Part 9

Advanced Types

Types Overview

- Types have names that are compared

```
int != float
```

- A major part of checking is finding type-mismatches in the code (type errors)
- "Nominal Typing"

Type Extensions

- Many languages introduce arrays and pointers

```
var a int[20];
bar b int *;
```

- These are perhaps viewed as type modifiers
- There is a "base type" involved
- An array is an extension to that type

Example Implementation

```
class BaseType:
    def __init__(self, name):
        self.name = name

class PointerTo:
    def __init__(self, type):
        self.type = type

class ArrayOf:
    def __init__(self, type, size):
        self.type = type
        self.size = size

/*  int *[10] */
ArrayOf(PointerTo(BaseType('int')), 10)
```

Derived Types

- Structures/Records

```
struct Point {
   x int;
   y int;
}
```

- An instance contains values for all fields

```
p = Point(2, 3);
print p.x;
print p.y;
```

- Related concept: tuples

Derived Types

- Enums/Unions

```
enum MaybeInt {
   Nothing;
   Just(int);
}
```

- An instance contains only <u>one</u> of the values

```
x = MaybeInt::Nothing;
y = MaybeInt::Just(42);
```

- Values are labeled. Use requires case matching

```
var a int = match y {
    Nothing => 0;
    Just(x) => x * 10;
}
```

Function Types

- Functions also represent a type

```
func mul(x int, y int) int {
   return x * y;
}
```

- Type consists of argument types and result

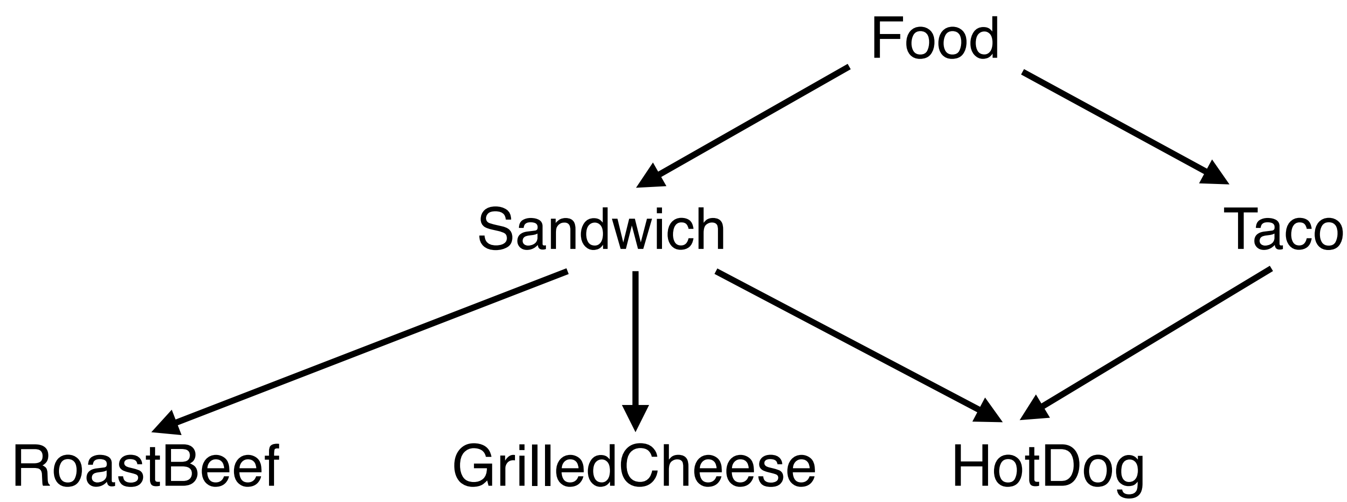
```
(int, int) -> int
```

- Note: Functions might be first-class objects just like integers, floats, etc.

```
var m = mul;
...
z = m(x, y);     # Requires x=int, y=int, z=int
```

Complexity: Ontologies

- Type system might support "inheritance"



- Suddenly you're now in OO hell...
- We're not doing that (but be aware of it)

Algebraic Data Types

- Modern programming languages often implement or cite the concept of an "algebraic type system"

> Enums are a feature in many languages, but their capabilities differ in each language. Rust’s enums are most similar to *algebraic data types* in functional languages, such as F#, OCaml, and Haskell.

- WHAT is that?!?!?

