Functional Programming (in C++)

boostcon 2010

David Sankel
Sankel Software

Functional Programming

What? Why? How (in C++)?

- Algebraic Data Types : (), :=, ::=, $\otimes$, $\oplus$
- Functions : $a \to (b \to c)$
- Generic Programming

  : $(c \blacktriangleleft C) \to c \to int$
- Category Theory

  : monoid, monad, etc.

Algebraic Data Types

unit, primitive type, one value, denoted ()

Algebraic Data Types

unit, primitive type, one value, denoted ()

+

product, binary type operation,

denoted a $\otimes$ b, “a and b”, one of each

Algebraic Data Types

unit, primitive type, one value, denoted ()

+

product, binary type operation,

denoted a $\otimes$ b, "a and b", one of each

+

Sum, binary type operation,

Denoted a $\oplus$ b, "a or b",

one of one

Algebraic Data Types

unit, (), one value

product, a $\otimes$ b, “a and b”

sum, a $\oplus$ b, “a or b”

Algebraic Data Types

unit, (), one value

product, a $\otimes$ b, “a and b”

sum, a $\oplus$ b, “a or b”

is the same as, a := b

is implemented with, a ::= b

Examples:

true ::= ()
false ::= ()

bool := true $\oplus$ false

true is implemented with unit.
false is implemented with unit.
A bool value is the same as a true value or a false value.

Examples:

$Z ::= ()$

$N ::= Z \oplus N$

Z is implemented with unit.
A N value is the same as a Z value or an N value.

Examples:

$Z ::= ()$

$N := Z \oplus N$

$z \in Z$

$N_0 = (0,z)$

$N_1 = (1,N_0) = (1,(0,z))$

$N_2 = (1,N_1) = (1,(1,(0,z)))$

Type Functions

Add parameter on left side of ::= or := symbol that can be used on the right.

$[] ::= ()$

$L\ a := [] \oplus (a \otimes L\ a)$

A value of "L of a" is either a value of [] or (a value of a and a value of "L of a").

Type Functions

$[] ::= ()$

$L\ a := [] \oplus (a \otimes L\ a)$

Say a_i is a value of type a. and e is the value of type [].

$(0,e)$

$(1,(a_i,(0,e))$

$(1,(a_j,(1,(a_i,(0,e))$

Type Functions

$T\ a ::= (T\ a \otimes T\ a) \oplus a$

Binary tree with values of type 'a' at the leaves.

Algebraic Data Types

- Abstract

- Simple ((), :=, ::=, $\otimes$, $\oplus$)

- Powerful

- ...

Algebraic Data Types (<u>in C++!</u>)

<u>Our critera for functional concepts in C++</u>

- No (minimal) syntax sugar, it scares away the new bees.
- Mixes well with typical C++.
- No copycatting other languages and their limitations..

Algebraic Data Types (in C++!)

Easy ones

- Unit (): use boost::mpl::void_

- New unit types: T ::= ()
  struct T {};

- "is the same as": a ::= b
  typedef b a;

Algebraic Data Types (in C++!)

Product Types

- $z ::= a \otimes b \otimes c$. struct. Named accessors
- $a \otimes b$. boost.fusion.vector. Access by index
- boost.fusion.map. Both accessor methods.

Algebraic Data Types (in C++!)

Sum Types

- Can use enum when underlying types
  - are units, and
  - aren't used elsewhere.
- Can use a product type with an index.
  - common and error prone
- Can use polymorphic base class.
  - not really nice syntax/error prone

Algebraic Data Types (in C++!)

Sum Types

- boost.variant (best option!)
  - arbitrary underlying types
  - small syntax overhead
  - access by index or type

$$a \oplus b \oplus c$$

```
boost::variant<a,b,c>
```

Algebraic Data Types (<u>in C++!</u>)

"<u>is implemented with</u>", ::=, wrap it in a struct

R01 ::= double

```
struct R01
{ explicit R01( const double impl__ )
                    : impl( impl__ ) {}
  double impl;
};
```

- accessor functions (invariant guaranteed)
- internal impl access function (invariant requirement)

Algebraic Data Types (in C++!)

