Multiparadigm programming: Novel devices for implementing functional and logic programming constructs in C++

Brian McNamara

Why multiparadigm?

Multiparadigm approach proven useful in a number of application domains

Choose paradigm that is best suited to the individual task at hand

Research problems

Functional => OO

- Lack of expressiveness
- High complexity (polymorphism)

Logic => OO

- Difficulty integrating data/type systems
- Mitigating control flow mismatch

Both Functional & Logic => OO

- Preserving syntax
- Embedding language

Thesis

Constructs for functional and logic programming can be smoothly integrated into an existing object-oriented language. We demonstrate this in the context of C++, and show that the resulting multiparadigm language has useful applications in real-world domains.

Thesis

Constructs for functional and logic programming can be smoothly integrated into an existing object-oriented language. We demonstrate this in the context of C++, and show that the resulting multiparadigm language has useful applications in real-world domains.

Thesis

Constructs for functional and logic programming can be smoothly integrated into an existing object-oriented language. We demonstrate this in the context of C++, and show that the resulting multiparadigm language has useful applications in real-world domains.

Thesis

Constructs for functional and logic programming can be smoothly integrated into an existing object-oriented language. We demonstrate this in the context of C++, and show that the resulting multiparadigm language has useful applications in real-world domains.

Talk outline

Smooth integration

- FC++
- LC++
- Limitations

Reusable lessons

Applications / Impact

Talk outline

Smooth integration

- FC++
- LC++
- Limitations

Reusable lessons

Applications / Impact

Implementing higher-order polymorphic functions

Haskell (a functional language)

```
map twoTimes (enumFromTo 1 10)
```

Implementing higher-order polymorphic functions

Haskell (a functional language)

```
map twoTimes (enumFromTo 1 10)
```

Läufer (prior approach in C++)

```
Map<int,int>()(
  Fun1<int,int>( new TwoTimes<int>() ),
  enumFromTo(1,10)
  )
```

Implementing higher-order polymorphic functions

Haskell (a functional language)

```
map twoTimes (enumFromTo 1 10)
```

Läufer (prior approach in C++)

```
Map<int,int>()(
  Fun1<int,int>( new TwoTimes<int>() ),
  enumFromTo(1,10)
  )
```

"...the type information required in more complex applications of the framework is likely to get out of hand, especially when higher numbers of arguments are involved."

FC++ functoids and Sigs

```
struct Map {
   template <class F, class L>
   struct Sig : public FunType<F,L,
     List< RT<F,L::ElementType>::ResultType > >{};

   template <class F, class L>
   Sig<F,L>::ResultType
   operator()( const F& f, const L& l ) const {
     if( null(l) ) return NIL;
     else return cons( f( head(l) ),
                      thunk2( Map(), f, tail(l) ) );
   }
} map;
F = TwoTimes        L = List<int>
Sig<F,L>::ResultType = List<int>
```

Use of FC++ functoids

Haskell (a functional language)

```
map twoTimes (enumFromTo 1 10)
```

Läufer (prior approach in C++)

```
Map<int,int>()(
  Fun1<int,int>( new TwoTimes<int>() ),
  enumFromTo(1,10)
  )
```

FC++ (our approach)

```
map( twoTimes, enumFromTo(1,10) )
```

FC++ lambda

```
// basic example            \x -> x-3
lambda(X)[ minus[X,3] ]
```

FC++ lambda

```
// basic example             \x -> x-3
lambda(X)[ minus[X,3] ]

// infix syntax              \x y -> 3*x + y
lambda(X,Y)[ (3 %multiplies% X) %plus% Y]
```

FC++ lambda

```
// basic example              \x -> x-3
lambda(X)[ minus[X,3] ]

// infix syntax               \x y -> 3*x + y
lambda(X,Y)[ (3 %multiplies% X) %plus% Y]

// letrec example             factorial
lambda(X)[ letrec[ F ==
  lambda(Y)[if1[Y %equal% 0,
                1,
                Y %multiplies% F[Y%minus%1]]
].in[ F[X] ] ]
```

FC++ lambda versus Boost Lambda Library

```
3*_1 + _2
lambda(X,Y)[ (3 %multiplies% X) %plus% Y]

bind(foo,_1) + bind(bar,_2)
lambda(X,Y)[ foo[X] %plus% bar[Y] ]
```

FC++ can handle nested lambdas

FC++ can do recursion

FC++ uses familiar lambda syntax

Other features

Sugars: currying, monads and comprehensions

Expressiveness: indirect functoids, subtype polymorphism, interface to normal C++ functions and methods

