Reasoning with Types

Ben Deane

Sometime 2015

What is a type?

What do you think?

What is a type?

- A way for the compiler to know what opcodes to output (dmr's motivation)?
- The way data is stored (representational)?
- Characterised by what operations are possible (behavioural)?
- Determines the values that can be assigned?
- Determines the meaning of the data?

What is a type?

- We can think about axes of type systems
  - static vs dynamic
  - strong vs weak
  - structural vs nominal
  - manifest vs inferred
  - dependent types?

For today's purposes...

- The set of values that can inhabit an expression
  - may be finite or "infinite"
  - characterized by cardinality
- Expressions have types
  - A program has a type

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?

```
bool;
```

Answer

2 (`true` and `false`)

Level 1

How many values?

```
char;
```

Level 1

How many values?

```
char;
```

Answer

256

Level 1

How many values?

```
void;
```

Level 1

How many values?

```
void;
```

Answer

0 (but cf function vs procedure)

Level 1

How many values?

```
struct Foo
{
};
```

Level 1

How many values?

```
struct Foo
{
};
```

Answer

1

Level 1

How many values?

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

Level 1

How many values?

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

Answer

3

Level 1

How many values?

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

Level 1

How many values?

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

Answer

`Foo` has as many values as `T`

End of Level 1

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

These types are equivalent

```
bool;

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

- Let's move on to level 2.

Level 2

How many values?

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

Level 2

How many values?

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

Answer

256 * 2 = 512

Level 2

How many values?

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

Level 2

How many values?

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

Answer

256 * 2 = 512

Level 2

How many values?

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

Level 2

How many values?

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

Answer

2 * 2 * 2 = 8

Level 2

How many values?

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

Level 2

How many values?

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

Answer

T * 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?

```
optional<char>;
```

Answer

$256 + 1 = 257$

Level 3

How many values?

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

Level 3

How many values?

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

Answer

256 + 2 = 258

Level 3

How many values?

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

Level 3

How many values?

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

Answer

$T + U$

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
{
  variant<T,T> m_v;
};
```

Answer

$T + T = 2T$

Level 4

How many values?

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

Level 4

How many values?

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

Answer

2T (equivalent to `variant<T,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?

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

Answer

256 * 256 = 65536

Miniboss: Algebraic Conundrum "Function"

How many values (for `f`)?

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

Miniboss: Algebraic Conundrum "Function"

How many values (for `f`)?

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

Answer

$2^8 * 2^8 * 2^8 = 2^{24}$

Miniboss: Algebraic Conundrum "Function"

How many values?

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

Miniboss: Algebraic Conundrum "Function"

How many values?

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

Answer

U^T

Victory!

- The type of a function from A to B has B^A possible values.

Victory!

- 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?

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

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.

It also helps us prevent errors by making illegal states unrepresentable.

Make Illegal States Unrepresentable

```
class InterfaceImpl : ...
{
  ...
  ConnectionState m_connectionState;
  ...
  ConnectionId m_connectionId;
  Timer* m_reconnectTimer;
  Region m_connectedRegion;
  u64 m_sessionToken;
  ...
};
```

- Some data members are dependent on others?
- Use types to express this

Make Illegal States Unrepresentable

```
class Friend
{
  ...
  std::string m_friendNote;
  bool m_friendNotePopulated;
  ...
};
```

- We still use `bool` to guard access/provide lazy initialization?
- We could use `optional` instead

Make Illegal States Unrepresentable

- Construct in a legal state
  - or you'll be checking *everywhere*
- Any time you have a state variable
  - consider pushing dependent state down into an object
- Look for state in the wrong place
  - per instance vs per class
  - take care over caching
- Consider `optional` vs `bool` or pointers
  - maybe `weak_ptr` makes sense for externalized state

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 <u>total</u> functions.

What's That Function?

Name/Implement `f`

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

What's That Function?

Name/Implement `f`

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

Answer

`identity`

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

Possible answer

```
first
```

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>
T f(bool, T, T);
```

Possible answer

select

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

Possible answer

`apply`

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

Possible answers

`shuffle`, `reverse`

What’s That Function?

Name/Implement `f`

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

What's That Function?

Name/Implement `f`

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

Possible answer

Not possible! (partial function)

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

Possible answer

`head`

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

Possible answer

map

What's That Function?

Name/Implement `f`

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

What's That Function?

Name/Implement `f`

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

Possible answers

`filter`, `partition`

What’s That Function?

Name/Implement `f`

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

What's That Function?

Name/Implement `f`

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

Possible answer

Not possible!

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

Possible answer

Not possible! But nevertheless:

```
V& map<K,V>::operator[](const K&);
```

What's That Function?

Name/Implement f

```
template <class K, class V>
optional<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);
```

Possible answer

`lookup`

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)
- Effective ML(Making Illegal States Unrepresentable)

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

Reasoning with types helps with all of this - try it!