Types and Type Systems

surprise! there's calculus in C++

Jack Leightcap

Northeastern University Wireless Club

October 21, 2021

Types: Intro

C++

```
unsigned long length(std::String s) {
    // ...
}
```

Python

```
def length(s):
    # ...
```

Types: Intro (cont.)

what's even the point of types?

C++: Type Checking

```
unsigned long l = length(2 + 3);
// fails at compile-time
// you can't even run this program
```

Python: Duck Typing

```
l = length(2 + 3)
# does this fail? maybe.
```

Types: Type Systems

even built-in types can't save us. some more nuance,

```
uint32_t read_adc(void) {
    // returns the ADC reading in millivolts.
    return reading;
}
```

```
uint32_t volts = read_adc();
volts_to_expensive_delicate_device(volts);
```

this type checks, but does not convey the meaning we intended. **types didn't save us here**.

Type Systems: Adding Types

```
typedef struct {
    uint32_t mv;
} Millivolt;

// no longer compiles.
uint32_t v = read_adc(void);

// instead,
Millivolt v = { .mv = read_adc(void) };
// this is called a 'wrapper type'
```

Type Systems: Syntax

Wrapper Type

```
typedef struct {
    uint32_t mv;
} Millivolt;
```

becomes

```
newtype Millivolt = Millivolt Int
```

Structures

collection of things.

```
struct Record {
    std::string name;
    int value;
};
```

becomes

```
data Record = Record { name :: String, value :: Int }
```

Type Systems: Syntax (cont.)

Tagged Unions

exactly one of these possible things.

```
enum RecordType { A, B, C };
struct RecordA { int a; };
struct RecordB { int b; };
struct RecordC { int c; };
struct Record {
    enum RecordType type;
    union {
        struct RecordA a;
        struct RecordB b;
        struct RecordC c;
    } value;
};
```

becomes

```
data Record = RecordA Int | RecordB Int | RecordC Int
```

Algebra of Types

Definition	Size	Name
`data Unit = Unit`	1	
`data Bool = True | False`	2	
`data Maybe a = Just a | None`	$1 + a$	
`data Either a b = Left a | Right b`	$a + b$	*Sum Type*
`data (a, b) = (a, b)`	$a \times b$	*Product Type*

- Product Type $\rightarrow$ "a and b"
- Sum Type $\rightarrow$ "a or b"

Algebra of Types: Linked List

what size is a `LinkedList`?

Haskell

```haskell
data LinkedList a = Nil | Cons a (LinkedList a)
```

C++

```cpp
template <typename a>
struct LinkedList<a> {
    a first;
    LinkedList<a> * rest;
}
```

Algebra of Types: Linked List (cont.)

```
data LinkedList a = Nil | Cons a (LinkedList a)
```

$$
\begin{aligned}
L &= 1 + aL \\
&= 1 + a(a + aL) \\
&= 1 + a^2 + a^2(1 + aL) \\
&= 1 + a + a^2 + a^3 + \ldots
\end{aligned} \tag{1}
$$

a `LinkedList` is:

- 1
- a
- $a \times a$
- $a \times a \times a$
- $\ldots$

remember Taylor Series?

Algebra of Types: Linked List (cont.)

$$
\begin{aligned}
L &= 1 + aL \\
L(1 - a) &= 1 \\
L &= \boxed{\frac{1}{1 - a}}
\end{aligned}
\tag{2}
$$

instead of considering this as *derived*, you can *define* a `LinkedList` as this function.

Algebra of Types: Data Structures

```haskell
data BTree a = Leaf a | Branch (BTree a) (BTree a)
```

$$
\begin{aligned}
T &= a + T^2 \\
&= a + (a + T^2)^2 \\
&= a + (a + (a + T^2)^2)^2 \\
&= \dots \\
&= a + a^2 + 2a^3 + 5a^4 + 14a^5 + \dots
\end{aligned} \tag{3}
$$

(those are the *Catalan numbers*). how the did we even get here?

Algebra of Types: Computability

Curry-Howard Isomorphism

computer programs *are* mathematical proofs

Logic side	Programming side
universal quantification	generalised product type
existential quantification	generalised sum type
implication	function type
conjunction	product type
disjunction	sum type
true formula	unit type
false formula	bottom type

Algebra of Types: Bonus – `zipper`

a *zipper* is a data structure that 'focuses' on one element.

a linked list,

```
: -> : -> : -> : -> []
|    |    |    |
1    2    3    4
```

'focusing' that list on 3:

```
[] <- : <- : <- FOCUS -> : -> []
      |    |       |       |
      1    2       3       4
```

Algebra of Types: Bonus – Derivatives

what is the corresponding function for this zipper of a linked list?

$$\begin{aligned} L_z &= \frac{1}{(1-a)^2} \\ &= \frac{\partial L}{\partial a} \end{aligned} \tag{4}$$

this is true in general; if you want the zipper of a data structure, take its derivative.