Type Functions (:= style)

Use "type function trait" from boost.mpl.

none ::= (), Op a := none ⊕ a

```
struct none{};
template< typename a >
struct Op {
  typedef boost::variant<none,a>
                    type;
};
```

Algebraic Data Types (in C++!)

Type Functions (::= style)

Use a wrapper template struct

none ::= (), Op a ::= none $\oplus$ a

```cpp
struct none{};
template< typename a >
struct Op {
  explicit Op( boost::variant<none,a>);
  //...
  boost::variant<none,a> impl;
};
```

Algebraic Data Types (in C++!)

Recursive Types

- sum types: use make_recursive_variant
- product types: use make_recursive_variant (see paper for details)

What the heck is a recursive product type?

$$S\ a ::= a \otimes (S\ a)$$

- Seems like nonsense!?

What the heck is a recursive product type?

$$S\ a ::= a \otimes (S\ a)$$

- Seems like nonsense!?
- Nope
- Think of the recursion as being a computation of type (S a)
- ...

Laziness (in C++!)

How can we represent a computation of a value in C++?

Laziness (in C++!)

How can we represent a computation of a value in C++?

- A 0 argument function.

Laziness (in C++!)

How can we represent a computation of a value in C++?

- A 0 argument function.

```
template< typename a>
struct lazy
{ typedef boost::function< a () >
              type;
};
```

What the heck is a recursive product type?

$$S\ a ::= a \otimes (S\ a)$$

- Streams of type a. See paper for a direct implementation.

Functions

$$f : A \to B$$

A set S of pairs (a,b) where $a \in A$ $b \in B$. For every $a \in A$, there exists exactly one corresponding pair in S.

What is a C++ function?

```
bool f(int);
```

What is a C++ function?

bool f(int); => $int \rightarrow bool$

What is a C++ function?

bool f(int); => int→ bool

What about multiple arguments?

What is a C++ function?

Lets try another case.

```
bool f2(int, int);
```

What is a C++ function?

Lets try another case.

bool f2(int, int); => $(int \otimes int) \rightarrow bool$

- Use our product type operator!
- c++-function-tuples "f2 (2,13)"
- Still something missing...

What is a C++ function?

```
bool f3(int, int)
{  ++someglobalvar;
  return true; }
```

- $(\text{int} \otimes \text{int}) \rightarrow \text{bool}$ doesn't work!
- Consider the corresponding set.

What is a C++ function?

```
bool f3(int, int);
{  ++someglobalvar;
  return true; }
```

$(int \otimes int \otimes World) \rightarrow (World \otimes bool)$

- $(int \otimes int) \rightarrow bool$ doesn't work!
- Consider the corresponding set
- Introduce new parameter and return value, World...

What is a C++ function?

Translation

$$R\ f(A_1, A_2, \ldots, A_n);$$

^^^

$$(A_1 \otimes A_2 \ldots A_n \otimes World) \rightarrow (World \otimes R)$$

Currying

$$a \otimes b \to c$$

to

$$a \to (b \to c)$$

- function returns a function (convenient)
- $\to$ is right associative
- Works with any function where the domain is a product.

What is a C++ function?

Translation

$$R\ f(A_1, A_2, \ldots, A_n);$$

$$(A_1 \otimes A_2 \ldots A_n \otimes \widehat{\text{World}}) \rightarrow (\text{World} \otimes R)$$

$$A_1 \rightarrow A_2 \ldots A_n \rightarrow \widehat{\text{World}} \rightarrow (\text{World} \otimes R)$$

Introducing IO

$$io\ a := World \rightarrow (World \otimes a)$$

- Simple type function
- We can actually implement io a in C++.

```
template< typename a >
struct io
{
  typedef boost::function<a ()> type; }
```

What is a C++ function?

