Modern Functional Programming in C++

David Sankel
Sankel Software
Spencerport, New York 14559
david@sankelsoftware.com

ABSTRACT

Functional programming (FP) techniques produce code that is general, concise, composable, and correct. Until recently many of these techniques were limited to the realm of academia and esoteric programming languages. New C++ and boost developments enable us to embed FP in C++ in a seamless way.

We'll be covering recursive algebraic data types, curried functions, purity, generic programming, and category theory.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.2.13 [**Software Engineering**]: Reusable Software—*reusable libraries*

General Terms

Design, Languages

Keywords

C++, Boost, Haskell

1. INTRODUCTION

Although the benefits of functional programming have been widely recognized since the 80s[8] and continues to be a popular area of academic research[2, 3], popular industrial incantations have largely come in the form of new programming languages, such as F♯[1] and Scala[4].

While switching to a functional languages for a project has its benefit, use of functional languages often doesn't take advantage of preexisting developer expertise and the resulting code sometimes has difficult to solve performance issues[15, 7].

Our goal is to implement modern functional programming in C++ directly. By doing so, we hope to present a framework that has familiar usage to C++ programmers, but is also powerful enough to implement modern functional programming designs.

It is important to us that the design expands upon and integrates well with widely used C++ libraries and constructs. We feel our approach has uniquely met this goal.

While specific functional idioms have been attempted in C++[13, 11, 17], none that we are aware of has been able to handle all of recursive abstract data types, IO, polymorphic values, polymorphic classes (also known as type classes), and category theoretic constructs.

2. ALGEBRAIC DATA TYPES

The FP feature of Algebraic Data Types (ADTs) concerns itself with the creation of complex types from simpler ones using simple type composition operators. In this section, we'll overview the basic ADTs and later we'll show how to implement these in C++. Our syntax is mostly borrowed from Schmidt[16].

2.1 Unit Type

A unit type is a type that has exactly one possible value. There is a primitive unit type () that is called unit.

2.2 Product Operation

Given two types, A and B, we call the type $A \otimes B$ the product of A and B. A value of type $A \otimes B$ can be thought of as containing an element of A *and* an element B. Mathematically, this is often referred to as a pair where the value (a, b) is of type $A \otimes B$ if $a \in A$ and $b \in B$.

Pairing can be generalized to form n-tuples. For example, if $\Re$ is the type of the real numbers, $\Re \otimes \Re \otimes \Re$ is the type of three dimensional points.

2.3 Sum Operation

Given two types, A and B, we call the type $A \oplus B$ the sum of A and B. A value of type $A \oplus B$ can be thought of as containing an element of of type A *or* an element of type B.

If we have two distinct unit types T and F, $T \oplus F$ could be used for the type of booleans.

2.4 Type Definitions

Type definitions allow us to use shorthand names for more complex types. The general form is $a := e$ where a is the newly bound identifier and e is the type expression. We read $:=$ as "is the same as".

Expanding on our previous example, $B := T \otimes F$ allows us to use B wherever we would otherwise use $T \otimes F$.

Another form of type definition, $a ::= e$, allows us to create *new* types based on type expressions. We read $::=$ as "is implemented with".

For example, if we wanted to create a boolean type using *int* as the underlying representation (ignoring values other than 0 and 1), we would use $B ::= \mathit{int}$ instead of $B := \mathit{int}$. In this case we wouldn't want an arbitrary *int* to be replaceable with a B and vice-versa.

2.5 Type Functions

Type functions allow us to create reusable type templates. The general forms for type functions are $f\ a_1\ a_2 \ldots a_n := e$ and $f\ a_1\ a_2 \ldots a_n ::= e$ where f is the name of the newly bound type function and the type expression e may use $a_1 \ldots a_n$.

Consider an *OpInt* type that can be either an integer or nothing. A simple way to declare *OpInt* is as follows:

$$\mathit{none} ::= ()$$

$$\mathit{OpInt} := \mathit{none} \oplus \mathit{int}$$

If we think optional values are useful in a more general context, we can declare the type function *Op* that creates various optional types based on its type argument:

$$\mathit{none} ::= ()$$

$$\mathit{Op}\ a := \mathit{none} \oplus a$$

Creating a concrete type definition for *OpInt* then becomes:

$$\mathit{OpInt} := \mathit{Op}\ \mathit{int}$$

2.6 Recursive Types

Interesting types occur when type definitions are allowed to be recursive. That is, they may use the bound name on the right hand of the $=$ symbol. Here are a few examples:

The natural numbers N:

$$Z ::= ()$$

$$N := Z \oplus N$$

Lists L of arbitrary types a:

$$[] ::= ()$$

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

Binary trees, T, with values of type a at leaves:

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

3. ALGEBRAIC DATA TYPES IN C++

C++ gives us many options for implementing algebraic datatypes. We'll look at the most common forms and compare them.

