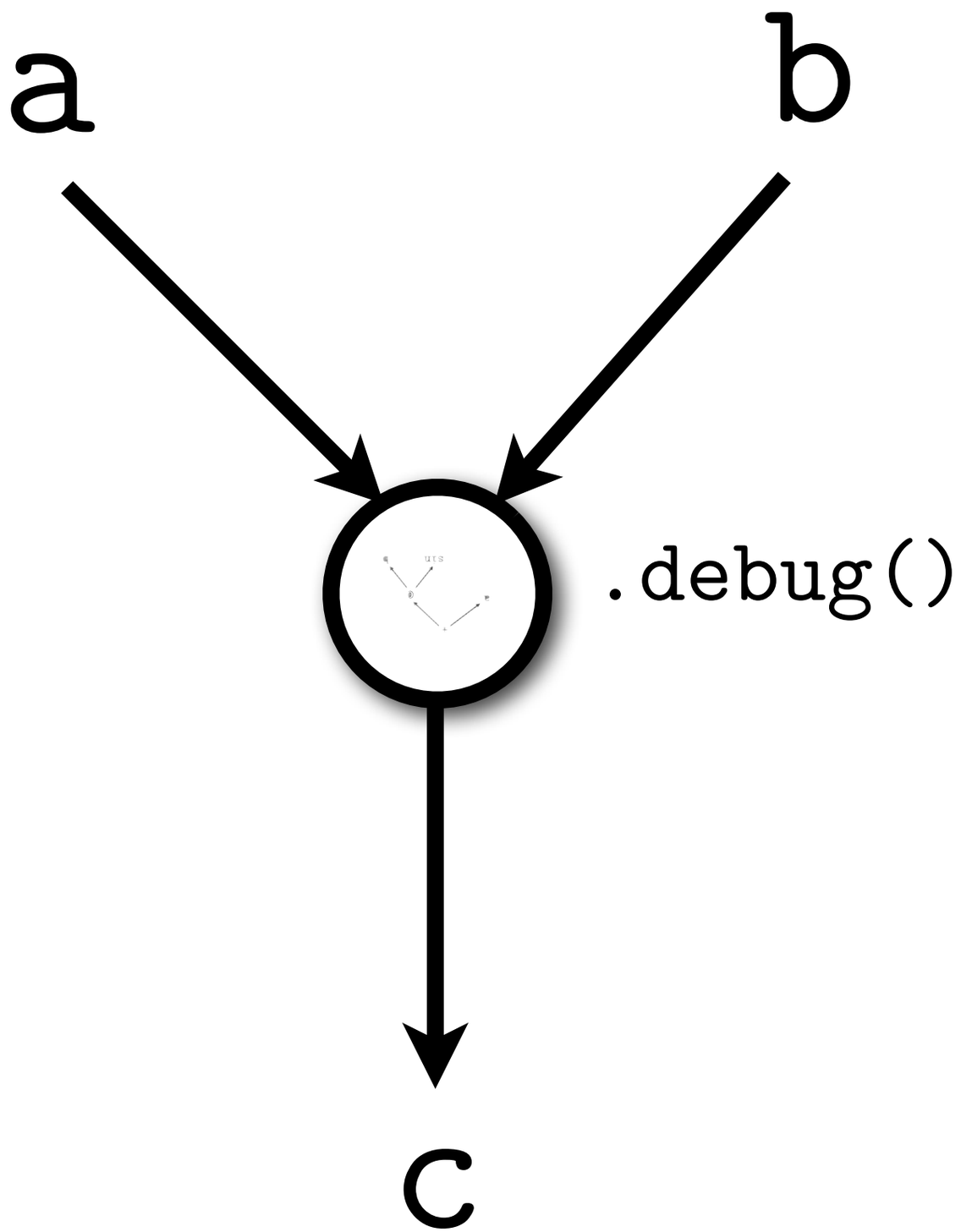

a
b
c

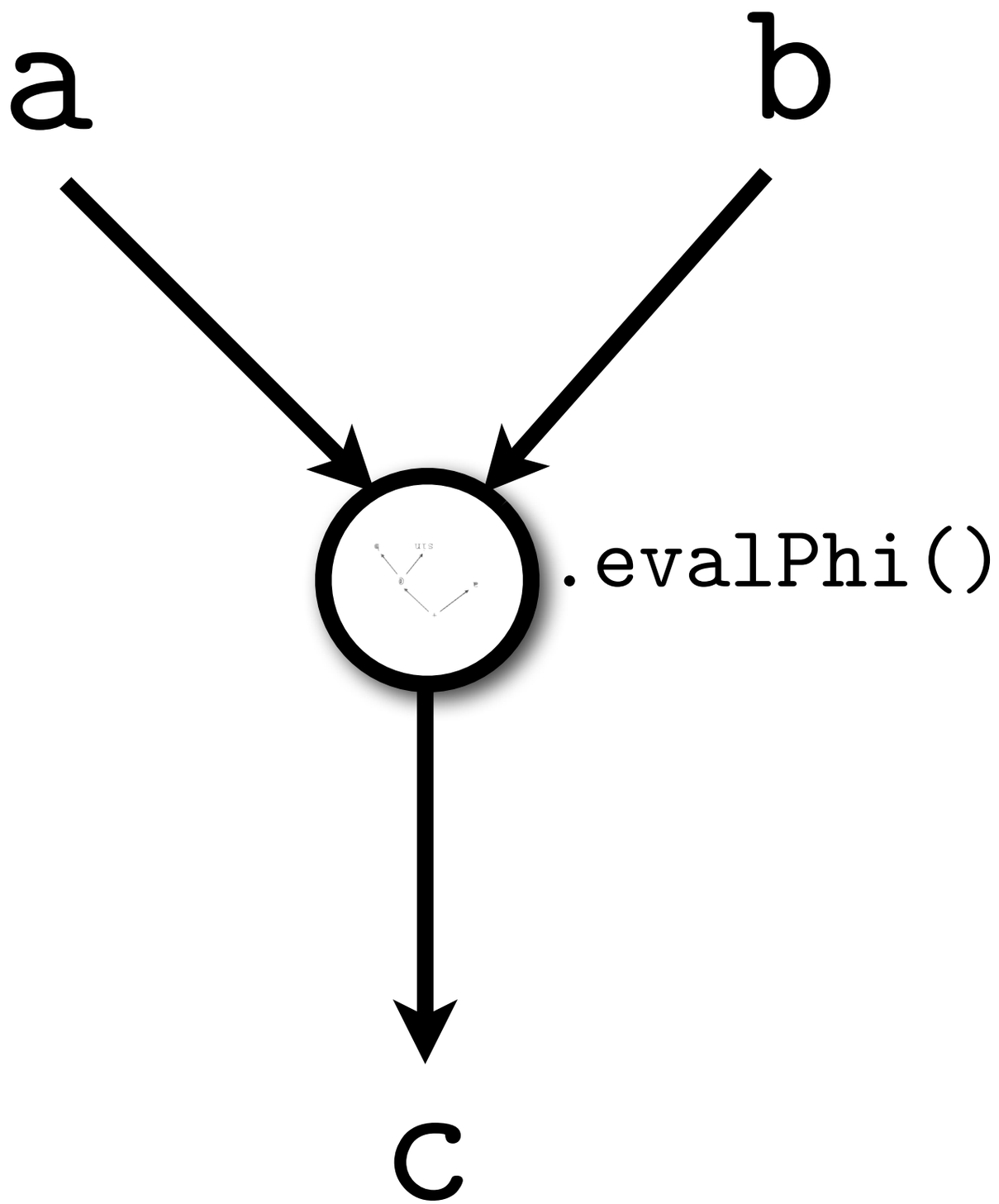

a
b
.execute()
c

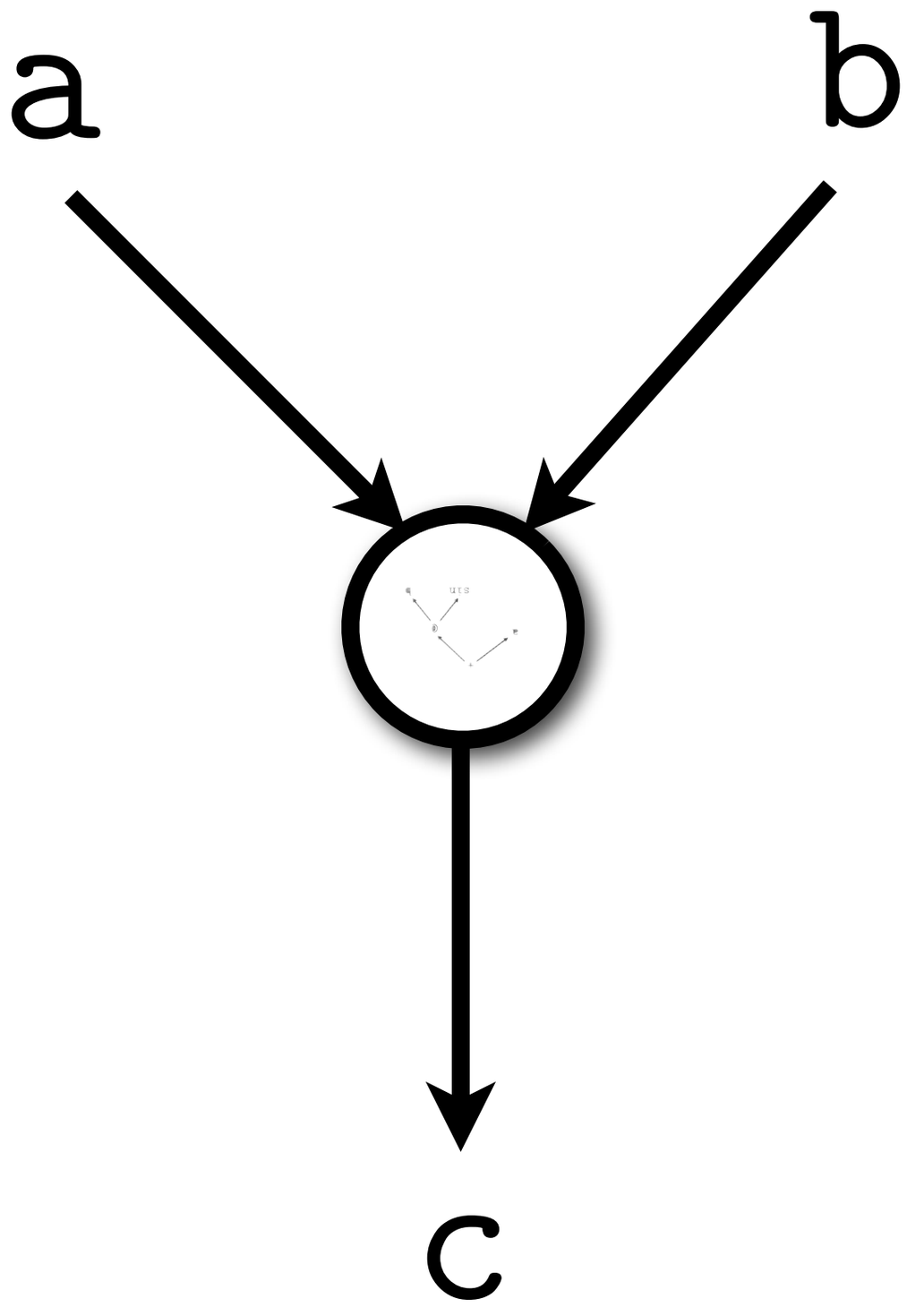

a
b
.evalCPU()
c


a
b
.evalMulti()
c


a
b
.evalGPU()
c


a
b
.debug()
c


a
b
.evalPhi()
c


a
b
c

For example...

```
c <<= a + sin(b); ⇒
```

Thread 1

```
Field::const_iterator ia1 = a1.begin();
Field::const_iterator ib1 = b1.begin();
for(Field::iterator ic1 = c1.begin();
    ic1 != c1.end();
    ++ic1, ++ia1, ++ib1) {
  *ic1 = *ia1 + sin(*ib1);
};
```

