The Intellectual Ascent to Agda

David Sankel
Stellar Science

•

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";
}
```

```
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 intermediate 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:

{ ...,(-1,44), (0,45), (1,46),...}

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

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

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

Function Meaning

What about this?

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

…,(-1, ⊥),(0,⊥),(1, ⊥),…

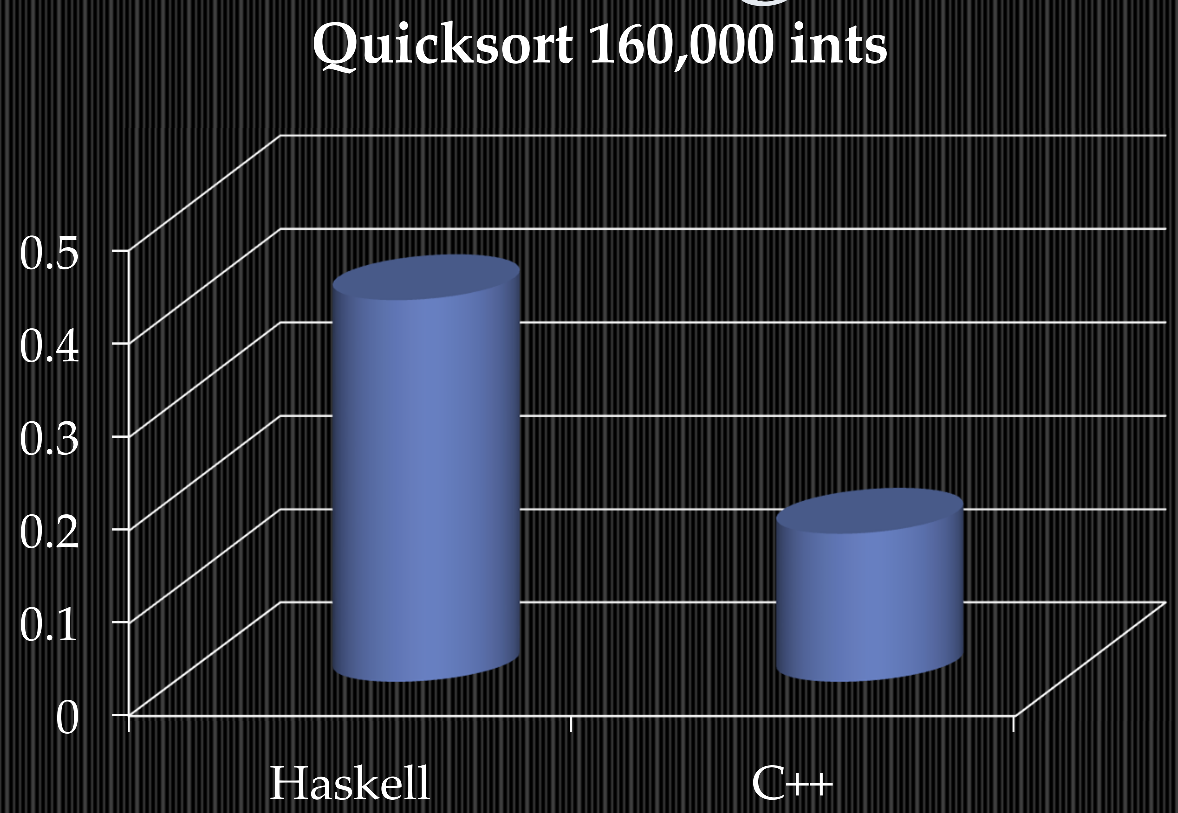
⊥ is “bottom”