Pragmatics/embedding: static analysis/diagnostics, lazy lists (even/odd), memory management, strict lists, library, performance/optimizations

Talk outline

Smooth integration

- FC++
- LC++
- Limitations

Reusable lessons

Applications / Impact

Other approaches to logic programming in OO languages

J/MP, SOUL, MPC++

```
// defining facts
public static Relation
father( +String x, +String y ) {
   return (eq(x,"homer") && eq(y,"bart"))
       || (eq(x,"grandpa") && eq(y,"homer"));
}
```

Other approaches to logic programming in OO languages

J/MP, SOUL, MPC++

```
// defining facts
public static Relation
father( +String x, +String y ) {
   return (eq(x,"homer") && eq(y,"bart"))
       || (eq(x,"grandpa") && eq(y,"homer"));
}

// example query
ancestor( x, "bart" ).apply( new Relation(){
   public static boolean apply(Relation w) {
      System.out.println( x );
      return false;
   }
} )
```

LC++ approach

```
FUN2(father,string,string);
lassert( father( "homer", "bart" ) );
lassert( father( "grandpa", "homer" ) );
```

LC++ approach

```
FUN2(father,string,string);
lassert( father( "homer", "bart" ) );
lassert( father( "grandpa", "homer" ) );

List<IE> l = lquery( ancestor(X,"bart") );
// process elements of “l” as seen fit...
// each result is produced “on demand”
```

Results as a lazy list

```
// example to generate all natural numbers
lassert( nat(0) );
lassert( nat(X) -= nat(Y) && X.is(plus,Y,1) );
```

Results as a lazy list

```
// example to generate all natural numbers
lassert( nat(0) );
lassert( nat(X) -= nat(Y) && X.is(plus,Y,1) );
...
List<IE> l = lquery( nat(X) );
```

Results as a lazy list

```
// example to generate all natural numbers
lassert( nat(0) );
lassert( nat(X) -= nat(Y) && X.is(plus,Y,1) );
...
List<IE> l = lquery( nat(X) );
// print only first 3 results now
for( int i=0; i<3; ++i ) {
   print( head(l) );
   l = tail(l);
}
...
// perhaps process more results later
```

Talk outline

Smooth integration

- FC++
- LC++
- Limitations

Reusable lessons

Applications / Impact

Limitations

- Sig specification complexity
- Error messages
- Polymorphic function variables
- Max number of arguments
- Syntax / operator limitations
- No automatic lambda / closures

Talk outline

Smooth integration

- FC++
- LC++
- Limitations

Reusable lessons

Applications / Impact

Infix syntax in FC++

With *one* ad-hoc-overloadable operator…

`x ^plus^ y` means `plus(x,y)`

Infix syntax in FC++

With *one* ad-hoc-overloadable operator…

`x ^plus^ y` means `plus(x,y)`

Actually parses as `(x^plus) ^ y`

Infix syntax in FC++

With *one* ad-hoc-overloadable operator…

`x ^plus^ y` means `plus(x,y)`

Actually parses as `(x^plus) ^ y`

```
template <class X, class F>
Tmp<X,Full2<F> >
operator^( X x, Full2<F> f );

template <class X, class F, class Y>
RT<F,X,Y>::ResultType
operator^( Tmp<X,Full2<F> > tmp, Y y );
```

Infix without overloading

With *two* fresh user-definable infix operators…

`` x `plus' y ``	backquote & quote
`x \plus/ y`	slash & backslash

```
-- using Haskell notation
data Tmp x f = Tmp x f

(\) :: x -> (x->y->r) -> Tmp x (x->y->r)
x \ f = Tmp x f

(/) :: Tmp x (x->y->r) -> y -> r
(Tmp x f) / y = f x y
```

Lazy lists in an eager language

Two representations: *odd* and *even*

odd:	easy to encode	sometimes too eager
even:	harder to encode	sufficiently lazy

Wadler, Taha, and MacQueen proposed new language constructs

We found an alternative solution

We used C++'s *ad-hoc overloading*, but in fact *bounded parametric polymorphism* is sufficient

Even/odd lists

```
take 3 (map sqrt (countdown 2.0))
```

`[1.414, 1.000, 0.000]` *or* `Exception: sqrt -1.0`

Even/odd lists

```
take 3 (map sqrt (countdown 2.0))
```

`[1.414, 1.000, 0.000]` *or* `Exception: sqrt -1.0`

```
data TH a =  () -> a

data OL a =  nil | cons a (EL a)
data EL a =  TH (OL a)
```