3.1 Unit Type

boost.mpl includes a convenient unit type, `void_`, that we use for (). The following code implements `A` $:= ()$.

```
#include <boost/mpl/void.hpp>
typedef boost::mpl::void_ A;
```

For implementing new unit types, such as $B ::= ()$, we simply declare empty structs:

```
struct B {};
```

3.2 Product Operator

The most simple way to represent an n-tuple in C++ is with a C struct. The following struct represents `R3` $::=$ `R` $\otimes$ `R` $\otimes$ `R`.

```
struct R3
{
    R x;
    R y;
    R z;
};
```

Using a struct for products has many advantages. There is not a lot of syntax overhead and accessors for the underlying values are given convenient names. The disadvantages are that we must bind the struct to a name (`R3` in this case), and we cannot access the underlying values by the sequence in which they were declared, something useful in generic programming.

The Boost library offers several alternatives for product types and we'll consider a couple of our favorites here.

The boost.fusion vector template overcomes most of the disadvantages of using a struct, but sacrifices convenient accessor names.

```
#include <boost/fusion/include/vector.hpp>
#include <boost/fusion/include/at_c.hpp>

//...
boost::fusion::vector<A,B,C> p;
boost::fusion::at_c<0>( p ); //extract the value
                             //of the first type
boost::fusion::at_c<1>( p ); //extract the value
                             //of the second type
boost::fusion::at_c<2>( p ); //extract the value
                             //of the third type
```

boost.fusion's map provides an implementation of n-tuples with both named *and* integer accessors. The index names are implemented with unit types.

```
#include <boost/fusion/include/map.hpp>
#include <boost/fusion/include/at_key.hpp>
#include <boost/fusion/include/at_c.hpp>

using boost::fusion::pair;
using boost::fusion::at_key;
using boost::fusion::at_c;

struct x {};
struct y {};
struct z {};

boost::fusion::map
         < pair<x, R>
         , pair<y, R>
         , pair<z, R>
         > v;
at_key<x>( v ); //access x
at_key<y>( v ); //access y
at_key<z>( v ); //access z

at_c<1>( v ); //access y by integer
```

Although boost.fusion maps overcome all the disadvantages of a struct, they have more syntax overhead and require extra types for the accessor names.

3.3 Type level sum operators

Standard C++ has very limited support for implementation of sum. If all the underlying types are units and are not used elsewhere, the simplest method is to use enum.

For a simple `Direction` type,

$$\texttt{Left} ::= ()$$

$$\texttt{Right} ::= ()$$

$$\texttt{Direction} := \texttt{Left} \oplus \texttt{Right},$$

enum can be used as follows:

```
enum Direction { Left
               , Right
               };
Direction d;
d = Left;
d = Right;
```

As with struct-products, enum-sums must be bound to a name.

For any underlying types, we may use boost.variant. The following implements $A \oplus B$ for arbitrary distinct types A and B:

```
boost::variant<A, B> v;
v = a; //Where a is of type A.
v = b; //Where b is of type B.
```

Extraction of values from a variant involves use of the `apply_visitor` function. Below we define a function f from type $A \oplus B$ to `int`.

```
//This structure handles the different cases. We'll
//see later how this is a polymorphic function
//class.
struct F
{
    typedef int result_type;
    int operator()( const A & a ) const
    {
        //return result for a variant with value a
        //of type A...
    }
    int operator()( const B & b ) const
    {
        //return result for a variant with value b
        //of type B...
    }
};
int f( boost::variant<A,B> v )
{
    return boost::apply_visitor( F(), v );
}
```

boost.variant includes several variations of `apply_visitor` that are useful in common scenarios.