Algebra Review

- In math class, you build expressions that consist of sums and products

```
a + b*c
a*b + c*b
(a + c)*b
```

- There are also some "identities"

```
a * 1 = a          a + 0 = a          a * 0 = 0
1 * a = a          0 + a = a          0 * a = 0
```

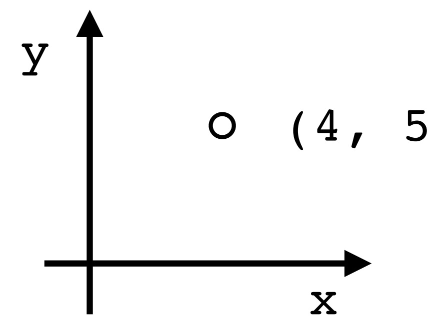
- And some rules (e.g., associativity, distribution)

```
(a * b) * c = a * (b * c)
(a + b) + c = a + (b + c)
a*(b + c) = a*b + a*c
```

Product Type

- A structure represents a "product"

```
struct Point {
   x int;
   y int;
}
```



- Known as a "product type"
- Think of it as a cartesian product
- How many possible values? (the product)

```
struct A {
   x bool;
   y bool;
}
```

```
{ false, false }
{ false, true }
{ true, false }
{ true, true }
```

2x2 possibilities

Sum Type

- An enum represents a <u>choice</u> of values

```
enum MaybeBool {
   Nothing;
   Just(bool);
}
```

- It's <u>one</u> of the possible values.

- The "sum" terminology is a bit weird, but it also represents all of the possibilities

```
enum A {
   Nothing;
   Just(bool);
}
```

⇨

```
Nothing
Just(false)
Just(true)
```

1 + 2 possibilities

Note: Enums

- Enums are essentially a "tagged" value

```
x = MaybeBool::Nothing;        ('Nothing', None)
y = MaybeBool::Just(True);     ('Just', True)
```

- Could be represented a 2-tuple

- Case analysis

```
if a[0] == 'Nothing':
    return -1;
elif a[0] == 'Just':
    return 1 if a[1] == True else 0
```

Algebraic Laws

- Multiplication

```
int * int  ────────▶  struct { int, int }
```

- Associativity

```
(int * int) * float     struct {
                           struct { int, int },
                           float
                        }

int * (int * float)     struct {
                           int,
                           struct { int, float}
                        }
```

- Basically the same thing (3 values together)

```
int * int * float ────▶ struct { int, int, float }
```

Algebraic Laws

- Addition

```
int + bool ──────▶ enum { int, bool }
```

- Associativity

```
(int + bool) + float    enum {
                            enum { int, bool },
                            float
                        }

int + (bool + float)    enum {
                            int,
                            enum { bool, float }
                        }
```

- Again, the same thing (Choice of 3 values)

```
int + bool + float ──▶ enum { int, bool, float }
```

Algebraic Laws

- unit - A singleton object (like Python None)
- Now consider this:

```
struct {
   a int;
   b unit;     // What values? (only one)
}
```

- You can get rid of unit. Why bother?

```
struct {
   a int;
   b unit;
}
```

```
struct {
   a int;
}
```

- Unit is the multiplicative identity (the 1)

```
int * unit = int
unit * int = int
```

Algebraic Laws

- Another interpretation of unit...
- Imagine you're code reviewing this class

```
class SomeClass:
    def __init__(self, x):
        self.x = x
        self.y = None
```

- Now, imagine that self.y is assigned no-where (you search millions of lines, never assigned)
- Bah! Delete it... totally unnecessary

```
class SomeClass:
    def __init__(self, x):
        self.x = x
```

Algebraic Laws

- void - A type that can never be instantiated

```
var void x;        // ERROR!  Can't instantiate void
```

- What if it's part of an enum?

```
enum {
   A(int);
   B(void);
}
```

- Can eliminate. You could never pick that option

```
enum {                    enum {
   A(int);         =>        A(int);
   B(void);               }
}
```

- Void is the additive identity (the 0)

```
int + void = int          int * void = void
void + int = int          void * int = void
```

also

Algebraic Laws

- Another interpretation of void
- Suppose you came across this class...

```
class Void:
    def __init__(self):
        # TO-DO
        raise TypeError("Never!")
```

- And this code...

```
if choice == 'A':
    x = 42
elif choice == 'B':
    x = Void()
```

- Forget that...delete (drop the void branch)

```
if choice == 'A':
    x = 42
```

Algebraic Laws

- Distributive property

```
a * (b + c) = a * b + a * c
```

- Consider:

```
enum MaybeInt {
    Nothing;
    Just(int);
}
```

```
struct A {
    x float;
    y MaybeInt;
}
```

- The same as rewriting like this

```
struct A_x {
    x float;
}
```

```
struct A_xy {
    x float;
    y int;
}
```

```
enum A {
    With_x(A_x);
    With_xy(A_xy);
}
```

- (Might have to squint a bit. I've also renamed)

Algebraic Type System

- "Algebraic type system" basically means that the type system is abstracted within this framework of algebraic products, sums, and identities
- It's more of a theoretical foundation for mathematical reasoning about types
- Comment: programming languages have had structs and enums for basically forever. Algebra is not an implementation/design requirement

Algebraic Types & Logic

- Logic: True, False

```
False and False = False        False or False = False
False and True = False         False or True = True
True and False = False         True or False = True
True and True = True           True or True = True
```

- You can map: unit -> True, void -> False

```
void * void = void             void + void = void
void * unit = void             void + unit = unit
unit * void = void             unit + void = unit
unit * unit = unit             unit + unit = unit
```

- A type system can encode logical statements
- Type checking => mathematical proof
- "Howard-Curry correspondence"