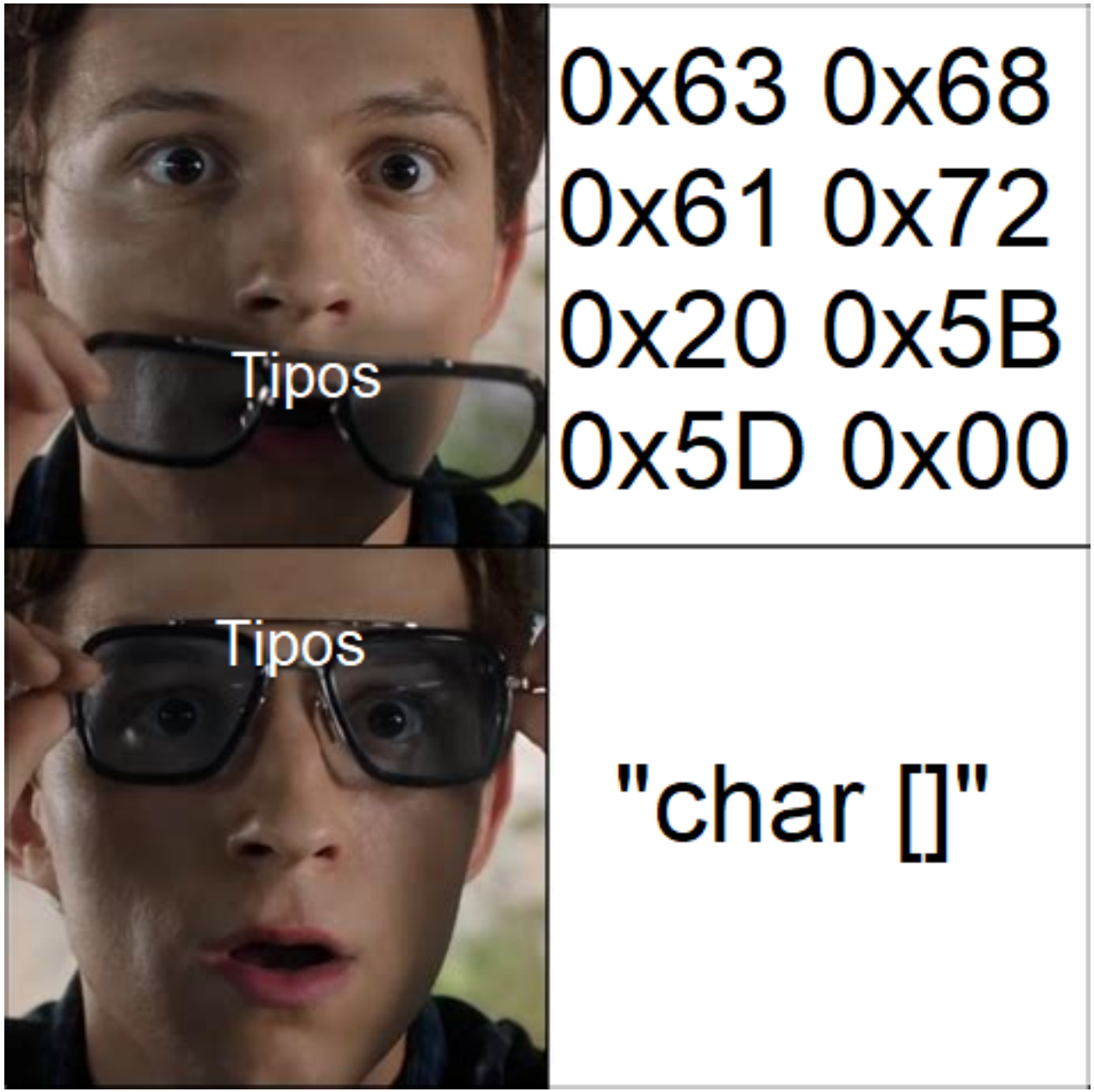
Tipos do Céu ao Chão

Hashimoto

22 de Junho de 2022

Meme



ToC

Intro

Notação

Alto nível:

- Tamanho de um tipo: $\#(A)$
- Dois tipos são isomorficos (equivalentes): $A \simeq B$
- Função que recebe A, B, C, e retorna D:

$$A \to B \to C \to D$$

Baixo nível:

- Inteiro com sinal de n bits: i_n
- Inteiro com sinal de 32 bits: $i32$
- Inteiro sem sinal de n bits: u_n
- Inteiro sem sinal de 32 bits: $u32$
- Booleano de tamanho byte: $bool$
  Nota: $bool \simeq u1$

Caso Recusivo

O "caso recursivo" é para criar *tipos novos*.

Para *usar* os tipos novos, precisamos de formas (funções) para:

- definir/criar novos elementos (*construtores*)
- transformar os elementos já definidos: (*destrutores*)

Nota: *Em lógica, os construtores são chamados de regras de introdução e os destrutores de regras de eliminação*

Produto $A \times B$

- Construtor

$$A \to B \to A \times B$$

- Destrutor (projeções)

$$\pi_A : A \times B \to A$$

$$\pi_B : A \times B \to B$$

- Valores

$$\#(A \times B) = \#(A) \times \#(B)$$

Corolário

- É associativo:

$$(A \times B) \times C \simeq A \times (B \times C)$$

- É comutativo:

$$(A \times B) \simeq (B \times A)$$

Exemplos: $A \times B$

- C *(Struct)*:

```c
struct ProdutoIntDouble {
    int a;
    double b;
};
```

- Java *(Class/Object)* (*):

```java
class ProdutoIntDouble {
    private int a;
    private double b;
}
```

- Haskell *(Record)*:

```haskell
data ProdutoIntDouble = PIntDouble
    { a :: Int
    , b :: Double
    }
```

Matemática de Struct

Informações importantes de uma *struct*:

- Tamanho *(sizeof)*: quantos bytes ocupa?
- Alinhamento *(alignment)*: endereço é múltiplo de quanto?

Nota: *Nem sempre o tamanho final é a soma dos tamanhos!*

Alinhamento:
www.delftstack.com/howto/c/struct-alignment-in-c/

Exemplos: Matemática de Struct

```
struct Exemplo1 {
    u32 a;
    u32 b;
};

struct Exemplo2 {
    u32 a;
    bool b;
};

struct Exemplo3 {
    u64 a;
    u32 b;
    u32 c;
};
```

```
struct Exemplo4 {
    u32 b;
    u64 a;
    u32 c;
};

struct Exemplo5 {
    u32 b; u32 pad1;
    u64 a;
    u32 c; u32 pad2;
};
```

Tipo	sizeof	alignment
bool	1	1
u32	4	4
u64	8	8

Respostas: Matemática de Struct

Tipo	sizeof	alignment	waste
bool	1	1	0
u32	4	4	0
u64	8	8	0
Exemplo1	8	4	0
Exemplo2	8	4	3
Exemplo3	16	8	0
Exemplo4	24	8	8
Exemplo5	24	8	0

Soma $A + B$

- Construtores

$$left : A \to A + B$$

$$right : B \to A + B$$

- (Família de) Destrutor(es)

$$\forall X \,.$$

$$A + B \to (A \to X) \to (B \to X) \to X$$

- Valores

$$\#(A + B) = \#(A) + \#(B)$$

- Outros nomes: Coproduto, Variante

Corolário

- É associativo:

$$(A + B) + C \simeq A + (B + C)$$

- É comutativo:

$$(A + B) \simeq (B + A)$$

Unions

Em C, temos *Unions*!

```
union quase_soma {
    int a;
    float b;
    char c;
};
```

Nota: *Depois que criamos um elemento da `union` perdemos a informação de qual caso ele representa!*

Isso aloca um espaço na memória que pode armazenar um `int` ou(-exclusivo) um `float` ou(-exclusivo) um `char`.