The final sum implementation we'll be looking at piggy-backs on Object Oriented programming. We create an abstract base class and a subclass for every contained type. `shared_ptr` from boost is then used to represent the final type. The following implements $C ::= A \oplus B$:

```
#include <boost/shared_ptr.hpp>

struct CBase {
    //virtual function required to make
    //CBase polymorphic
    virtual void dummy(){}
};
//C0 wraps values of type A.
struct C0 : public CBase
{
    C0( const A & val_ ) : val(val_) { }
    A val;
};
//C1 wraps values of type B.
struct C1 : public CBase
{
    C1( const B & val_ ) : val(val_) { }
    B val;
};

typedef boost::shared_ptr<CBase> C;

//Creation functions
C newC( const A & a )
    { return C(new C0( a ) ); }
C newC( const B & b )
```

```
    { return C(new C1( b ) ); }
```

To extract values, we use a boilerplate case function that is used in a similar way to `apply_visitor` from boost.variant.

```
template< typename F >
typename F::result_type
c_case( F f, C c )
{
    using boost::dynamic_pointer_cast;

    if( boost::shared_ptr<C0> c0
        = dynamic_pointer_cast<C0>( c ) )
        f( c0->val );
    else if( boost::shared_ptr<C1> c1
        = dynamic_pointer_cast<C1>( c ) )
        f( c1->val );
    else
    {
        // Should never get here
        assert( false );
    }
}
```

Compared to variant-sums, oo-sums require a lot of error-prone boilerplate per sum. oo-sums also are always created on the heap where variant-sums can be created on the stack[1]. On the other hand, oo-sums allow an underlying type to be repeated and have no limitations on the number of subtypes.

3.4 Type Definitions

Recall from earlier that we're using two ways to bind types to names, "is the same as" (:=) and "is implemented with" (::=).

As we have already seen, := is easily implemented with `typedef` as in the following declaration of $R3 := R \otimes R \otimes R$:

```
typedef boost::fusion::vector<R,R,R> R3;
```

To implement "is implemented with" binding, we use a wrapper struct. Lets say we want to implement a $[0, 1)$ bounded real number `R01` in terms of `double`. We wouldn't want to treat these as the same types[2] so our type is `R01` ::= `double`:

```
struct R01
{
    //stops implicit conversions
    explicit R01( const double impl_ )
        : impl( impl_ )
    {
    }
    double impl;
};
```

[1]This is true except for the case of recursive variant-sums which we'll see later.

[2]For example, adding $[0, 1)$ bounded numbers doesn't have a straightforward meaning as with `double`.

Separate construction and access functions are typically provided when using ::=.

```
R01 r01FromDouble( const double r )
{
    assert( r >= 0 && r < 1.0 );
    return R01( r );
}
double r01ToDouble( const R01 & a )
{
    return a.impl;
}
//Caller must ensure impl stays in the proper
//range.
double & r01_impl( R01 & a )
{
    return a.impl;
}
//...
```

3.5 Type Functions

The C++ type function trait from mpl suits our purposes for "is the same as" type functions (:=). The following implements our *Op* type function from earlier.

```
struct none {};

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

//An example use of the function
typedef Op<int>::type OpInt;
```

"Is implemented with" type functions are encoded as a template struct with a nested value instead of a nested type. An implementation of *pair* $a ::= a \otimes a$ follows:

```
template< typename a >
struct pair
{
    explicit pair( boost::fusion::vector<a,a>
                   impl_ )
                 : impl( impl_ )
    {
    }
    boost::fusion::vector<a,a> impl;
};
```

Separate construction and access functions would likely be created as with "is implemented with" type definitions.

3.6 Recursive Types

3.6.1 *Recursion over sums*

Consider a simple int list type,

`empty` ::= ()

$$\texttt{IntList} := \texttt{empty} \oplus (\texttt{int} \otimes \texttt{IntList})$$

The following well-intentioned variant implementation won't work:

```
using boost::variant;
using boost::fusion::vector;
struct empty{}
typedef variant< empty
               , vector< int
                       , IntList //C++ scope
                                 //rules don't
                                 //allow this!
                       >
               > IntList;
```

Fortunately, boost.variant includes a handy workaround called `make_recursive_variant`[3]:

```
#include <boost/variant/recursive_variant.hpp>
//...
struct empty{};
typedef boost::make_recursive_variant
    < empty
    , boost::fusion::vector2
        < int
        , boost::recursive_variant_
        >
        >::type IntList;
```

3.6.2 *Recursion over products*

Before we present our recursive product implementation, it would be beneficial to review laziness, also called delayed evaluation.

For a typical C++ function, all arguments must be evaluated before the function is called, even if the arguments aren't used. Consider the following function:

```
int iff( bool b, int a0, int a1 )
{
    if( b ) return a0;
    else    return a1;
}
```

This function requires the caller to compute both `a0` and `a1` even though both aren't required. If computing $a0$ and $a1$ is expensive, this could create performance issues.