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)
```

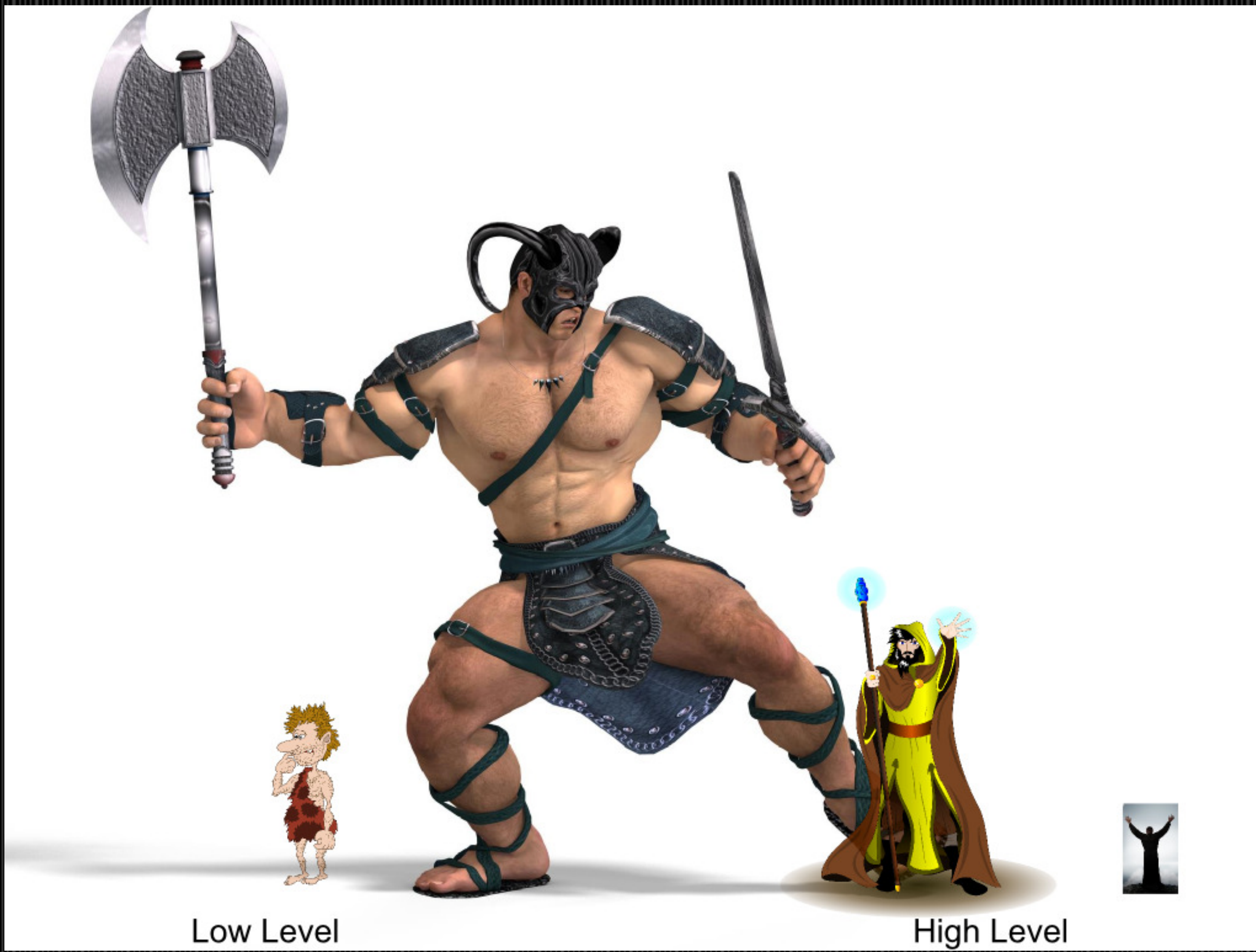
Why not drop the C++ nonsense and go Haskell?



Haskell variant with optimizations: 0.41s

C++ variant without optimizations: 0.16s

Languages and Machines



Denotational Design

- Conal Elliott, various applications of denotational design throughout career.
- See ‘Denotational design with type class morphisms’.
- Main idea:
  - Design semantics in Haskell without regard to performance.
  - Derive a speedy implementation in Haskell using the semantics.


Agda

Types in Agda

a : int

a = 5

Agda as semantic domain for C++ expressions:

$\mu[\![3 + 2]\!] = 5$

$\mu[\![3 + 2]\!] : int$

$\mu[\![e_1 + e_2]\!] : int$ where e_i is an int expression

Functions in Agda

"a → b" is the type of functions with input 'a' and output 'b'.

Consider:

```
int f( char c ) { return 4; }
```

In Agda we would write something like ,

```
f : char → int
f c = 4
```

Calling functions is done without parentheses. So, *f('a')* would become *f 'a'* instead.

Multiple parameter function types are written like:

```
char → int → char
```

Pairs in Agda

a × **b** is a pair type where the first element has type **a** and the second element has type **b**. It is also called a product type.

A value of type **a** × **b** is written as (x,y) where x has type **a** and y has type **b**.

So with denotational semantics, we can say

$\mu[\![\text{std::pair<a,b>}]\!]_T = \mu[\![a]\!] \times \mu[\![b]\!]$

Note: We're using T and E subscripts to differentiate between type and expression contexts where there is ambiguity.

Magic: Types of Types

Types have a "type" too (a universe actually).

Int : Set

Here's a function type. It's "type" is also Set.

Int → Char : Set

Set has a type too: Set : Set1

Any ideas why Set's type isn't Set?

Magic: Type-Value Mixing

Types and values can be mixed.

```
intToType : Int -> Set
intToType 0 = Char
...          _ = Int
```

How would you write this function in C++?

Magic: Type-Value Mixing

```
intToType : Int -> Set
intToType 0 = Char
...            _ = Int

template< int i > struct intToType { typedef int type; }
template<> struct intToType<0>{ typedef char type; }

// Call it like this (z = intToType 0)
typedef intToType<0>::type z
```

Magic: Dependent Types

We can figure out the saturated vector type's meaning's type. It is just another type. (We're still in the type context)

$\mu[\![$ std::vector<char> $]\!]_T$: Set

So what is 'std::vector' by itself (lets ignore the allocator)?

$\mu[\![$ std::vector $]\!]_T$: ?

$\mu[\![$ std::vector $]\!]_T$: Set → Set

Magic: Dependent Types

$\mu[\![\text{std::pair}]\!]_T : \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$
$\mu[\![\text{std::pair}]\!]_T = \lambda a\ b.\ a \times b$

What about std::pair in an expression context? Here we have two type parameters and two value parameters.

$$\mu[\![\text{std::pair}]\!]_E = \lambda t_0\ t_1\ v_0\ v_1.\ (\ v_0, v_1\ )$$

What about the type?

$$\mu[\![\text{std::pair}]\!]_E\ :\ ?$$

$\mu[\![\text{std::pair}]\!]_E : \text{Set} \rightarrow \text{Set} \rightarrow\ ? \rightarrow\ ? \rightarrow\ ?$

$\mu[\![\text{std::pair}]\!]_E : (t_0 : \text{Set}) \rightarrow (t_1 : \text{Set}) \rightarrow t_0 \rightarrow t_1 \rightarrow t_0 \times t_1$

Magic: Dependent Types

Dependent function type

General form is $(x : t) \rightarrow e(x)$.

- x is an identifier.
- t is a type.
- e(x) is an expression that uses 'x' in it.

Dependent Types 2

Another kind of dependent type.

type_and_value : (a : Set) × a
type_and_value = (Int,3)

Here we have a pair of a type and an element of that type.

make_type_and_value : (a : Set) → a → (b : Set) × b
make_type_and_value t v = (t, v)

Implicit types

Recall the type of std::pair's meaning:

$\mu[\![\text{std::pair}]\!]_E : (t_0 : \text{Set}) \rightarrow (t_1 : \text{Set}) \rightarrow t_0 \rightarrow t_1 \rightarrow t_0 \times t_1$

std::make_pair also has all these arguments, but the type arguments are implied. Use {} for implied types

$\mu[\![\text{std::make\_pair}]\!] : \{t_0 : \text{Set}\} \rightarrow \{t_1 : \text{Set}\} \rightarrow t_0 \rightarrow t_1 \rightarrow t_0 \times t_1$

Now std::make_pair's meaning can be called with only two arguments.

Denotational Design

What is a movie?

What is a movie?

- $\mu[\![\text{Movie}]\!] : \text{Set} \to \text{Set}$
- $\mu[\![\text{Movie}]\!] = \lambda\ a \to (\mathbb{R} \to a)$

Operations:

$\mu[\![\text{always}]\!] : \{ A : \text{Set} \} \to A \to \mu[\![\text{Movie}]\!]\ A$

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

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

$\mu[\![\text{snapshot}]\!]\ m\ t = m\ t$

What is a movie?

μ⟦ leftTrim ⟧ : { A : Set } → ℝ → μ⟦Movie⟧ A → μ⟦Movie⟧ (μ⟦boost::optional⟧$_T$ A)

μ⟦ leftTrim ⟧ trimTime m = λ t.
    if (t < trimTime)
      then μ⟦boost::none⟧
      else μ⟦boost::make_optional⟧ (m t)

μ⟦ app ⟧ : { A : Set } → { B : Set } → μ⟦Movie⟧ (A → B) → μ⟦Movie⟧ A → μ⟦Movie⟧ B

Semantics to C++

