USING TYPES EFFECTIVELY

BEN DEANE

bdeane@blizzard.com

SEPTEMBER 19TH, 2016

USING TYPES EFFECTIVELY?

WHAT DOES THAT MEAN?

The recent evolution of C++ is (from one point of view) largely about **strengthening** and **expanding** the **capabilities for dealing with types**.

- expansion of `type_traits`
- `decltype` to utter types
- `auto` to preserve types, prevent conversions, infer return types
- `nullptr` to prevent `int` / pointer confusion
- scoped `enum`
- GSL: `owner<T>`, `not_null<T>`
- Concepts TS

FP ISN'T (ONLY) ABOUT

- first class functions
- higher order functions
- lexical scoping, closures
- pattern matching
- value semantics
- immutability
- concurrency through immutability
- laziness
- garbage collection
- boxed data types / "inefficient" runtime models
- the M-word

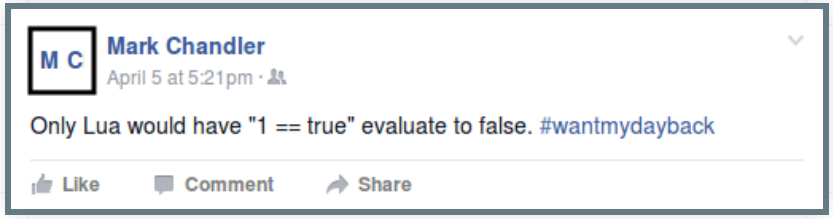
FP IS (ALSO, IMPORTANTLY) ABOUT

- using types effectively and expressively
- making illegal states unrepresentable
- making illegal behaviour result in a type error
- using total functions for easier to use, harder to misuse interfaces

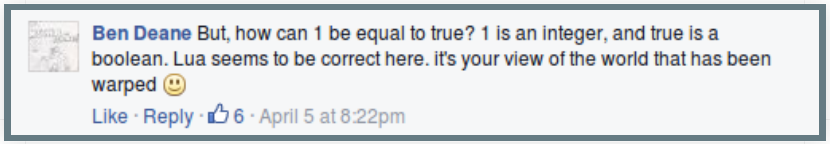
WHAT IS A TYPE?

- 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?



"Only Lua would have '`1 == true`' evaluate to `false`. #wantmydayback"



"But, how can `1` be equal to `true`? `1` is an integer, and `true` is a boolean. Lua seems to be correct here. It's your view of the world that has been warped."

(Smiley faces make criticism OK!)

WHAT IS A TYPE?

- 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.

There are no tricks: if it seems obvious, it is!

LEVEL 1

Types as sets of values

LEVEL 1

How many values?

```
bool;
```

2 (`true` and `false`)

LEVEL 1

How many values?

```
char;
```

256

LEVEL 1

How many values?

```
void;
```

0

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

```
struct Bar { template <typename T> Bar(); };
```

LEVEL 1

How many values?

```
struct Foo {};
```

1

LEVEL 1

How many values?

```
enum FireSwampDangers : int8_t {
  FLAME_SPURTS,
  LIGHTNING_SAND,
  ROUSES
};
```

3

LEVEL 1

How many values?

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

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 InatorButtons {
  ON_OFF,
  SELF_DESTRUCT
};
```

Let's move on to level 2.

LEVEL 2

Aggregating Types

LEVEL 2

How many values?

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

256 * 2 = 512

LEVEL 2

How many values?

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

256 * 2 = 512

LEVEL 2

How many values?

```
std::tuple<bool, bool, bool>;
```

2 * 2 * 2 = 8

LEVEL 2

How many values?

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

(# of values in `T`) * (# of values in `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

Alternating Types

LEVEL 3

How many values?

```
std::optional<char>;
```

256 + 1 = 257

LEVEL 3

How many values?

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

256 + 2 = 258

LEVEL 3

How many values?

```
template <typename T, typename U>
struct Foo {
  std::variant<T, U>;
}
```

(# of values in T) + (# of values in 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.

LEVEL 4

Function Types

LEVEL 4

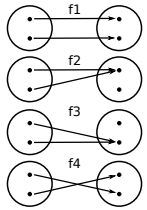
How many values?

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

4

LEVEL 4

Four possible values



LEVEL 4

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

LEVEL 4

How many values?

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

256 * 256 = 65,536

LEVEL 4

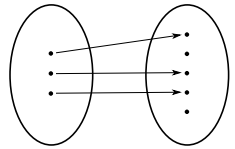
How many values (for `f`)?

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

256 * 256 * 256 = 16,777,216

LEVEL 4

The number of values of a function is the number of different ways we can



draw arrows between the inputs and the outputs.

LEVEL 4

How many values?

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

$$|U|^{|T|}$$

END OF LEVEL 4

When we have a function from A to B, we raise the # of inhabitants of B to the power of the # of inhabitants of A.

END OF LEVEL 4 (COROLLARY)

Hence a curried function is equivalent to its uncurried alternative.

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

VICTORY!

EQUIVALENCES