Functional programming languages solve this by using the concept of lazy values. Lazy values are just like normal (strict) values, but are only computed when required. An easy way to implement laziness in C++ is to use a `boost::function` without arguments[4]. The type function `lazy` abstracts this implementation.

[3]In the implementation below, we use vector2, the nonvariadic form of fusion vectors, to work around an issue with make_recursive_variant as of boost 1.39.0

[4]For a performance penalty, a more sophisticated version of lazy could cache repeated computations as in McNamara Smaragdakis[10]

```
#include <boost/function.hpp>

template< typename a >
struct lazy
{
    //a function that simply returns a when called
    typedef boost::function< a ()> type;
};
```

Now we can implement an `iff` that only evaluates its second and third arguments when required:

```
int iff( bool b
       , lazy<int>::type a0
       , lazy<int>::type a1
       )
{
    if( b ) return a0(); //Calling a lazy value
                         //with no arguments
                         //forces evaluation.
    else    return a1();
}
```

We could also implement a version of `iff` that *returns* a lazy value:

```
lazy<int>::type iff( bool b
                   , lazy<int>::type a0
                   , lazy<int>::type a1
                   )
{
    if( b ) return a0;
    else    return a1;
}
```

Now that we've had a look at laziness, lets return to recursion over products. Consider the following type:

$$\texttt{IntStream} := \texttt{int} \otimes \texttt{IntStream}$$

At first glance, this type may seem nonsensical and to require an infinite amount of space to represent. However, if we allow the recursion of `IntStream` to be lazy we can indeed represent this infinite sequence.

Unfortunately, there isn't a `make_recursive_variant` specifically for fusion vectors. On the other hand, by wrapping our product in a recursive variant, we get the required recursion.

```
typedef boost::make_recursive_variant
    < boost::fusion::vector
        < int
        , lazy< boost::recursive_variant_ >::type
        >
    >::type IntStream;
```

We include an implementation of general streams,

$$\texttt{stream}\ a := a \otimes (\texttt{stream}\ a),$$

in the appendix.

Exercise 1. Implement the type N of natural numbers, $Z := (), N := Z \otimes N$.

Exercise 2.1. Write, in abstract form, the type function for general binary trees with one type for leaves and another for branches.

Exercise 2.2. Implement the type function from exercise 2.1 in C++.

4. FUNCTIONS

The meaning of a function in the context of functional programming is very simple. It is simply a mapping from one type to another. One mathematical representation of a function is a set of pairs (a, b) where $a \in A$, $b \in B$, where for every value of type A there is exactly one pair in the set. We write $A \to B$ to refer to the type of functions from A to B. When we use the word function, we're referring to a mathematical function. We use the term C++ function to refer to C++ functions.

So, just what is a C++ function and how does it relate to functions? Consider the following C++ function declaration:

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

Here we have something that resembles a function with type $\texttt{int} \to \texttt{bool}$. This general mapping won't always work though. For instance, C++ functions might take more than one argument.

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

Using our algebraic data types, we can say this resembles a function with type $(\texttt{int}\otimes\texttt{int}) \to \texttt{bool}$. A C++ function call can be thought of as constructing a special C++-function-tuple and using that as the argument.

However we still don't have a general meaning for C++ functions. Consider this C++ function:

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

Our straightforward C++-function-tuple mapping won't work in this case because the set representation cannot encode the change to the global variable. To handle this situation, FP researchers add another parameter to the input tuple of type World and a similar parameter to the output type. World represents the general state of the world and would include, in this case, some_global_var's value. f3 has type

$$(\texttt{int} \otimes \texttt{int} \otimes \texttt{World}) \to (\texttt{bool} \otimes \texttt{World}).$$

We now have a formula for getting the type of an arbitrary C++ function. The type of:

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

is

$$(A_1 \otimes A_2 \cdots A_n \otimes \texttt{World}) \to (\texttt{World} \otimes R).$$

Without loss of generality, if a C++ function has a return type of void we consider this to be our unit type ().

4.1 Currying

Currying is the process by which we transform a function with a tuple parameter into a function that returns another function. Consider the following two C++ functions:

```
double add(double a, double b)
{
    return a+b;
}

//curried version of add
boost::function<double (double)>
addcurry( double a )
{
    return boost::bind( add, a, _1 );
}
```

Ignoring World for now, the type of add is

$$(\texttt{double} \otimes \texttt{double}) \to \texttt{double}$$

and the type of addcurry is

$$\texttt{double} \to (\texttt{double} \to \texttt{double}).$$

This currying process can be generalized for n arguments (we will hereafter omit the parenthesis and understand $\to$ to be right associative) . The resulting function is called "curried".