```
template< typename T >
Movie<T> cutAndPlace(
            Movie<T> mb, Time bStart, Time bDuration,
            Movie<T> ma, Time aPlacement )
{
   const auto leftTrimmed = trimLeft( mb, bStart )
   const auto rightTrimmed = trimRight( leftTrimmed, bStart + bDuration );
   // join converts Movie<optional<optional<T>>> to Movie< optional<T> >
   const auto joined = join( rightTrimmed );
   const auto shifted = shift( joined, bStart - aPlacement );

   // μ⟦overlay⟧ : {a : Set} → Movie a → Movie (optional a) → Movie a
   return overlay( ma, shifted );
}
```

Denotational Design

1. Discover the essence of the problem you'd like to solve in pure mathematics augmented with Agda notation.
2. Implement the solution efficiently in C++ while *retaining the interface* your semantics imply.

What is the meaning of this?

```
class Measurement
{
public:
    void setId( const int id )
    int id() const;
    std::string name;
    virtual void storeToDisk(...);
    virtual void loadFromDisk(...);
    // when returns false, this isn't valid for the geography
    virtual bool validate( Geography & )=0;
    virtual ~Measurement();
protected:
    virtual int calculateId() = 0;
}
```

What is the meaning of this?

elsewhere…

```
class GrassMeasurement : public Measurement
{
public:
    virtual void storeToDisk(...);
    virtual void loadFromDisk(...);
    virtual bool validate( Geography &);
    Color greenThreshold;
protected:
    virtual int calculateId();
}

GrassMeasurementResult makeGrassMeasurement(
  GrassMeasurement & m,
  Geography & g );
```

Meaning of Measurement

What is a measurement?

$\mu[\![\text{Measurement}]\!]_T = (a : \text{Set}, \text{Geography} \to a)$

What is a grass measurement?

$\mu[\![\text{GrassMeasurement}]\!]_E : \text{Color} \to \text{Measurement}$

$\mu[\![\text{GrassMeasurement}]\!]_E =$

$\lambda\ \text{greenThreshold} \to (\text{GrassMeasurementResult}$

$, (\lambda\ \text{geography} \to \ldots))$

Implementation of Measurement

$\mu[\![\text{Measurement}]\!]_T = (a : \text{Set}, \text{Geography} \rightarrow a)$

```
template< typename T >
struct Measurement
{
   typedef T measurement_result;
   virtual T run_measurement( const Geography & ) const=0;
};
```

Implementation of Measurement

$\mu[\![\text{Measurement}]\!]_T = (a : \text{Set}, \text{Geography} \rightarrow a)$

```
template< typename T >
struct Measurement
{
   typedef T result_type;
   virtual T operator()( const Geography & ) const=0;
};
```

Id thing

$\mu[\![\text{Measurement}]\!]_T = (a : \text{Set}, \text{Geography} \rightarrow a)$

What about that id thing?

```
template< typename T, typename Derived >
struct Measurement
{
   typedef T result_type;
   virtual T operator()( const Geography & ) const=0;
   virtual bool operator==( const Derived &) const=0;
};
```

Validation Thing

What about that validation thing?

$\mu[\![\text{Measurement}]\!]_T = (a : \text{Set}, \text{Geography} \rightarrow a)$

$\mu[\![\text{Measurement}]\!]_T = (a : \text{Set}, \text{Geography} \rightarrow \text{Optional } a)$

```
template< typename T, typename Derived >
struct Measurement
{
    typedef boost::optional<T> result_type;
    virtual result_type operator()( const Geography & ) const=0;
    virtual bool operator==( const Derived &) const=0;
};
```

Validation Speed

μ⟦ Measurement ⟧$_T$ = (a : Set, Geography → Optional a)