```
template <typename T>
struct Foo {
  std::variant<T, T> m_v;
};

template <typename T>
struct Bar {
  T m_t;
  bool m_b;
};
```

We have a choice over how to represent values. `std::variant` will quickly become a very important tool for proper expression of states.

This is one reason why `std::variant`'s "never-empty" guarantee is important.

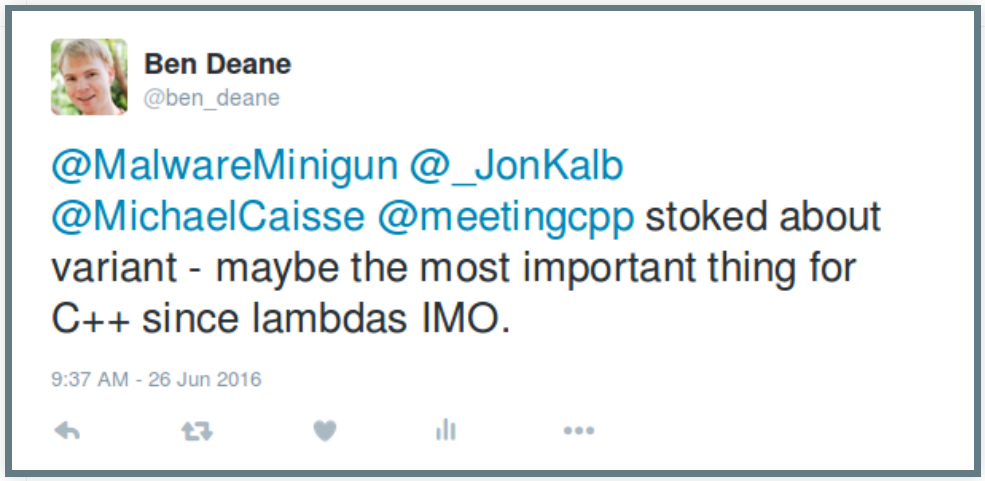
ALGEBRAIC DATATYPES

This is what it means to have an algebra of datatypes.

- the ability to reason about equality of types
- to find equivalent formulations
  - more natural
  - more easily understood
  - more efficient
- to identify mismatches between state spaces and the types used to implement them
- to eliminate illegal states by making them inexpressible

MAKING ILLEGAL STATES UNREPRESENTABLE

`std::variant` is a game changer because it allows us to (more) properly express types, so that (more) illegal states are unrepresentable.



MAKING ILLEGAL STATES UNREPRESENTABLE

Let's look at some possible alternative data formulations, using sum types (`variant`, `optional`) as well as product types (structs).

EXAMPLE: CONNECTION STATE

```
enum class ConnectionState {
  DISCONNECTED,
  CONNECTING,
  CONNECTED,
  CONNECTION_INTERRUPTED
};

struct Connection {
  ConnectionState m_connectionState;

  std::string m_serverAddress;
  ConnectionId m_id;
  std::chrono::system_clock::time_point m_connectedTime;
  std::chrono::milliseconds m_lastPingTime;
  Timer m_reconnectTimer;
};
```

EXAMPLE: CONNECTION STATE

```
struct Connection {
  std::string m_serverAddress;

  struct Disconnected {};
  struct Connecting {};
  struct Connected {
    ConnectionId m_id;
    std::chrono::system_clock::time_point m_connectedTime;
    std::optional<std::chrono::milliseconds> m_lastPingTime;
  };
  struct ConnectionInterrupted {
    std::chrono::system_clock::time_point m_disconnectedTime;
    Timer m_reconnectTimer;
  };

  std::variant<Disconnected,
               Connecting,
               Connected,
               ConnectionInterrupted> m_connection;
};
```

EXAMPLE: NULLABLE FIELD

```
class Friend {
  std::string m_alias;
  bool m_aliasPopulated;
  ...
};
```

These two fields need to be kept in sync everywhere.

EXAMPLE: NULLABLE FIELD

```
class Friend {
  std::optional<std::string> m_alias;
  ...
};
```

`std::optional` provides a sentinel value that is outside the type.

EXAMPLE: MONSTER AI

```
enum class AggroState {
  IDLE,
  CHASING,
  FIGHTING
};

class MonsterAI {
  AggroState m_aggroState;

  float m_aggroRadius;
  PlayerId m_target;
  Timer m_chaseTimer;
};
```

EXAMPLE: MONSTER AI

```
class MonsterAI {
  struct Idle {
    float m_aggroRadius;
  };
  struct Chasing {
    PlayerId m_target;
    Timer m_chaseTimer;
  };
  struct Fighting {
    PlayerId m_target;
  };

  std::variant<Idle, Chasing, Fighting> m_aggroState;
};
```

EXAMPLE: DESIGN PATTERNS

The addition of sum types to C++ offers an alternative formulation for some design patterns.

State machines and expressions are naturally modelled with sum types.

EXAMPLE: DESIGN PATTERNS

- Command
- Composite
- State
- Interpreter

DESIGNING WITH TYPES

`std::variant` and `std::optional` are valuable tools that allow us to model the state of our business logic more accurately.

When you match the types to the domain accurately, certain categories of tests just disappear.

DESIGNING WITH TYPES

Fitting types to their function more accurately makes code easier to understand and removes pitfalls.

The bigger the codebase and the more vital the functionality, the more value there is in correct representation with types.

USING TYPES TO CONSTRAIN BEHAVIOUR

We've seen how an expressive type system (with product and sum types) allows us to model state more accurately.

"Phantom types" is one technique that helps us to model the *behaviour* of our business logic in the type system. Illegal behaviour becomes a type error.

PHANTOM TYPES: BEFORE