Practically, applying curried functions, sometimes called partial application, often results in other useful functions. For example, the function resulting from addcurry(1) always adds 1 to its argument.

Curried functions can often be used instead of boost.bind and boost.lambda. Their lightweight syntax and simple meaning often makes them preferable.

4.2 IO

Returning to our general type of a C++ function

$$(A_1 \otimes A_2 \cdots A_n \otimes \texttt{World}) \to (\texttt{World} \otimes R)$$

we curry all the arguments to produce

$$A_1 \rightarrow A_2 \cdots A_n \rightarrow \texttt{World} \rightarrow (\texttt{World} \otimes R).$$

By convention, functional programmers often use a type function called IO,

$$IO\ a := \texttt{World} \rightarrow (\texttt{World} \otimes a).$$

It is easy to see that $IO\ a$ represents C++ functions without arguments with return values of type a. We now write our general type for C++ functions as

$$A_1 \rightarrow A_2 \cdots A_n \rightarrow IO\ R$$

With the goal in mind of converting C++ functions into something that works more like functions, we use the following type function to represent IO:

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

and the following type function to represent functions:

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

The gfp[5] library includes a type function called `cfunc` that returns a curried function type of its arguments. For example:

```
gfp::cfunc<int,int,bool>::type
```

is equivalent to:

```
func< int, func<int, bool >::type >::type
```

Given all this, we can now implement a `ciof` function that converts a C++ function pointer into a curried io function.

```
bool f( int, int );
//...
using namespace gfp;

cfunc< int
     , int
     , io<bool>::type >::type fc = ciof( f );
```

[5]Latest version available from http://netsuperbrain.com/gfp

```
//fc1 is the partial application of fc with 12.
cfunc< int
     , io<bool>::type > fc1 = fc( 12 );

io<bool>::type fc1 = fc1( 0 );

bool b = fc1(); //f actually called here
                //with 12 and 0 as the
                //arguments.
```

4.3 Purity

As we've seen above, not all C++ functions require a `World` argument. These functions are called pure. When converting pure C++ functions to functions it is often nice to omit the IO part. gfp includes a `gfp::cf` function that converts a C++ function pointer into a pure function. The caller of `gfp::cf` must prove that its argument is indeed pure.

```
bool greaterThan_( int a, int b )
{
    //Pure, because no global state
    //is changed.
    return a > b;
}

gfp::cfunc<int, int, bool>
greaterThan = gfp::cf( greaterThan_ );

bool b = greaterThan( 12 )( 6 );
```

5. GENERIC PROGRAMMING

The Generic Programming features of FP allow one to write programming constructs that can be applied to unforeseen types. In C++ we implement generics with polymorphic values, polymorphic functions, and polymorphic classes.

5.1 Polymorphic Values

A polymorphic value is like a normal value in C++ except it has multiple supported types. For example, the polymorphic value `empty` supports `std::vector<A>` types for any `A`, but would not support the type `int`.

To represent polymorphic values in C++, we introduce the polymorphic value trait. $v \in V$ is a polymorphic value if:

1. if A is a supported type, $v((A*)(0))$ returns type A, and `boost::result_of<V(U)>::type` is type A.
2. if U is not a supported type, $v((U*)(0))$ doesn't compile.

What follows is an implementation of the `empty` polymorphic value for arbitrary standard vectors.

```
//Empty is the polymorphic value type
struct Empty
{
    //For boost::result_of support
    template< typename T >
    struct result;
    template< typename A >
    struct result< Empty( std::vector< A > ) >
    {
        typedef std::vector< A > type;
    };

    //our operator()
    template< typename T >
    typename result< Empty( T ) >::type
    operator()( T const * const dummy ) const
    {
        typedef typename result< Empty( T ) >::type V;
        return V(); //empty vector
    }
};
//empty is the polymorphic value
const Empty empty = Empty();
```

gfp introduces a simple template function that resolves a polymorphic value into a value of one of its supported types:

```
namespace gfp
{
    //Takes a polymorphic value pv and a type T as
    //parameters and returns the result of
    //resolving pv into a value of type T.
    template< typename T, typename PV >
    T resolve( const PV & pv )
    {
        return pv( (T const * const)0 );
    }
}
```

Use of polymorphic values is then straightforward:

```
typedef std::vector<std::string> T1;
typedef std::vector<int> T2;
T1 t1 = gfp::resolve<T1>( empty ); // ok
T2 t2 = gfp::resolve<T2>( empty ); // ok
int i = gfp::resolve<int>( empty ); // error, int
                                    // not
                                    // supported
```

5.2 Polymorphic Functions

One simple way to implement polymorphic functions is to just use polymorphic values with functions as the supported types. To call a polymorphic function of this variety, the library user must explicitly resolve the type. This two-stage approach of resolving a function's type and then calling it can become quickly tedious.