O *sizeof* é o $\max(sizeof(int), sizeof(float), sizeof(char))$.
O *alignment* é o $\max(align(int), align(float), align(char))$.

Enums

Em C, também temos *Enums*!

```
enum casos_soma {
      CASO_SOMA_A, CASO_SOMA_B, CASO_SOMA_C,
};
```

Isso dá "um nome" para alguns inteiros de 0 até $2^n - 1$ (em C $n := 32$, mas a princípio pode ser qualquer potência de 2)

O *sizeof* é o *sizeof*(*un*) (no caso do C é *sizeof*(*u32*)).
O *alignment* é o *align*(*un*) (no caso do C é *align*(*u32*)).

Tagged Unions

Agora podemos juntar as duas!

```
enum casos_soma {
    CASO_SOMA_A, CASO_SOMA_B, CASO_SOMA_C,
};

union quase_soma {
    int a; float b; char c;
};

struct soma {
    union quase_soma as;
    enum casos_soma tag;
};
```

Tagged Unions

Equivalente!

```
struct soma {
    union quase_soma {
        int a; float b; char c;
    } as;

    enum casos_soma {
        CASO_SOMA_A, CASO_SOMA_B,
        CASO_SOMA_C,
    } tag;
};
```

Matemática de Tagged Union

No fim, uma `Tagged Union` é só uma `struct` de um `enum` e uma `union`.

Exemplos: Matemática de Tagged Union

```
struct Exemplo1 {
    union { u32 a; u64 b; } as;
    enum { CASO_U32, CASO_U64, } tag; };

struct Exemplo2 {
    union { u32 a; u64 b; bool c; } as;
    enum { C_U32, C_U64, C_BOOL, } tag; };

struct Exemplo3 {
    union { u32 a; u32 b; u32 c; u32 d;
            u32 e; u32 f; u32 g; u32 h; } as;
    enum { A, B, C, D, E, F, G, H, } tag; };
```

Tipo	sizeof	alignment	Tipo	sizeof	alignment
bool	1	1	u32	4	4
			u64	8	8

Respostas: Matemática de Tagged Union

Tipo	sizeof	alignment
bool	1	1
u32	4	4
u64	8	8
Exemplo1	16	8
Exemplo2	16	8
Exemplo3	8	4

Exemplos: $A + B$

- `Zig` *(Tagged Union)*:

```
const UnionIntDouble = union(enum) {
    caso_int: u32,
    caso_double: f64,
};
```

Exemplos: $A + B$

- Java *(Inheritance/Dynamic Dispatch)* (*):

```
abstract class UnionBase {}

class UnionInt extends UnionBase {
    private int val;
}

class UnionDouble extends UnionBase {
    private double val;
}
```

Exemplo Bônus: $A + B$

- Algo parecido em C *("Struct Inheritance")*:

```
enum UnionTag { CASO_INT, CASO_DOUBLE };

struct UnionHeader {
    enum UnionTag tag;
}

struct UnionInt {
    struct UnionHeader header;
    int val;
}

struct UnionDouble {
    struct UnionHeader header;
    double val;
}
```

Exemplos: $A + B$

- `Rust` *(Enum)*:

```
enum UnionIntDouble {
    CasoInt(i32),
    CasoDouble(f64)
}
```

- `Haskell` *(Algebraic Data Type)*:

```
data UnionIntDouble
    = CasoInt Int
    | CasoDouble Double
```

Exponencial B^A

- Construtor

$$(A \to B) \to B^A$$

- Destrutor

$$eval : B^A \to A \to B$$

- Valores

$$\#\left(B^A\right) = \#(B)^{\#(A)}$$

Corolário

- **Não** é associativo:

$$\left(A^B\right)^C \not\cong A^{(B^C)}$$

- **NÃO** é comutativo:

$$\left(A^B\right) \not\cong \left(B^A\right)$$

Exponencial B^A

Opções de implementação:

- Look-up Table (Arrays)
  - Memória: $sizeof(B) \times \#(A)$
    Nota: *Precisa converter A para inteiro*

