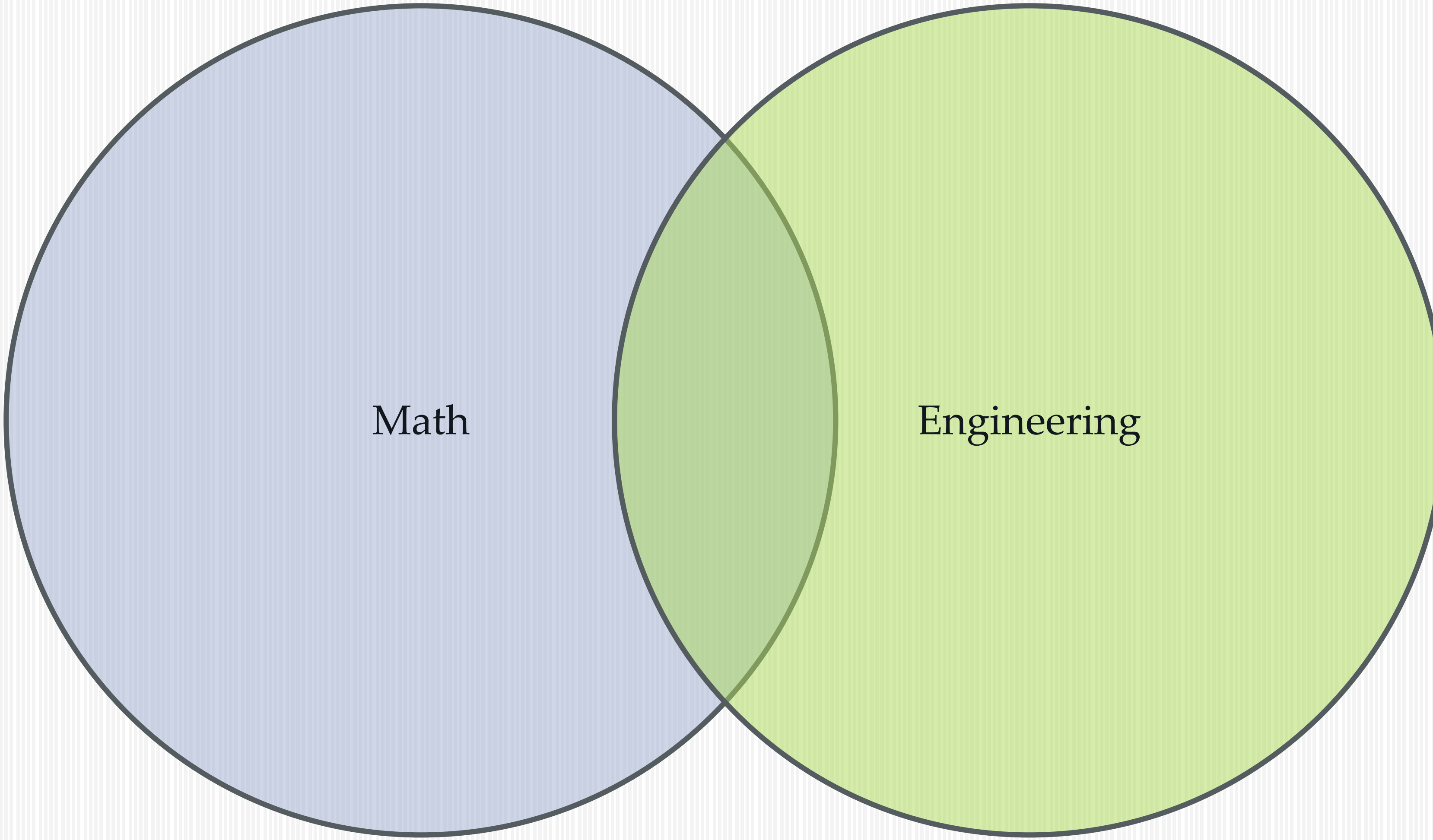
Functional Design Explained

David Sankel - Stellar Science
david@stellarscience.com
CppCon 2015


Math
Engineering

Quiz

Are the following two C++ programs the same?