```
std::string GetFormData();

std::string SanitizeFormData(const std::string&);

void ExecuteQuery(const std::string&);
```

An injection bug waiting to happen.

PHANTOM TYPES: THE SETUP

```
template <typename T>
struct FormData {
  explicit FormData(const string& input) : m_input(input) {}
  std::string m_input;
};

struct sanitized {};
struct unsanitized {};
```

`T` is the "Phantom Type" here.

PHANTOM TYPES: AFTER

```
FormData<unsanitized> GetFormData();

std::optional<FormData<sanitized>>
SanitizeFormData(const FormData<unsanitized>&);

void ExecuteQuery(const FormData<sanitized>&);
```

TOTAL FUNCTIONS

A *total function* is a function that is defined for all inputs in its domain.

```
template <typename T> const T& min(const T& a, const T& b);

float sqrt(float f);
```

LET'S PLAY ANOTHER GAME

To help us see how total functions with the right types can result in unsurprising code.

I'll give you a function signature with no names attached.

You tell me what it's called... (and you'll even know how to implement it).

The only rule... it must be a *total* function.

NAME THAT FUNCTION

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

identity

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

NAME THAT FUNCTION

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

first

NAME THAT FUNCTION

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

---

select

NAME THAT FUNCTION

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

---

apply or call

NAME THAT FUNCTION

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

---

reverse, shuffle, ...

NAME THAT FUNCTION

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

Not possible! It's a partial function - the `vector` might be empty.

```
T& vector<T>::front();
```

NAME THAT FUNCTION

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

NAME THAT FUNCTION

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

transform

NAME THAT FUNCTION

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

remove_if, partition, ...

NAME THAT FUNCTION

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

Not possible!

NAME THAT FUNCTION

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

Not possible! (The key might not be in the `map`.)

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

NAME THAT FUNCTION

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

---

lookup

WHAT JUST HAPPENED?

I gave you *almost nothing*.

No variable names. No function names. No type names.

Just bare type signatures.

You were able to tell me exactly what the functions should be called, and likely knew instantly how to implement them.

You will note that partial functions gave us some issues…

WELL-TYPED FUNCTIONS

Writing *total functions* with well-typed signatures can tell us a lot about functionality.

Using types appropriately makes interfaces unsurprising, safer to use and harder to misuse.

Total functions make more test categories vanish.

ABOUT TESTING...

In a previous talk, I talked about unit testing and in particular property-based testing.

Effectively using types can reduce test code.

Property-based tests say "for all values, this property is true".

That is exactly what types *are*: universal quantifications about what can be done with data.

Types scale better than tests. Instead of TDD, maybe try TDD!

FURTHER DOWN THE RABBIT HOLE

- http://en.wikipedia.org/wiki/Algebraic_data_type
- http://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-algebraic-data-types/
- https://vimeo.com/14313378 (Effective ML: Making Illegal States Unrepresentable)
- http://www.infoq.com/presentations/Types-Tests (Types vs Tests: Strange Loop 2012)

THANKS FOR LISTENING

"On the whole, I'm inclined to say that when in doubt, make a new type."

– Martin Fowler, *When to Make a Type*

"Don't set a flag; set the data."

– Leo Brodie, *Thinking Forth*

GOALS FOR WELL-TYPED CODE

- Make illegal states unrepresentable
- Use `std::variant` and `std::optional` for formulations that
  - are more natural
  - fit the business logic state better
- Use phantom types for safety
  - Make illegal behaviour a compile error
- Write total functions
  - Unsurprising behaviour
  - Easy to use, hard to misuse

EPILOGUE

A taste of algebra with datatypes

A TASTE OF ALGEBRA WITH DATATYPES

How many values?

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

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

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

(empty vector or (+) head element and (*) tail vector)

A TASTE OF ALGEBRA WITH DATATYPES

And rearrange...

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

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

$$v(t)(1 - t) = 1$$

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

What does that mean? Subtracting and dividing types?

A TASTE OF ALGEBRA WITH DATATYPES

When we don't know how to interpret something mathematical?

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

Let's ask Wolfram Alpha.

A TASTE OF ALGEBRA WITH DATATYPES

Series expansion at $t = 0$:

$$1 + t + t^2 + t^3 + t^4 + \ldots$$

A `vector<T>` can have:

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

GOALS FOR WELL-TYPED CODE

- Make illegal states unrepresentable
- Use `std::variant` and `std::optional` for formulations that
  - are more natural
  - fit the business logic state better
- Use phantom types for safety
  - Make illegal behaviour a compile error
- Write total functions
  - Unsurprising behaviour
  - Easy to use, hard to misuse