Even/odd lists

```
take 3 (map sqrt (countdown 2.0))
[1.414, 1.000, 0.000] or Exception: sqrt -1.0

data TH a =  () -> a

data OL a =  nil | cons a (EL a)
data EL a =  TH (OL a)

force  :: TH a -> a
thunk2 :: (x->y->r) -> x -> y -> TH r
delay  :: a -> TH a
```

Solution with odd lists

```
data OL a =  nil | cons a (EL a)
data EL a =  TH (OL a)

take :: Int -> OL a -> OL a
take n l =
   if (n=0) || (null l)
   then nil
   else cons (head l)
              (thunk2 take (n-1) (force (tail l)))

take 3 (map sqrt (countdown 2.0))
Exception: sqrt -1.0
```

Solution with even lists

```
data OL a =  nil | cons a (EL a)
data EL a =  TH (OL a)
take :: Int -> EL a -> EL a
take  n l = if n=0
            then delay  nil
            else thunk2 take_ n (force l)
take_ n l = if null l
            then nil
            else cons (head l)
                      (take (n-1) (tail l))
take 3 (map sqrt (countdown 2.0))
[1.414, 1.000, 0.000]
```

"Alas, our definition has nearly doubled in size, and halved in perspicuity."

Our idea: bounded polymorphism

```
class (ListLike e e o, ListLike o e o)=>
      ListLike l e o | l -> e o where
   nil    :: l v
   cons   :: v -> e v -> o v
   head   :: l v -> v
   tail   :: l v -> e v
   null   :: l v -> Bool
instance ListLike EL EL OL where
   -- even list implementation ...
instance ListLike OL EL OL where
   -- odd list implementation ...
```

Solution using **ListLike** interface

```
take :: (ListLike l e o)=>
        Int -> l a -> o a
take n l =
   if (n=0) || (null l)
   then nil
   else cons (head l)
             (thunk2 take (n-1) (tail l))
take 3 (map sqrt (countdown 2.0))
[1.414, 1.000, 0.000]
```

Since this **take** is polymorphic, it plays *both* roles (**take**,**take_**) from the original even solution

An unexpected bonus

```
data OL a =  nil | cons a (EL a)
data EL a =  TH (OL a)

instance ListLike EL EL OL where ...
instance ListLike OL EL OL where ...

data SL a =  nil | cons a (SL a)

instance ListLike SL SL SL where
   -- strict-list implementation...
```

Summing up

Problem: odd lists are sometimes too eager, even lists are hard to code

Wadler, Taha, and MacQueen propose new language constructs

Bounded parametric polymorphism provides an alternative solution, best of all worlds

Talk outline

Smooth integration

- FC++
- LC++
- Limitations

Reusable lessons

Applications / Impact

Applications

Design pattern implementations

- Command, Observer, Virtual proxy, Decorator, Builder

Client applications

- XR, BSFC++, MPC++, monadic parser combinator library

Impact

Influenced Boost libraries

- lambda
- spirit / phoenix
- optional / variant

Future

- `result_of`
- `auto/typeof`
- concepts

Boost comments

I'd be very much interested in collaborating with you. ... FC++ has been very influential in my style of programming. I'd be honored to help make FC++ a part of boost someday.

Joel de Guzman, author or spirit/phoenix

It is clear that C++ has some form of functional programming in its future and I'm glad we have people of this caliber working on it.

Jon Kalb

Brian is a great asset to the C++ community at large.

Gennadiy Rozental

FC++ is a very inspirational library and that I have a great deal of respect for your work... bring FC++ into boost as your time may free up later on... I hope you will find renewed energy for the project over time.

Mat Marcus

Conclusion

Hidden slides

Example logic application

```
lassert( compatible( someVideoCard, someOS ) );
...
lassert( price( someComponent, 150 ) );
...
lassert( powerConsumption( someComponent, 5 ) );
...
lassert( computer( Components, Video, OS, Price )
           -= totalPrice(...) && checkCompat(...) );
...
lquery( computer(chosenComps,chosenVid,chosenOS,P) );
lquery( computer(chosenComps,Vid,chosenOS,P) );
```

Currying

```
f(x,y,z)    // normal call

f(x,y)      // \z   -> f x y z
f(x)        // \y z -> f x y z

f(x,_,z)    // \y   -> f x y z
f(_,_,z)    // \x y -> f x y z
// etc
```