Quiz

Are the following two C++ programs the same?

```
#include <iostream>

int main( int argc, char** argv )
{
  std::cout << "Hello World\n";
}
```

```
#include <iostream>

int main( int argc, char** argv )
{
  std::        cout << "Hello World\n";
}
```

Quiz

Are the following two programs the same?

```
#include <iostream>

int main( int argc, char** argv )
{
  std::cout << "Hello World\n";
}
```

```
#!/usr/bin/env python

print("Hello World")
```

Quiz

Are the following two programs the same?

```
int f( int c )
{
  if( false )
    return 45;
  else
    return c + 5;
}
```

```
int f( int c )
{
  int j = 5;
  j += c;
  return j;
}
```

Essence of Programs

Ideally we would like:

- Strong equivalence properties
- Something written down
- A set of rules we can apply to any program

Any ideas of what would be a good essence language?

How about math?

$$3 + 2 = 5$$

$$5 = 3 + 2$$

Denotational Semantics

Developed by Dana Scott and Christopher Strachey in late 1960s

Write a mathematical function to convert syntax to meaning (in math).

$\mu[\![e_1 + e_2]\!] = \mu[\![e_1]\!] + \mu[\![e_2]\!]$ where e_i is an expression
$\mu[\![i]\!] = i$ where i is an integer

What is the meaning of this?

```
int f( int c )
{
  if( false )
    return 45;
  else
    return c + 5;
}
```

Function Meaning

We could represent a function as a set of pairs
As in:

{ ….,(-1,4), (0,5), (1,6),…}

Or as a lambda equation: $\lambda c.\ c + 5$

Or something else: $f(c) = c + 5$

```
int f( int c )
{
  if( false )
    return 45;
  else
    return c + 5;
}
```

Function Meaning

What about this?

```
int f( int c )
{
  for(;;) ;
  return 45;
}
```

…,(-1, ⊥),(0,⊥),(1, ⊥),…

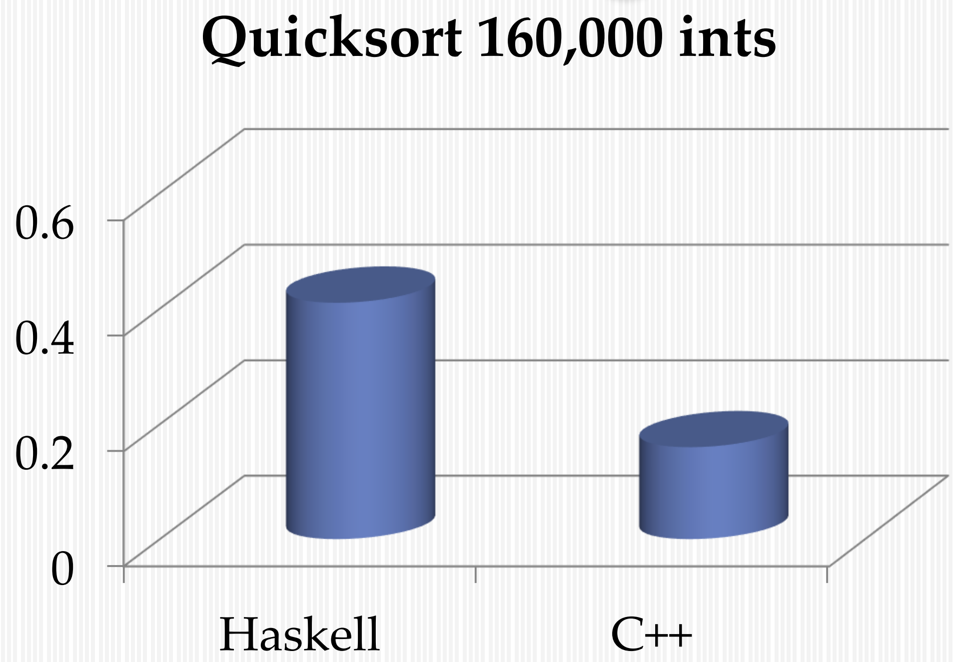
⊥ is “bottom”

The Next 700 Programming Languages

P. J. Landin wrote in 1966 about his programming language ISWIM (If you See What I Mean)

```
f(b+2c) + f(2b-c)
where f(x) = x(x+a)
```

Why not drop the C++ nonsense and go Haskell?



Haskell variant with optimizations: 0.41s

C++ variant without optimizations: 0.16s

Languages and Machines

Semantics Discovery

- Discover the mathematical essence of a problem and derive an implementation.
- Conal Elliott, various applications of semantics discovery throughout career.
- See 'Denotational design with type class morphisms'.

Ingredients

- Math augmented with some extra constructs can represent the essence of code.
- Write programs in math.
- C++ straddles more levels of abstraction than any other language.
- Discover essence of problem using math and derive implementation.