To reduce the aforementioned difficulty, we took an approach similar to McNamara Smaragdakis[?]. Our approach allows one to apply a polymorphic function without having to explicitly resolve its type beforehand. This simplifies the use of polymorphic functions tremendously.

$f \in F$ is a polymorphic function if:

1. If A is a supported argument type and $a \in A$, $f(a)$ returns type `boost::result_of<F(A)>::type`.
2. If U is not a supported argument type and $u \in U$, $f(u)$ results in a compile error.
3. The return type of f can be any of a polymorphic value type, a polymorphic function type, or a normal type.

What follows is a polymorphic function that computes the length of any `std::vector` or `std::set` types.

```
//Length is the polymorphic function type
struct Length
{
    //For boost::result_of support
    typedef int result_type;

    template< typename T >
    result_type operator()
        ( const std::vector<T> & v ) const
    {
        return v.size();
    }
    template< typename T >
    result_type operator()
        ( const std::set<T> & s ) const
    {
        return s.size();
    }
};
//length is the polymorphic function
Length length = Length();
```

Polymorphic functions are used just the same as normal functions.

```
std::vector<int> v;
//...
int i = length( v ); //here length is auto-
                     //matically resolved
                     //to the std::vector
                     //argument type when
                     //called.

typedef gfp::cfunc< std::set<char>
                  , int >::type F;
F f = F( length ); //here length is
                   //explicitly resolved
                   //to type F.
```

5.3 Polymorphic Classes

Extending a polymorphic entity (function or value) to handle a new type requires direct modification of that entity's code. Oftentimes it is preferable to allow someone to extend several related polymorphic entities in separate programming modules. Polymorphic classes allow this.

A polymorphic class C consists of

1. A single argument template struct declaration called C.

2. A boost concept checking class named IsC that serves to verify if a type T is supported by C and verify that T's corresponding extension class has the correct polymorphic member types.

3. For each of the related polymorphic entities of C, a corresponding implementation that uses extension classes to select a specific implementation for a supported type.

4. For all supported types, a partial or complete template specialization the C struct with valid implementations for each of the related polymorphic entities. These are called extension classes.

To illuminate the idea, we'll look a at a polymorphic class called Showable. It has one entity called show.

```
#include <boost/concept_check.hpp>

//Single argument declaration
template< typename T >
struct Showable;

//Concept checking class
template< typename T >
struct IsShowable
{
    //Nested type show must be a polymorphic
    //function type from T to string.
    typedef typename Showable<T>::show t;
};

//Show is the polymorphic function type
struct Show
{
    typedef string result_type;

    template< typename T >
    string operator()( const T & t ) const
    {
        //The polymorphic functions of
        //a class use BOOST_CONCEPT_ASSERT
        //ensuring a supported type.
        BOOST_CONCEPT_ASSERT((IsShowable<T>));
        //Here the specific implementation is
        //selected and used.
        return typename Showable<T>::show()( t );
    }
};
//show is the polymorphic function.
const Show show = Show();
```

At this point the show polymorphic function of the Showable polymorphic class has no supported types. Adding a supported type involves specializing the Showable template. Below we make such an extension for the int type:

```
//A show polymorphic function type that works
//only on int.
struct ShowInt
{
    typedef std::string result_type;
    std::string operator()( const int i ) const
    {
        return boost::lexical_cast<std::string>(i);
    }
};
template<>
struct Showable<int>
{
    //declares ShowInt as the polymorphic
    //function type for ints.
    typedef ShowInt show;
};
```