Translation

$$R\ f(A_1, A_2, \ldots, A_n);$$

c++-function-tuple

$$(A_1 \otimes A_2 \ldots A_n \otimes World) \to (World \otimes R)$$

curry

$$A_1 \to A_2 \ldots A_n \to World \to (World \otimes R)$$

io

$$A_1 \to A_2 \ldots A_n \to io\ R$$

Functions (in C++)

gfp library (netsuperbrain.com/gfp)

- gfp::ciof (curried io function)
  - Converts a c++ function pointer into a function as we formulated.
- gfp::cfunc (curried function)
  - cfunc<a,b,c>::type => function<function<c (b)> a>

Generic Programming

Intuition:

- "an empty thing", could refer to several things, but not all.

- "empty", a property of many things, but not all.

Generic Programming

Formulation:

- "an empty thing"
  - requires a type to be concrete
  - type (a◄Emptiable) → a
  - read ◄ as "has a profile in"
  - a function from types to values
- "empty"
    - type (a◄HasEmpty) → (a → bool)

Generic Programming

Formulation:

- Emptiable, a type class
  - A typeclass is a set of pairs (a,p)
    - a is a ::= type or type function
    - p is a profile that fits certain patterns and laws
- Our restrictions
  - At most one pair per type
  - Profile is a simple value

Generic Programming

HasEmpty:

- An element in HasEmpty
  - (std::vector<int>, z) where
    bool z(std::vector & z)
      { return z.empty(); }
- To implement "empty" we need to get the corresponding value (function) from a type in HasEmpty.

Generic Programming

HasEmpty:

- An call to “empty”:

```
std::vector<int> v;
empty<std:::vector<int>>::profile()( v );
```

- looks up a profile given a type.
- but wait…

Generic Programming

<u>HasEmpty:</u>

```
empty<std::vector<int>>::profile()( v );
```

- Passing v, of known type, makes the explicit type redundant. ☹
- empty cannot be passed to functions. ☹

Generic Programming (in C++!)

Polymorphic Functions:

- Idea is to infer the type arguments from the value arguments.

```
struct Empty
{ typedef bool result__type;
   bool operator()( std::vector<int> )
   //... one operator() for each type.
} empty;
```

- empty(v); //infers the type from v.

Generic Programming (in C++!)

Polymorphic Values:

- Same type inference trick doesn't work for values.

- Introduce resolve:

```
resolve<std::vector<int>>( emptyThing );
```

- Polymorphic values must follow a certain trait . . .

Generic Programming (in C++!)

Polymorphic Values:

```
struct EmptyThing
{ template<typename T>
  struct result;
  //...
  result<this_type(vector<int>*)>::type
  operator()( vector<int>* dummy );
  //...
} emptyThing;
```

Generic Programming (in C++!)

Polymorphic Values & Functions:

- Covers all generics we care about
- Cannot be extended to support new types without modification of underlying code.
- No relation between related polymorphic values and functions.

Generic Programming (in C++!)

Polymorphic Classes:

- Extendable collections of polymorphic values and functions.
- Use partial template instantiation to select supported type.
- The polymorphic entities select appropriate instantiation when used.

Category Theory

- deals abstractly with mathematical structures and relations between them.

- Gives some guidance as to what to do with generics.

- Very powerful and expressive.

Category Theory

Monoids

$$A$$

$$0 \in A$$

$$+ : A \to A \to A$$

A, 0, and + form a monoid when + is associative and 0 is an identity for +.

Category Theory (Monoids)

There are lots of monoids!

- int, +, 0 : sum monoid
- bool, &&, true: all monoid
- string, concat, "": string monoid
- a → m: function monoid
  - m monoid
  - forwards monoid operations to results.
- io m: io monoid. Similar to function monoid.

Category Theory (Monoids)

Quick example:

- header : Message → string
- contents : Message → string

```
gfp::cfunc<Message,string>
payload = gfp::mplus( header )( contents );
```

- called point free (pointless) programming!.

Category Theory

Much much more:

- functor/pointed: functions, containers...
- idiom (applicative functors): FRP, streams
- monad: arbitrary computations
- foldable: compress collections

Functional Programming (in C++!)

FP Benefits:

- Cleaner design → Cleaner code
- less code/static types → Less bugs
- Powerful tools

FP (in C++) Benefits:

- No need to switch languages
- Integrates well
- Easy to use (no special syntax)
- Highly Capable