⋮

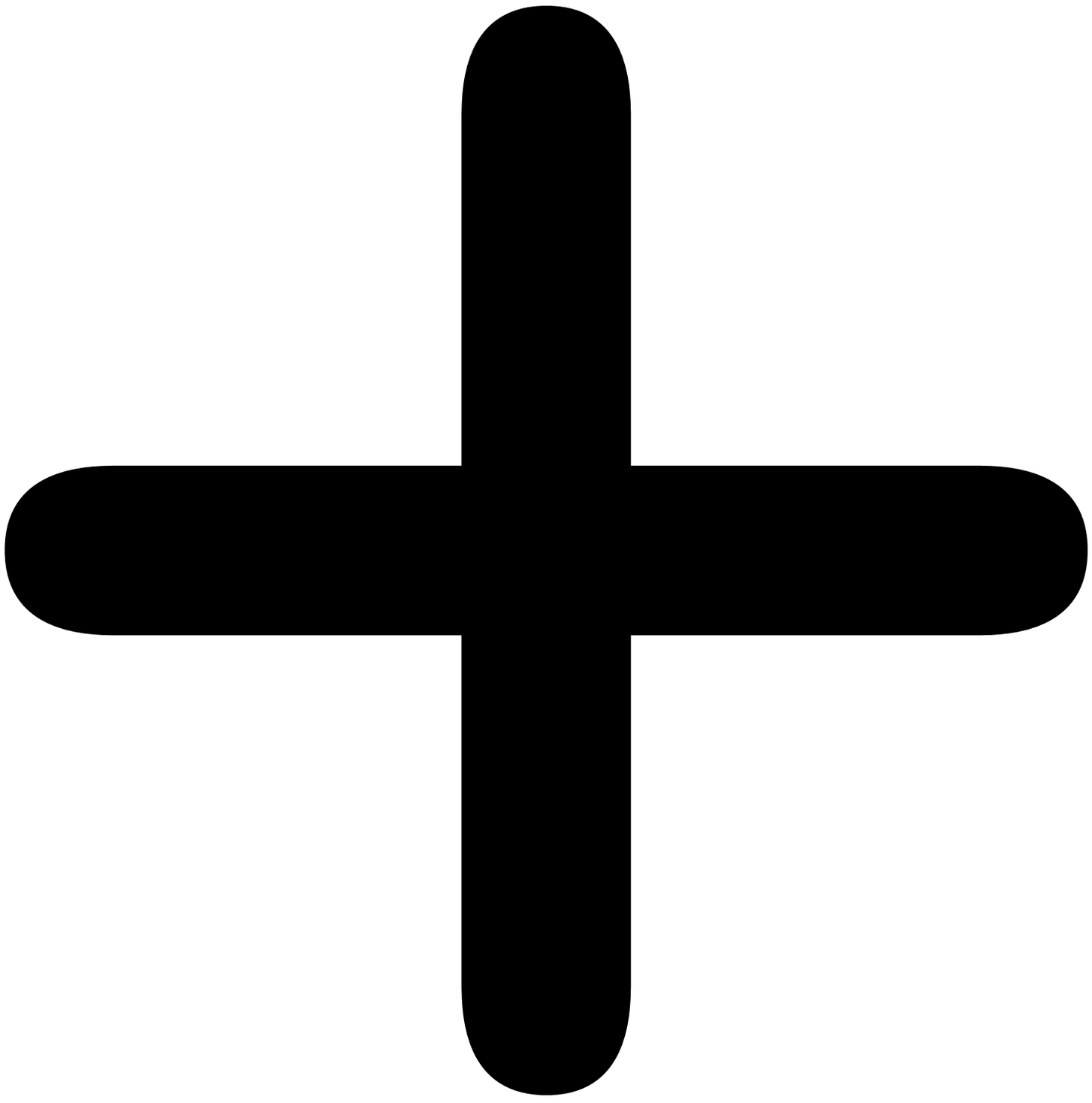
```
…
```

Thread *n*

```
Field::const_iterator ian = an.begin();
Field::const_iterator ibn = bn.begin();
for(Field::iterator icn = cn.begin();
    icn != cn.end();
    ++icn, ++ian, ++ibn) {
  *icn = *ian + sin(*ibn);
};
```