We can even extend Showable to support arbitrary variants of other supported types using partial template specialization:

```
//The polymorphic function type for variants
//of types supported by Showable.
struct ShowVariant
{
    typedef std::string result_type;

    template< BOOST_PP_ENUM_PARAMS
                ( BOOST_VARIANT_LIMIT_TYPES
                , typename T
                )
            >
    std::string operator()
        ( const boost::variant
            < BOOST_PP_ENUM_PARAMS
                ( BOOST_VARIANT_LIMIT_TYPES
                , T
                )
            > & v
        ) const
    {
        //this will only compile when all
        //of v's subtypes are supported by
        //Showable.
        return boost::apply_visitor( show, v );
    }
};
template< BOOST_PP_ENUM_PARAMS
            ( BOOST_VARIANT_LIMIT_TYPES
            , typename T
            )
        >
struct Showable
        < boost::variant
            < BOOST_PP_ENUM_PARAMS
                ( BOOST_VARIANT_LIMIT_TYPES
                , T
                )
            >
        >
{
    typedef ShowVariant show;
};
```

6. CATEGORY THEORY

Category theory is a field of pure mathematics that deals with relationships between abstract sets. Functional Programming uses several category theoretical concepts to come up with interesting and highly reusable polymorphic entities.

A complete exposition of applied category theory is outside the scope of this paper. We hope to whet the reader's appetite with a few examples. See [19] for an extensive treatment of the different polymorphic classes and [14, 9, 12, 18, 6] for some interesting applications.

6.1 Monoids

One of the simplest concepts of Category Theory is the monoid. A type (M), an element of that type (0), and a binary operator over that type (+ with type $M \to M \to M$) form a monoid if the operator is associative and 0 is an identity.

For example: the `std::string` type, the empty string, and `std::string`'s concatenation operator (`operator+`) form a monoid. gfp includes a polymorphic class Monoid with corresponding polymorphic entities `mzero` and `mplus`.

Things start to get interesting when we try to form monoids out of more complex types. For instance, a function ($A \to B$) can form a monoid if we require B to be a monoid. We define 0 to a function that always returns the 0 of B. We define + to return a function that applies its first two parameters to the third and uses B's + on the result. Consider the following example:

```
string header_( const string & s )
{
    return show( s.size() ) + ";";
}
gfp::cfunc<string,string>::type
header = gfp::cf( header_ );

string contents_( const string & s )
{
    return s + ";";
}
gfp::cfunc<string,string>::type
contents = gfp::cf( contents_ );

gfp::cfunc<string,string>::type
message = gfp::mplus( header )( contents );
```

The `message` function would return "5;hello;" with an input of "hello".

Let's look at another monoid example. Our unit type is part of a trivial monoid and *IO A* forms a monoid in a similar way to functions. This combination allows us to easily sequence procedures functionally:

```
//gfp includes the following typedef:
//typedef gfp::io
//        < boost::mpl::void_ >::type Action;

void writeLn_( const string & s )
{
    std::cout << s << std::endl;
}
gfp::cfunc<string,Action>::type
writeLn = gfp::ciof( writeLn_ );

gfp::cfunc<Action>::type
writeWorm = writeLn("_/\__/\__0>");

gfp::cfunc<Action>::type
notifyLoading = gfp::mplus( writeLn("Loading...")
                          , writeWorm
                          );
```

6.2 Polymorphic Classes for Standard Types

One positive aspect of category inspired polymorphic classes is their broad applicability. Algorithms written for a functor, for instance, will work for any number of built-in types as well as those that have yet to be designed. To give an idea of what is possible, we list below where common types in boost and standard C++ fall into category theoretic polymorphic classes[6]:

1. Monoid/Functor/Pointed/Idiom/Monad/Foldable/Traversable
   (a) std::basic_string
   (b) std::deque
   (c) std::list
   (d) std::stack
   (e) std::queue
   (f) boost::optional
2. Monoid/Functor/Pointed/Idiom/Monad
   (a) std::pair
   (b) boost::compressed_pair
   (c) boost::function1
   (d) boost::fusion::vector
   (e) boost::tuple
3. Monoid/Functor/Pointed/Idiom/Foldable/Traversable
   (a) boost::array
4. Monoid/Foldable
   (a) std::string
   (b) std::bitset
   (c) std::priority_queue
   (d) std::set
   (e) std::multiset
   (f) boost::bimap
5. Monoid/Functor/Foldable/Traversable
   (a) std::map
   (b) std::multimap

[6]Our selection comes from [19] but we use the term idiom instead of applicative functor

7. CONCLUSIONS

In this paper, we presented a solution to the problem of embedding modern functional programming concepts in the C++ language. We demonstrated recursive algebraic data types implemented with standard C++ constructs, boost.fusion, and boost.variant. We showed how curried functions, with and without io, build upon and integrate with boost.function. Our design of polymorphic functions is similar to that of McNamara and Smaragdakis[11], but integrates with boost's result-of operator. A novel design for polymorphic values and polymorphic classes was presented which allowed for category theoretic concepts to be integrated.

