Reasoning with Types

Ben Deane

Sometime 2015

What is a type?

What do you think?

Let’s play a game

To help us get thinking about types.
I’ll tell you a type.
You tell me how many values it has.

Level 1

How many values?

```
bool;
```

Level 1

How many values?

```
char;
```

Level 1

How many values?

```
void;
```

Level 1

How many values?

```
struct Foo
{
};
```

Level 1

How many values?

```
enum Foo
{
  BAR,
  BAZ,
  QUUX
};
```

Level 1

How many values?

```
template <class T>
struct Foo
{
  T m_t;
};
```

End of Level 1

- Algebraically, a type is the number of values that inhabit it.

These types are equivalent

```
bool;

enum Foo
{
  BAR,
  BAZ
};
```

- Let's move on to level 2.

Level 2

How many values?

```
pair<char, bool>;
```

Level 2

How many values?

```
struct Foo
{
  char a;
  bool b;
};
```

Level 2

How many values?

```
tuple<bool, bool, bool>;
```

Level 2

How many values?

```
template <class T, class U>
struct Foo
{
  T m_t;
  U m_u;
};
```

End of Level 2

- When two types are "concatenated" into one compound type, we multiply the # of inhabitants of the components.
- This kind of compounding gives us a product type.
- On to Level 3.

Level 3

How many values?

```
optional<char>;
```

Level 3

How many values?

```
variant<char, bool>;
```

Level 3

How many values?

```
template <class T, class U>
struct Foo
{
  variant<T,U> m_v;
};
```

End of Level 3

- When two types are "alternated" into one compound type, we add the # of inhabitants of the components.
- This kind of compounding gives us a sum type.
- Caution: Miniboss detected ahead.

Level 4

How many values?

```
template <class T>
struct Foo
{
  variant<T,T> m_v;
};
```

Level 4

How many values?

```
template <class T>
struct Foo
{
  bool b;
  T m_t;
};
```

Level 4

How many values?

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

Level 4

Four possible values

```
bool f1(bool) { return true; }
bool f2(bool) { return false; }
bool f3(bool b) { return b; }
bool f4(bool b) { return !b; }
```

Miniboss: Algebraic Conundrum "Function"

How many values?

```
char f(bool);
```

Miniboss: Algebraic Conundrum "Function"

How many values (for `f`)?

```
enum Foo
{
  BAR,
  BAZ,
  QUUX
};
char f(Foo);
```

Miniboss: Algebraic Conundrum "Function"

How many values?

```
template <class T, class U>
U f(T);
```

Victory!

- The type of a function from A to B has B^A possible values.
- Hence a curried function is equivalent to its uncurried alternative:

$$
\begin{aligned}
F_{uncurried} :: (A, B) \rightarrow C &\Leftrightarrow C^{A*B} \\
&= C^{B*A} \\
&= (C^B)^A \\
&\Leftrightarrow (B \rightarrow C)^A \\
&\Leftrightarrow F_{curried} :: A \rightarrow (B \rightarrow C)
\end{aligned}
$$

- WARNING: Boss detected ahead!

Boss: Algebraic Enigma "Data Structure"

How many values?

```
template <typename T>
class vector<T>;
```

Boss: Algebraic Enigma "Data Structure"

We can define a `vector<T>` recursively:

$$v(t) = 1 + tv(t)$$

And rearrange…

$$\begin{aligned} v(t) - tv(t) &= 1 \\ v(t)(1 - t) &= 1 \\ v(t) &= \frac{1}{1 - t} \end{aligned}$$

Boss: Algebraic Enigma "Data Structure"

$$v(t) = \frac{1}{1-t}$$

What does it mean? Let's ask Wolfram Alpha.

Boss: Algebraic Enigma "Data Structure"

A `vector<T>` can have:

- 0 elements (1)
- 1 element (t)
- 2 elements (t^2)
- etc...

Boss: Algebraic Enigma "Data Structure"

How many values?

$$\texttt{vector<T>} \Leftrightarrow 1 + t + t^2 + t^3 + \ldots$$
$$= \frac{1}{1-t}$$

Victory!

Reasoning about types in an algebraic way allows us to discover equivalent formulations for APIs, Data Structures, etc which may be more natural or more efficient.

Let's play another game

I'll give you a mystery function type.
You tell me possible ways to write and name the function.
There's one rule: I insist on total functions.

What's That Function?

Name/Implement `f`

```
template <class T>
T f(T);
```

What's That Function?

Name/Implement `f`

```
template <class T, class U>
T f(pair<T,U>);
```

What’s That Function?

Name/Implement `f`

```
template <class T>
T f(bool, T, T);
```

What's That Function?

Name/Implement f

```
template <class T, class U>
U f(function<U(T)>, T);
```

What's That Function?

Name/Implement `f`

```
template <class T>
vector<T> f(vector<T>);
```

What's That Function?

Name/Implement `f`

```
template <class T>
T f(vector<T>);
```

What's That Function?

Name/Implement f

```
template <class T>
optional<T> f(vector<T>);
```

What's That Function?

Name/Implement `f`

```
template <class T, class U>
vector<U> f(function <U(T)>, vector<T>);
```

What's That Function?

Name/Implement `f`

```
template <class T>
T f(optional<T>);
```

What’s That Function?

Name/Implement f

```
template <class K, class V>
V f(map<K,V>, K);
```

What's That Function?

Name/Implement `f`

```
template <class K, class V>
optional<V> f(map<K,V>, K);
```

Victory!

Type signatures can tell us a lot about functionality. Using the type system appropriately and writing total functions makes interfaces safer to use.

The rabbit hole goes deeper

- Algebraic data type (Wikipedia)
- The Algebra of Algebraic Data Types (blog)
- The Algebra of Algebraic Data Types (video)

Let's hope C++ gets sum types (variant) in the standard soon…

Goals for well-typed interfaces

- Achieve formulations that:
  - are more natural
  - perform better
- Write total functions
- Make illegal states unrepresentable