Under the hood

```
f <<= x + y ;
```



```
namespace SpatialOps {
   template<typename CurrentMode, typename Operand1, typename Operand2>
    struct SumOp;
   template<typename Operand1, typename Operand2>
    struct SumOp<Initial, Operand1, Operand2> {
      public:
       SumOp<SeqWalk,
             typename Operand1::SeqWalkType,
             typename Operand2::SeqWalkType> typedef SeqWalkType;

       #ifdef ENABLE_THREADS
          SumOp<Resize,
                typename Operand1::ResizeType,
                typename Operand2::ResizeType> typedef ResizeType;
       #endif
       /* ENABLE_THREADS */

       #ifdef __CUDACC__
          SumOp<GPUWalk,
                typename Operand1::GPUWalkType,
          typename Operand2::GPUWalkType> typedef GPUWalkType;
#endif
/* __CUDACC__ */

SumOp(Operand1 const & operand1, Operand2 const & operand2)
: operand1_(operand1), operand2_(operand2)
{}

inline GhostData ghosts_with_bc(void) const {
   return min(operand1_.ghosts_with_bc(), operand2_.ghosts_with_bc());
}

inline GhostData ghosts_without_bc(void) const {
   return min(operand1_.ghosts_without_bc(), operand2_.ghosts_without_bc());
}

inline bool has_extents(void) const {
   return (operand1_.has_extents() || operand2_.has_extents());
}

inline IntVec extents(void) const {
   #ifndef NDEBUG
```

```
            const & arg1,
            NeboExpression<SubExpr2, FieldType> const & arg2) {
    SumOp<Initial, SubExpr1, SubExpr2> typedef ReturnType;

    NeboExpression<ReturnType, FieldType> typedef ReturnTerm;

    return ReturnTerm(ReturnType(arg1.expr(), arg2.expr()));
 }

/* SingleValueExpr X SingleValue */
template<typename SubExpr1, typename T>
 inline NeboSingleValueExpression<SumOp<Initial,
                                        SubExpr1,
                                        NeboConstSingleValueField<Initial,
                                                                  T> >,
                                  T> operator +(NeboSingleValueExpression<SubExpr1,
                                                                          T>
                                                const & arg1,
                                                SpatialOps::SpatialField<SpatialOps::
                                                                         SingleValue,
                                                                         T>
                                                const & arg2) {
    SumOp<Initial, SubExpr1, NeboConstSingleValueField<Initial, T> >
    typedef ReturnType;

    NeboSingleValueExpression<ReturnType, T> typedef ReturnTerm;

    return ReturnTerm(ReturnType(arg1.expr(),
                                 NeboConstSingleValueField<Initial, T>(arg2)));
 }

/* SingleValueExpr X SingleValueExpr */
template<typename SubExpr1, typename SubExpr2, typename T>
 inline NeboSingleValueExpression<SumOp<Initial, SubExpr1, SubExpr2>, T>
 operator +(NeboSingleValueExpression<SubExpr1, T> const & arg1,
            NeboSingleValueExpression<SubExpr2, T> const & arg2) {
    SumOp<Initial, SubExpr1, SubExpr2> typedef ReturnType;

    NeboSingleValueExpression<ReturnType, T> typedef ReturnTerm;

    return ReturnTerm(ReturnType(arg1.expr(), arg2.expr()));
 };
```

```
template<typename FieldType>
 inline NeboExpression<SumOp<Initial,
                             NeboConstField<Initial,
                                            typename NeboFieldCheck<typename
                                                                    FieldType::
                                                                    field_type,
                                                                    FieldType>::
                                            Result>,
                             NeboConstField<Initial,
                                            typename NeboFieldCheck<typename
                                                                    FieldType::
                                                                    field_type,
                                                                    FieldType>::
                                            Result> >,
                       FieldType> operator +(FieldType const & arg1,
                                             FieldType const & arg2) {
   SumOp<Initial,
         NeboConstField<Initial, FieldType>,
         NeboConstField<Initial, FieldType> > typedef ReturnType;

   NeboExpression<ReturnType, FieldType> typedef ReturnTerm;

   return ReturnTerm(ReturnType(NeboConstField<Initial, FieldType>(arg1),
                                NeboConstField<Initial, FieldType>(arg2)));
}
```

```
                                    NeboFieldCheck<typename
                                                   FieldType::
                                                   field_type,
                                                   FieldType>::
                                Result
```

```
template<typename Type1, typename Type2>
 struct NeboFieldCheck;

template<typename Type>
 struct NeboFieldCheck<Type, Type> { Type typedef Result; };
```