Exponencial B^A

Opções de implementação:

- Look-up Table (Arrays)
  - Memória: $sizeof(B) \times \#(A)$
    Nota: *Precisa converter A para inteiro*

- Ponteiros para função:
  - Memória: *código* (estático) + *ponteiro* (cada instância)

Exponencial B^A

Opções de implementação:

- Look-up Table (Arrays)
  - Memória: $sizeof(B) \times \#(A)$
    Nota: *Precisa converter A para inteiro*
- Ponteiros para função:
  - Memória: *código* (estático) + *ponteiro* (cada instância)
- (Código de uma) Máquina Virtual

Exponencial B^A

Opções de implementação:

- Look-up Table (Arrays)
  - Memória: $sizeof(B) \times \#(A)$
    Nota: *Precisa converter A para inteiro*
- Ponteiros para função:
  - Memória: *código* (estático) + *ponteiro* (cada instância)
- (Código de uma) Máquina Virtual
- ??? (Céu é o limite)

Por que B^A?

Um grande motivo é *Programação Funcional*:

- Receber/Passar uma função
- Closures
- Aplicação Parcial
- Callbacks

Por que B^A?

Um grande motivo é *Programação Funcional*:

- Receber/Passar uma função
- Closures
- Aplicação Parcial
- Callbacks

Também serve como base para *Virtual Tables (Vtables)*:

- Intefaces
- Dynamic Dispatch

Caso Base

- Aqui ficam os *tipos primitivos/atômicos*
- Não dá para quebrar mais que isso!
- Também têm *contrutores* e *destrutores*

Void

Pausa Dramática!

Void

- Construtor

  (não tem)

- Destrutor

  $$\forall X \,.$$

  $$absurd : Void \to X$$

- Valores

  $$0$$

- **Nota:** *Em lógica, é o* $\bot$ *(false)*

Unity – ()

- Construtor

$$() : ()$$

- Destrutor

(não tem)

- Valores

$$1$$

- **Nota:** *Em lógica, é o* $\top$ *(true)*

void == ()

Se "void == ()", por que eu não posso fazer isso:

```
void func(void) {}

int main() {
    void v;
    v = func();
}
```

void == ()

```
> gcc a.c
a.c: In function 'main':
a.c:4:10: error: variable or field
'v' declared void
    4 |     void v;
      |          ^
a.c:5:7: error: void value not ignored
as it ought to be
    5 |     v = func();
      |
```

void == ()

Em Zig, pode!

```
fn func() void {}

pub fn main() u8 {
    var v: void = undefined;
    v = func();
    return 0;
}
```

Bool

- Construtor

$$false : Bool$$

$$true : Bool$$

- (Família de) Destrutor(es)

$$\forall X \,.$$

$$Bool \to X \to X \to X$$

- Valores

$$2$$

$\text{Bool} \simeq () + ()$

Se a gente usar a Soma $() + ()$:

- Construtores

$$\mathit{left} : () \to () + ()$$

$$\mathit{right} : () \to () + ()$$

- (Família de) Destrutor(es)

$$\forall X \,.$$

$$() + () \to (() \to X) \to (() \to X) \to X$$

- Valores

$$\#(() + ()) = \#(()) + \#(()) = 2$$

- Só chamar *left* de `false` e *right* de `true` (ou o oposto)!

Nota: *Acredita em mim que* $A \simeq () \to A$

Vocês lembram da representação binária de um `I32`?

$$\begin{aligned} I32 &\simeq Bool \times Bool \times \cdots \times Bool \\ &\simeq (() + ()) \times \cdots \times (() + ()) \\ &\simeq () + () + () + \cdots + () + () \end{aligned}$$

Então dá para representar *muita coisa* usando só *Produto*, *Soma* e *Unity*.

Mas isso não quer dizer que a gente deva usar essas representações! O hardware sabe trabalhar muito bem com `I32`, então faz sentido ele ser um tipo primitivo.

