Types and type systems

- Type theory is any formal system which can serve as an alternative to set theory, and where logic is encoded in the types (e.g. algebraic types)
- Type theory was created to avoid paradoxes in formal logic and set theory. (Russel)
- Examples are: typed λ-calculus (Church), Martin-Löf type theory, Homotopy type theory
- Closely related to category theory and cartesian closed categories.
- Every <u>term</u> has a (single) <u>type</u>.
- We can losely think of types as (unique) sets of terms.
- Types are operationally important to virtually all programming languages (even) assembly language.
- The Howard-Curry-Lambek correspondence: Types as theorems, programs as proofs

- Type systems check for, and ensure soundness of our programs (to varying degree).
- Type systems exist to <u>forbid</u> various classes of <u>invalid</u> programs.
  (e.g JS {} + [] = 0!, {} + {} = NaN (sic)!)
- The more powerful the system, the more it can forbid more precisely, sometimes at the expense of the programmer, or forbidding legal programs.
- At an extreme we can <u>proove</u> the correctnes of our code:
  - Theorem and proof vs type and implementation
  - C vs. F# vs. Idris vs. Coq
- Dynamic languages do type-checking at runtime, right before execution.
- They give a lot of flexibility, at the expense of correctness

```
Ex.  if x == y:
       retun true
     else
       return "no"
```

- Type checkers consist of sets of rules, against which every type, expression, function, construct are tested. If none of the rules return true, it's an error.

- Static type systems are much more rigid and cumbersome than dynamic type systems.

  Ex.: • a function can only return one type
  • every branch must return the same type
  • etc.

- Strong static type systems (sytem F, ML, ..) can catch a lot of errors at compile time!

- Once mastered, it's hard working without a proper type system!

- Type checkers can be run "backwards": Type inference! The compiler figures it out

- Part of the difficulty learning FP in F# or Haskell is due to having to learn strong typing at the same time

Benefits of static, strong typing:

- Detecting errors at compile time
- Algebraic data types and making invalid state unrepresentable
- Documentation, intent a.k.a. follow the type (Hoogle in Haskell)
- Abstraction, TDD (Type driven development)
- Safety (dependent types and bounds)
- Efficiency; the more we know, the better we can optimize (safely)

Later:

- ADTs
- Polymorphism
- Generics
- Rank N polymorphism

Curious examples:

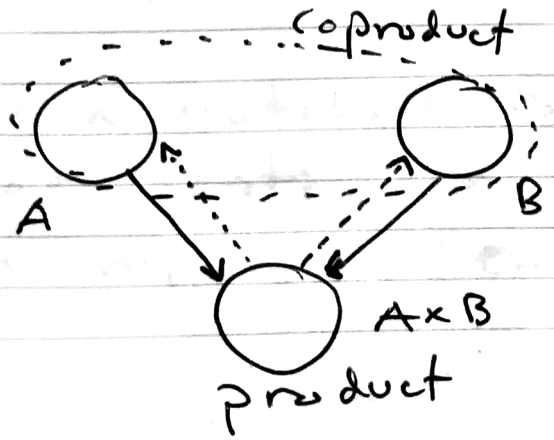
- Idris matmul
- Hallgren: add, mul, sort

Algebraic data types

- Most programming languages have record types, structs and/or tuples.
- These are known as <u>product types</u>, formed as cartesian products of sets: $A \times B : \{(a, b)\}$
- Much less common are union types, or <u>sum types</u> (coproduct) which are formed as the disjoint union of two sets: el, tag or variant

$$A + B : \{(a_n, 0), (b_n, 1)\}$$

- <u>Together</u> these form an algebra of conjunction and disjunction (AND and OR)



- Using the algebraic properties we can encode a lot of business logic in our types
- The type checker can then check the correctness of our code!

⇒ Make invalid state unrepresentable

- Avoid a lot of runtime tests: we know it cannot be wrong
- Always valid documentation!

- Product types are collections of several things
- Sums types are <u>one</u> of several things (ex. Int)

- Algebraic data types are deconstructed using <u>pattern matching</u>

- Pattern matching is (in principle) a static set of tests, and should not be confused with regexps. (except F# active patterns)

Algebraic data types in F#

- New types are declared i F# using the type keyword:

```
type Message = string
type Count = int
type Point = float*float
type Coord<'T> = 'T*'T*'T
```

- Record types:

```
type Coordinate = {
    Lat : float
    Lon : float
}
```

- When using records all members must be defined:

```
let pt = {Lat = 69.8; Lon = 12.1}
```

- Records can be updated (immutably):

```
let pt' = {pt with Lon = 11.6}
```

- Access: pt.Lat

- Discriminated unions :

```
type ThisOrThat =
    | This of string
    | That of string * int
    | Neither
```

(Type constructor: ThisOrThat; Data constructors: This, That, Neither)

- We create sum types using their data constructors:

```
let a = This "test"
let b = That ("meaning", 42)
let c = Neither
```

↑ all have type "ThisOrThat"

- Examples:

```
type Option<'a> = Some of 'a | Nothing
type Result<'a,'e> = Ok of 'a | Error of 'e
type List<'a> = Leaf of 'a * List<'a> | Nil
```

Pattern matching

- Pattern matching is a flexible way of deconstructing and extracting data from ADTs
- Used in match expressions, let-bindings and λ-functions.

Examples:

```
let pyth (a, b) = a**2 + b**2 |> sqrt
... |> fun [x; y; _] -> x + y

let head h::t = h

let tail _::t = t

let firstName {FirstName = x} = x

let f (_, x, _) as t = ...
```

Match expressions

- Match expressions are the most important device for pattern matching
- Match expressions can match on pretty much anything (active patterns)

Examples:

```
match str with
| "hello" -> "world"
| "world" -> "domination"
| _       -> "?"

match result with
| x when x = 1 -> ...
| x when x > 0 -> ...
| _            -> ...

match opt with
| Some x -> x
| None   -> 0
```