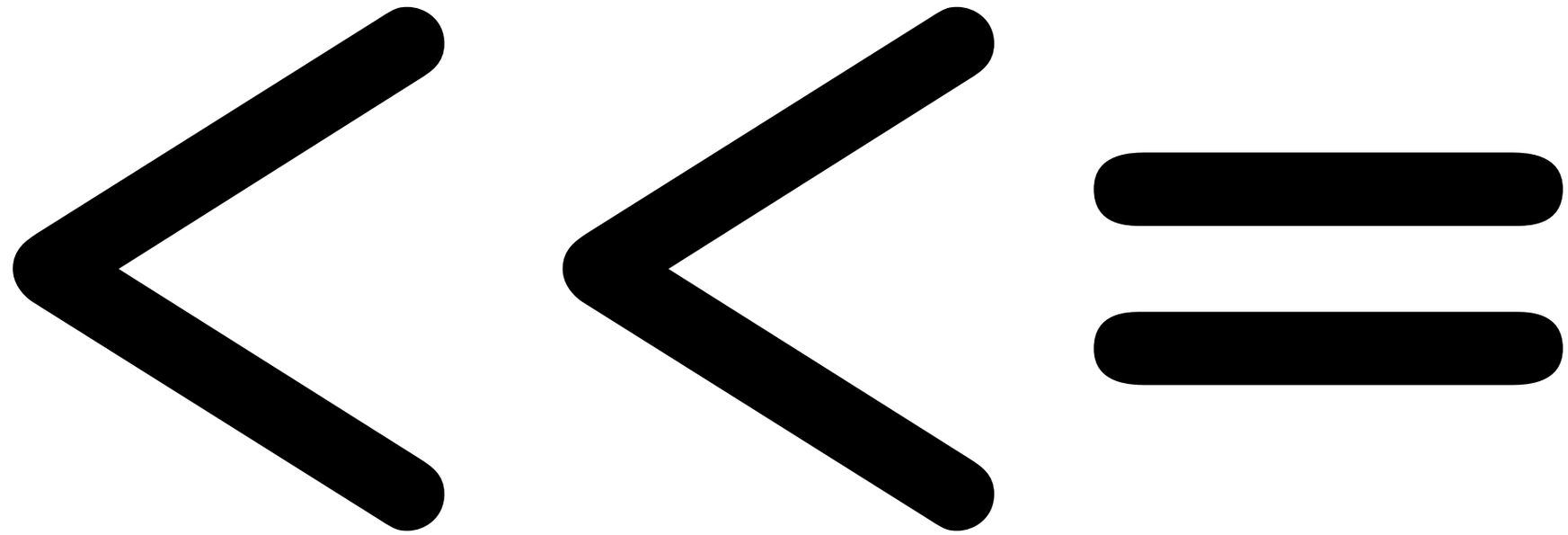
```
template<typename FieldType>
 inline NeboExpression<SumOp<Initial,
                             NeboConstField<Initial,
                                            typename NeboFieldCheck<typename
                                                                    FieldType::
                                                                    field_type,
                                                                    FieldType>::
                                            Result>,
                             NeboConstField<Initial,
                                            typename NeboFieldCheck<typename
                                                                    FieldType::
                                                                    field_type,
                                                                    FieldType>::
                                            Result> >,
                       FieldType> operator +(FieldType const & arg1,
                                             FieldType const & arg2) {
   SumOp<Initial,
         NeboConstField<Initial, FieldType>,
         NeboConstField<Initial, FieldType> > typedef ReturnType;

   NeboExpression<ReturnType, FieldType> typedef ReturnTerm;

   return ReturnTerm(ReturnType(NeboConstField<Initial, FieldType>(arg1),
                                NeboConstField<Initial, FieldType>(arg2)));
}
```

```
NeboExpression<SumOp<Initial,
                     NeboConstField<Initial,
                                    typename NeboFieldCheck<typename
                                                            FieldType::
                                                            field_type,
                                                            FieldType>::
                                    Result>,
                     NeboConstField<Initial,
                                    typename NeboFieldCheck<typename
                                                            FieldType::
                                                            field_type,
                                                            FieldType>::
                                    Result> >,
FieldType>
```

```
NeboExpression<SumOp<Initial,
                     NeboConstField<Initial,
                                                                FieldType
                                                            >,
                     NeboConstField<Initial,
                                                                FieldType
                                                            > >,
FieldType>
```

```
NeboExpression<SumOp<
                    NeboConstField<
                                                                FieldType
                                                >,
                    NeboConstField<
                                                                FieldType
                                                > >,
FieldType>
```

```
f <<= x + y ;
```



```
template<typename FieldType>
 inline FieldType const & operator <<=(FieldType & lhs,
                                       typename FieldType::value_type
                                       const & rhs) {
    NeboScalar<Initial, typename FieldType::value_type> typedef RhsType;

    NeboField<Initial, FieldType>(lhs).template assign<RhsType>(true,
                                                                RhsType(rhs));

    return lhs;
 };

template<typename FieldType>
 inline FieldType const & operator <<=(FieldType & lhs,
                                       FieldType const & rhs) {
    NeboConstField<Initial, FieldType> typedef RhsType;

    NeboField<Initial, FieldType>(lhs).template assign<RhsType>(true,
                                                                RhsType(rhs));

    return lhs;
 };

template<typename RhsType, typename FieldType>
 inline FieldType const & operator <<=(FieldType & lhs,
                                       NeboExpression<RhsType, FieldType>
                                       const & rhs) {
    NeboField<Initial, FieldType>(lhs).template assign<RhsType>(true, rhs.expr());
```

```
      operator <<=(SpatialOps::SpatialField<SpatialOps::SingleValue, T> & lhs,
                   SpatialOps::SpatialField<SpatialOps::SingleValue, T> const &
                   rhs) {