Functional Design

1. Discover the mathematical essence of the problem and write it out.
2. Derive an efficient implementation in C++ that has the interface discovered in #1.

Algebraic Data Types

- Mathematical fundamentals of base types.
- Two types, 1 and 0
- Two ops, $\oplus$ and $\otimes$, to compose them

0 Type

0 is the type with no values

```
struct Zero {
  Zero() = delete;
};
```

1 Type

1 is the type with one value

```
struct One {};
```

Product

Given types ‘a’ and ‘b’, the product of ‘a’ and ‘b’ (a ⊗ b) is a type whose values have an ‘a’ and a ‘b’.

```
using AAndB = std::pair<A,B>;

using AAndB = std::tuple<A,B>;

struct AAndB {
  A a;
  B b;
};
```

Product

Is this an implementation of a $\otimes$ b?

```
struct AAndB {
  std::unique_ptr<A> a;
  std::unique_ptr<B> b;
};
```

Sum

A ⊕ B is a type whose values are either a value of type 'A' or a value of type 'B'.

```
struct AOrB {
  bool hasA;
  union {
    A a;
    B b;
  } contents; // 'a' when 'hasA==true'
              // otherwise 'b'.
};

using AOrB = boost::variant< A, B >;

using AOrB = std::variant< A, B >; // hopefully
```

Function Type

$A \rightarrow B$ is a type whose values are pure functions with an input type A and an return type B.

```
using FunctionAB = std::function<B (A)>;
```

Meaning

$\mu[\![\text{ syntax }]\!] = \text{mathExpression}$

The **meaning** of "syntax" is the math expression

$\mu[\![\text{ expression }]\!] : \text{mathExpression}$

The **type** of "expression" is the math expression

$\mu[\![\text{ int }]\!] = \mathbb{Z}$

$\mu[\![3]\!] : \mathbb{Z}$

$\mu[\![3]\!] = 3$

Some Examples

$\mu[\![$ boost::optional<e_1> $]\!] = \mu[\![e_1]\!] \oplus 1$

$\mu[\![$ std::pair<e_1,e_2> $]\!] = \mu[\![e_1]\!] \otimes \mu[\![e_2]\!]$

$\mu[\![\text{double}]\!] = \mathbb{R}$

or maybe

$\mu[\![\text{double}]\!] = \mathbb{R} \oplus 1 \oplus 1 \oplus 1$

where the extra states are $-\infty$, $+\infty$, and NaN

What is a movie?

What is a movie?

$\mu[\![\ \text{Movie}<e>\ ]\!] = \mathbb{R} \to \mu[\![\ e\ ]\!]$

Operations:

$\mu[\![\ \text{always}<e>\ ]\!] : \mu[\![\ e\ ]\!] \to \mu[\![\ \text{Movie}<e>\ ]\!]$

$\mu[\![\ \text{always}<e>(a)\ ]\!] = \lambda\ t.\ \mu[\![\ a\ ]\!]$

$\mu[\![\ \text{snapshot}<e>\ ]\!] : \mu[\![\text{Movie}<e>]\!] \to \mathbb{R} \to A$

$\mu[\![\ \text{snapshot}<e>(\text{movie, time})\ ]\!] = \mu[\![\ \text{movie}\ ]\!]\ (\ \mu[\![\ \text{time}\ ]\!]\ )$

$\mu[\![\ \text{transform}<A,B>\ ]\!] : (\mu[\![A]\!] \to \mu[\![B]\!]) \to \mu[\![\text{Movie}<A>]\!] \to \mu[\![\text{Movie}<B>]\!]$

$\mu[\![\ \text{timeMovie}\ ]\!] : \mu[\![\ \text{Movie}<\text{double}>\ ]\!]$

$\mu[\![\ \text{timeMovie}\ ]\!] = \lambda\ t.\ t$

Grey flux movie

```
auto greyFluxMovie = transform(
  []( double timeInSeconds ) -> Image {
    double dummy;
    double greyness = std::modf( timeInSeconds, &dummy );
    return greyImage(  greyness );
  , time );
```

What is a stream?

What is a stream?

Let 'Action' be some side-effecting operation.