```
template< typename T, typename Derived >
struct Measurement
{
    typedef boost::optional<T> result_type;
    virtual result_type operator()( const Geography & ) const=0;
    virtual bool operator==( const Derived &) const=0;
    virtual bool returns_value( const Geography & g ) const
    {
        return bool( this->operator()( g ) );
    }
};
```

μ⟦ m.returns_value(g) ⟧ = boolFromOptional ((second μ⟦m⟧) μ⟦g⟧)

Grass Measurement

$\mu[\![\text{GrassMeasurement}]\!]_E : \text{Color} \rightarrow \text{Measurement}$

$\mu[\![\text{GrassMeasurement}]\!]_E = \lambda\ \text{greenThreshold} \rightarrow (\ \text{GrassMeasurementResult}$

$, (\lambda\ \text{geography} \rightarrow \ldots)\ )$

```
struct GrassMeasurement
   : Measurement< GrassMeasurementResult, GrassMeasurement>
{
    optional<GrassMeasurementResult> operator()( const Geography & ) const{...}
    bool operator==( const GrassMeasurement & gm) const
                {  return greenThreshold == gm.greenThreshold;     }
    bool returns_value( const Geography & g ) const { ... }
    GrassMeasurement( Color greenThreshold ) {...};
    Color greenThreshold;
};
```

Serialization

$\mu[\![\text{Measurement}]\!]_T = (a : \text{Set}, \text{Geography} \rightarrow a)$

1. Use Boost.Serialization for all the subclasses.
2. Make an AnyMeasurement type.

```
typedef boost::mpl::set
  < GrassMeasurement
   , AnotherMeasurement
   , …
   > MeasurementTypes;

typedef boost::make_variant_over<MeasurementTypes>
  AnyMeasurement;
```

AnyMeasurement

$\mu[\![\text{Measurement}]\!]_T = (a : \text{Set}, \text{Geography} \rightarrow a)$

AnyMeasurement looks a lot like a Measurement!

```
typedef boost::mpl::map
  < boost::result_of
    < _1 ( Geography ) >
  , MeasurementTypes
  > MeasurementResultTypes;
typedef boost::make_variant_over
  < MeasurementResultTypes > AnyMeasurementResult;
```

AnyMeasurementType

```
struct AnyMeasurement
  : Measurement< AnyMeasurementResult, AnyMeasurement >
{
  typedef boost::make_variant_over< MeasurementTypes > Impl;
  Impl impl;

  bool operator==( AnyMeasurement & m ) const { return m.impl
== impl; }
  optional< AnyMeasurementResult > operator()( const
Geography & g )
  {
    boost::apply_visitor(...);
  }
  //...
};
```

•

•

Other operations native to semantics

```
μ⟦ Measurement ⟧T = ( a : Set, Geography → a )

μ⟦ joinMeasurements( m1, m2 ) ⟧
  = ( first μ⟦m1⟧ × first μ⟦m2⟧
    , λg. ( ( second μ⟦m1⟧) g
          , ( second μ⟦m2⟧) g
          )
    )

μ⟦ map( f, m ) ⟧ = ( result_of μ⟦f⟧ , λg. f ((second μ⟦m⟧) g) )

result_of : {a : Set} {b : Set} → (a → b) → Set
result_of {a} {b} _ = a
```

Beaultiful/Powerful API

```
auto grassAndRocksMeasurement =
  map
    ( []( GrassMeasurementResult & gmr, RocksMeasurementResult & rmr )
      {
        return gmr.aboveThreshold() && rmr.over100RocksFound();
      }
    , joinMeasurements
        ( GrassMeasurement( Color( 0, 1, 0 ) )
        , RocksMeasurement()
        )
    );

Geography geo = ...
if( grassAndRocksMeasurement.return_value( geo ) )
 std::cout << "Cannot determine if there are grass and rocks\n";
else
 std::cout << "Finding of grass and rocks: " << grassAndRocksMeasurement( geo ) <<
'\n';
```

The Intellectual Ascent to Agda

1. Discover the essence
2. Derive the implementation

- Beautiful API's
- Screaming Speed
-