         NeboConstSingleValueField<Initial, T> typedef RhsType;

         NeboField<Initial,
                   SpatialOps::SpatialField<SpatialOps::SingleValue, T> >(lhs).template

assign<RhsType>(true,

RhsType(rhs));

         return lhs;
      };

      template<typename RhsType, typename T>
       inline SpatialOps::SpatialField<SpatialOps::SingleValue, T> const &
       operator <<=(SpatialOps::SpatialField<SpatialOps::SingleValue, T> & lhs,
                    NeboSingleValueExpression<RhsType, T> const & rhs) {
          NeboField<Initial,
                    SpatialOps::SpatialField<SpatialOps::SingleValue, T> >(lhs).template

assign<RhsType>(true,

rhs.expr());

          return lhs;
       };
```

```
template<typename RhsType, typename FieldType>
 inline FieldType const & operator <<=(FieldType & lhs,
                                       NeboExpression<RhsType, FieldType>
                                       const & rhs) {
    NeboField<Initial, FieldType>(lhs).template assign<RhsType>(true, rhs.expr());

    return lhs;
};
```

```
NeboField<Initial, FieldType>(lhs).template assign<RhsType>(true, rhs.expr());
```

```
template<typename CurrentMode, typename FieldType>
 struct NeboField;

template<typename FieldType>
 struct NeboField<Initial, FieldType> {
   public:
    FieldType typedef field_type;

    NeboField<SeqWalk, FieldType> typedef SeqWalkType;

#ifdef ENABLE_THREADS
   NeboField<Resize, FieldType> typedef ResizeType;
#endif
/* ENABLE_THREADS */

#ifdef __CUDACC__
   NeboField<GPUWalk, FieldType> typedef GPUWalkType;
#endif
/* __CUDACC__ */
```

```
template<typename RhsType>
 inline void assign(bool const useGhost, RhsType rhs) {
          // ...
          if(gpu_ready()) {
             if(rhs.gpu_ready(gpu_device_index())) {
                gpu_assign<RhsType>(rhs,
                                    extents,
                                    ghosts,
                                    hasBC,
                                    limits);
             }
             else {
                // ...
                throw(std::runtime_error(msg.str()));
             };
          }
          else {
             if(cpu_ready()) {
                if(rhs.cpu_ready()) {
                   cpu_assign<RhsType>(rhs,
                                       extents,
                                       ghosts,
                                       hasBC,
                                       limits);
                }
                else {
                   // ...
                   throw(std::runtime_error(msg.str()));
                };
             }
             else {
                // ...
                throw(std::runtime_error(msg.str()));
             };
          }
       // ...
       cpu_assign<RhsType>(rhs, extents, ghosts, hasBC, limits)
}
```

```
if(gpu_ready()) {
   if(rhs.gpu_ready(gpu_device_index())) {
```

```
   if(cpu_ready()) {
      if(rhs.cpu_ready()) {
```

```
// sequential:
template<typename FieldType>
struct NeboField<SeqWalk, FieldType> {

// multicore:
template<typename FieldType>
struct NeboField<Resize, FieldType> {

// GPU
template<typename FieldType>
struct NeboField<GPUWalk, FieldType> {
```

```
template<typename Operand1, typename Operand2>
 struct SumOp<SeqWalk, Operand1, Operand2> {
   public:
    typename Operand1::value_type typedef value_type;

    SumOp(Operand1 const & operand1, Operand2 const & operand2)
    : operand1_(operand1), operand2_(operand2)
    {}

    inline value_type eval(int const x, int const y, int const z) const {
       return (operand1_.eval(x, y, z) + operand2_.eval(x, y, z));
    }

   private:
    Operand1 operand1_;

    Operand2 operand2_;
};
```

```
#ifdef __CUDACC__
   template<typename Operand1, typename Operand2>
    struct SumOp<GPUWalk, Operand1, Operand2> {
      public:
       typename Operand1::value_type typedef value_type;

       SumOp(Operand1 const & operand1, Operand2 const & operand2)
       : operand1_(operand1), operand2_(operand2)
       {}

       __device__ inline value_type eval(int const x,
                                         int const y,
                                         int const z) const {
          return (operand1_.eval(x, y, z) + operand2_.eval(x, y, z));
       }

      private:
       Operand1 operand1_;

       Operand2 operand2_;
   }
#endif
/* __CUDACC__ */;
```