μ⟦ sink<e> ⟧ = μ⟦ e ⟧ → Action
μ⟦ source<e> ⟧ = (μ⟦ e ⟧ → Action) → Action

```
template< typename T >
using sink = std::function<void ( const T & )>;
```

```
template< typename T >
using source = std::function<void ( sink<T> ) >;
```

Example source/sink

```
source<char> consoleInput = []( sink<char> s ) {
  int inputChar;
  while( (inputChar = std::cin.get()) != EOF ) {
    s( static_cast< char >( inputChar ) );
};

sink<char> consoleOutput = []( char c ) {
  std::cout.put( c );
};
```

Connecting Sources and Sinks

$\mu[\![\ \text{connect}<e>\ ]\!] : \mu[\![\ \text{source}<e>\ ]\!] \rightarrow \mu[\![\ \text{sink}<e>\ ]\!] \rightarrow \text{Action}$

$\mu[\![\ \text{connect}<e>(\ so,\ si\ )\ ]\!] = \mu[\![\ so\ ]\!](\ \mu[\![\ si\ ]\!]\ )$

```
template< typename t >
void connect( source<t> so, sink<t> si ) {
  so( si );
}

int main( int argc, char** argv ) {
  connect( consoleInput, consoleOutput );
}
```

Transforming Streams

$$\mu[\![\ \text{sink<e>}\ ]\!] = \mu[\![\ \text{e}\ ]\!] \rightarrow \text{Action}$$

$$
\begin{aligned}
\mu[\![\ \text{transform<a,b>}\ ]\!] &= \mu[\![\ \text{Sink<b>}\ ]\!] \rightarrow \mu[\![\ \text{Sink<a>}\ ]\!] \\
&= \mu[\![\ \text{Sink<b>}\ ]\!] \rightarrow (\mu[\![\ \text{a}\ ]\!] \rightarrow \text{Action}) \\
&= \mu[\![\ \text{Sink<b>}\ ]\!] \rightarrow \mu[\![\ \text{a}\ ]\!] \rightarrow \text{Action}
\end{aligned}
$$

```
template< typename a, typename b >
using transform = std::function<void ( sink<b>, a ) >;
```

Application of Transforms

```
μ⟦ transform<a,b> ⟧ = μ⟦ sink<b> ⟧ → μ⟦ a ⟧ → Action

μ⟦ applyToSink<a,b> ⟧
  : μ⟦ transform<a,b> ⟧ → μ⟦ sink<b> ⟧ → μ⟦ sink<a> ⟧

μ⟦ applyToSource<a,b> ⟧
  : μ⟦ transform<a,b> ⟧ → μ⟦ source<a> ⟧ → μ⟦ source<b> ⟧

μ⟦ so >> t ⟧ = μ⟦ applyToSource<a,b> ⟧( t, so );
μ⟦ t >> si ⟧ = μ⟦ applyToSink<a,b> ⟧( t, si );
μ⟦ so >> si ⟧ = μ⟦ connect<t> ⟧( so, si );
```

Tranformers Continued…

```
transformer<char, std::string> getLines = //…
transformer<std::string, char> unWords = //…

source<string> inputLines = consoleInput >> getLines;
sink<string> wordOutput = unwords >> consoleOutput;
InputLines >> wordOutput;

transformer<char,char> linesToSpaces = getLines >> unwords;
```

What is command line processing?

What is command line processing?

μ⟦ CommandLineProcessor<a> ⟧ = ListOf String → μ⟦ a ⟧ ?

Hrm…

μ⟦ Parser<a,b> ⟧ = ListOf μ⟦ a ⟧ → μ⟦ b ⟧

μ⟦ CommandLineProcessor<a> ⟧ = μ⟦ Parser<String,b> ⟧

Command Line Parsing

```
struct HelpFlag{};
struct UserFlag{
  std::string user;
};

auto flagP = mix(
  args("--help", HelpFlag()),
  args("--user"  >> stringP,
        []( std::string username ) { return UserFlag{username}; })
```

Command Line Parsing

```
struct ListAccounts {};
struct ListJob {
  int jobId;
};

struct CommandLineParse {
  std::vector< boost::variant< HelpFlag, UserFlag > > globalFlags;
  boost::variant< ListAccounts, ListJob > mode;
};

auto parser = flagP
  >> (args( "listAccounts", ListAccounts() )  | |
        (args( "listJob" ) >> "--jobId" >> intP( []( int id ) { return ListJob{id};}));
```

Benefits

- Highly Flexible
- Highly Composible
- Type safe
- Simple

Functional Design

1. Discover the essence
2. Derive the implementation

- Beautiful API's
- Screaming Speed

David Sankel david@stellarscience.com