The concepts here have already been used successfully in two large scale CAD/CAM frameworks to implement embedded domain specific languages (EDSLs) for motion control and functional reactive programming[5] for operator user interfaces.

8. ACKNOWLEDGMENTS

We thank Robocasting Enterprises, Anygma, Sandia National Labs, and all our other customers whose programming problems inspired the present work. We also thank Conal Elliott for his guidance on modern functional programming concepts.

A special thanks to Reverend Adam Agnew and Trey Marcus for their helpful suggestions.

9. REFERENCES

[1] F♯ at microsoft research. http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/.

[2] International conference on functional programming. http://www.icfpconference.org/.

[3] Journal of functional programming. http://journals.cambridge.org.

[4] Scala. http://www.scala-lang.org/.

[5] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273, New York, NY, USA, 1997. ACM.

[6] C. M. Elliott. Push-pull functional reactive programming. In *Haskell '09: Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, pages 25–36, New York, NY, USA, 2009. ACM.

[7] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Commercial uses: Going functional on exotic trades. *J. Funct. Program.*, 19(1):27–45, 2009.

[8] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[9] C. Mcbride and R. Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[10] B. McNamara and Y. Smaragdakis. Syntax sugar for FC++: lambda, infix, monads, and more. In *Declarative Programming in the Context of OO Languages (DPCOOL) at PLI'03*, 2003.

[11] B. McNamara and Y. Smaragdakis. Functional programming with the fc++ library. *J. Funct. Program.*, 14(4):429–472, 2004.

[12] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 124–144, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[13] G. Munkby, A. Priesnitz, S. Schupp, and M. Zalewski. Scrap++: scrap your boilerplate in c++. In *WGP '06: Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming*, pages 66–75, New York, NY, USA, 2006. ACM.

[14] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84, New York, NY, USA, 1993. ACM.

[15] C. J. Sampson. Experience report: Haskell in the 'real world': writing a commercial application in a lazy functional lanuage. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 185–190, New York, NY, USA, 2009. ACM.

[16] D. A. Schmidt. *Denotational semantics: a methodology for language development.* William C. Brown Publishers, Dubuque, IA, USA, 1986.

[17] J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *In First Workshop on C++ Template Programming*, page 00, 2000.

[18] S. D. Swierstra. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, pages 252–300. Springer-Verlag, Berlin, Heidelberg, 2009.

[19] B. Yorgey. The typeclassopedia. *Monad reader*, 13.

APPENDIX

A. STREAMS

```
template< typename T >
struct lazy
{
    typedef boost::function0< T > type;
};

template< typename T >
struct stream
{
    typedef typename boost::make_recursive_variant
        < boost::fusion::vector2
            < T
            , typename lazy
                    < boost::recursive_variant_
                    >::type
            >
            >::type type;
};

namespace detail
{
    template< typename Stream >
    struct stream_inner_type
    {
        typedef typename boost::mpl::at_c
                    < typename Stream::types
                    , 0
                    >::type type;
    };
}

namespace result_of
{
    template< typename Stream >
    struct stream_head
    {
        typedef typename detail
                       ::stream_inner_type<Stream>
                       ::type inner_type;
        typedef typename
            boost::fusion::result_of::at_c
                 < const inner_type
                 , 0
                 >::type type;
    };
};

template< typename Stream >
typename result_of::stream_head<Stream>::type
stream_head( const Stream & s )
{
    typedef typename detail
                   ::stream_inner_type<Stream>
                   ::type inner_type;
    return boost::fusion::at_c<0>
            ( boost::get<inner_type>( s ) );
}
namespace result_of
{
    template< typename Stream >
    struct stream_tail
    {
        typedef typename detail
                       ::stream_inner_type<Stream>
                       ::type inner_type;
        typedef typename
            boost::fusion::result_of::at_c
                 < const inner_type
                 , 1
                 >::type type;
    };
};

template< typename Stream >
typename result_of::stream_tail<Stream>::type
stream_tail( const Stream & s )
{
    typedef typename detail
                   ::stream_inner_type<Stream>
                   ::type inner_type;
    return boost::fusion::at_c<1>
        ( boost::get<inner_type>( s ) );
}

template< typename T, typename F >
typename stream<T>::type
stream_create( const T & t, F f )
{
    typedef typename stream<T>::type Stream;
    typedef typename detail
                   ::stream_inner_type<Stream>
                   ::type inner_type;
    return inner_type( t, f );
}
```