How do I add a feature?

750 lines

Fulmar

C++ Template Metaprogramming

C++ Template

MetaMetaprogramming

```
a <<= b + c;
```

```
a <<= b + c;
```

```
template
 template
  template
   template
    template
     template
```

```
a <<= b + c;
```

```
   template
    template
     template
      template
       template
template
 template
  template
```

```
a <<= b + c;
```

```
    template
     template
      template
       template
        template
template
 template
  template
```

```
a <<= b + c;
```

metametaprogram

fulmar

metaprogram

template
template
template
template
template
template
template

a <<= b + c;

```
(build-binary-operator 'SumOp '+ (add-spaces 'operator '+))
```

```
(build-binary-operator 'SumOp '+ (add-spaces 'operator '+))
```

(+ about 500 lines of Scheme)

```
(build-binary-operator 'DivOp '/ (add-spaces 'operator '/))
```

https://github.com/cwearl/fulmar

Questions?

Latest: Travel hacks
Next: Self-inlining anonymous functions in C++
Prev: A pipelined, non-blocking, extensible web server in Scala
Rand: Relational shell programming

Lambda-style anonymous functions for C++ in less than 500 lines of code

[article index] [email me] [@mattmight] [+mattmight] [rss]

I prefer functional languages like Scheme, Scala or Haskell over C++ for most tasks. So, when I'm forced to program in C++, I write functional C++.

Fortunately, it's possible to add the most important ingredient in functional languages--lambda-style anonymous functions--to standard C++.

For example, the expression

```
lambda<int> (x) --> 3*x + 7
```

represents a function that multiplies by 3 and then adds 7. (The `<int>` means that the anonymous function returns an integer.)

These anonymous functions are first-class; that is, they can be assigned to variables, passed as parameters or returned from functions; for instance:

```
f = lambda<int> (x,y) --> x + y ;
```

Latest: Travel hacks
Next: A non-blocking lexing toolkit for Scala from regex derivatives
Prev: Lambda-style anonymous functions in C++
Rand: A Ph.D. thesis proposal is a contract

Self-inlining anonymous functions in C++

[article index] [email me] [@mattmight] [+mattmight] [rss]

In my last post, I explored a domain-specific embedded language for lambda-style anonymous functions in C++. Many liked the idea, but some quibbled about the potential run-time cost. Personally, I didn't think it was a large overhead, but some programmers aren't satisfied until the compiler outputs the assembly they would have written themselves.

To show that the overhead can be eliminated, I've created a second demo of anonymous functions. In this implementation, anonymous functions without free variables automatically inline themselves at their point of use, eliminating all run-time overhead.

For example, suppose you wanted to numerically integrate several functions repeatedly over different ranges. A sensible abstraction might involve an integration function that takes in a function pointer, a range and a number of partitions; for example:

```
double integrate (double (*f)(double), double a, double b, int n) ;
```

Latest: Travel hacks
Next: Logical literacy
Prev: 10 reasons Ph.D. students fail
Rand: HOWTO: Catalog a library with a $10 barcode scanner

Episode III: Self-inlining anonymous closures in C++

[article index] [email me] [@mattmight] [+mattmight] [rss]

C++ doesn't leap to mind when one thinks of pure, functional languages.

But, it should.

C++ template meta-programming is not just pure and functional by itself; meta-programming allows programmers to add functional features to C++.

In previous posts, I've explored

- how to add lambda-like lexical closures; and
- how to create self-inlining anonymous functions.

In this post, I'll combine these techniques to create a powerful, efficient hybrid: *self-inlining anonymous static closures*.

Like the lexical closures, static closures can have free variables (or really, free values), yet like the self-inlining anonymous functions, they have no run-time penalty either.

Latest: Travel hacks
Next: Fast vector-structs in Scheme from syntax-rules
Prev: Lazy-list-based streams in Scala
Rand: Parsing with derivatives (Yacc is dead: An update)

C++ templates: Creating a compile-time higher-order meta-programming language

[article index] [email me] [@mattmight] [+mattmight] [rss]

For the Halloween lecture in my advanced compilers class, I scare students with C++ template meta-programming.

To prove that C++ templates are Turing-complete, we constructed a higher-order functional programming language out of them.

The language supports higher-order functions, literal values (raw types), natural numbers (as types), booleans and conditionals.

Specifically, we made an SICP-like eval/apply interpeter out of templates.

This embedded language shows that any computable function can be computed at compile-time in C++.

Questions?