Tipos Algébricos

Deveria estar em **Tipos - Caso Recursivo**, mas se encaixa melhor aqui!

Tipos Algébricos é uma forma para representar somas e produtos:

```
Bool = False | True
```

```
I32_1 = I32_1 Bool Bool Bool ... Bool
```

```
I32_2 = Int32Min | ... | MenosUm
      | Zero | Um | ... | Int32Max
```

Polimorfismo

- Do grego: *muitas formas*
- Em programação, geralmente é uma forma de permitir que várias coisas diferentes exerçam o mesmo papel.
- Exemplos:
  - Herança
  - Overloading
  - ...
- Isso é bem legal, pois evita que a gente escreva "duas coisas iguais".

Polimorfismo Paramétrico

Algumas funções/tipos usam algum tipo já existente mas não precisam saber nada sobre esse tipo.

Outros nomes:

- Generics [Java]
- Macros (*) [C]
- Meta Programming (Comptime) [Zig]
- Templates (typename) [C++]

Polimorfismo Paramétrico

Tipos que guardam/organizam "coisas" (estruturas de dados) geralmente usam muito bem esse tipo de polimorfismo.
Exemplos: Lista, Fila, Árvore, Mapa, ...

Notação para tipos paramétricos:

$$[] : \forall x \,.\, List\ x$$

Notação para essas funções paramétricos:

$$id : \forall x \,.\, x \to x$$

Nota: *Geralmente usam letra minúscula para variáveis de tipo*

Simplificando Destrutores

Esse polimorfismo simplifica alguns destrutores, onde tinha "família de destrutores" agora é só uma função genérica:

- $A \times B$

$$\forall x \,.\, A \times B \to (A \to B \to x) \to x$$

- $A + B$

$$\forall x \,.\, A + B \to (A \to x) \to (B \to x) \to x$$

- *Void*

$$absurd : \forall x \,.\, Void \to x$$

- *Bool*

$$\forall x \,.\, Bool \to x \to x \to x$$

Nota: *Dá para reconstruir* π_A *e* π_B *usando:*

$$K : \forall a \,.\, \forall b \,.\, a \to b \to a$$

$$KI : \forall a \,.\, \forall b \,.\, a \to b \to b$$

Maybe a

Outros nomes:

- Option
- Optional

```
data Maybe a
     = None
     | Some a
```

Destrutor

```
maybe :: Maybe a -> b -> (a -> b) -> b
maybe  None      b _ = b
maybe (Some a) _ f = f a
```

Either a b

Outros nomes:

- Result
- Error Union

```
data Either a b
    = Left a
    | Right b
```

Destrutor

```
either :: Either a b -> (a -> c)
    -> (b -> c) -> c
either (Left  a) f _ = f a
either (Right b) _ g = g b
```

List a

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

Destrutor

```
list :: List a -> b -> (a -> List a -> b)
    -> b
list  Nil         b _ = b
list (Cons a as) _ f = f a as
```

int *p = &i;

```
data Pointer a = Null
               | Estrela  a
               | Colchete [a]
```

Problemas (teóricos) dessa representação:

- `C` não sabe diferenciar `Estrela` e `Colchete`
- Temos que checar se é `Null` o tempo todo
- **Invisível:** no caso `Colchete`, podemos não saber o tamanho da lista
  Nota: *ainda podemos ter o "null terminator" ou guardar um tamanho externo*
- Na prática, só queremos usar `Estrela` ou `Colchete`
  Nota: *o "ou" é exclusivo*
- Mais alguns (específicos sobre implementação) . . .

Soluções: Zig

```
const OptionalInt = ?i32;

const PtrToOneInt = *i32;

const PtrToManyInt = [*]i32;

const NullablePtrToOneInt = ?*i32;

const PtrToOneOptionalInt = *?i32;

const NullablePtrToManyInt = ?[*]i32;

const PtrToManyOptionalInt = [*]?i32;
```

Outros: Zig

```
// struct { len : usize, data : [*]i32, };
const Slice = []i32;

// Tem valores no array ate achar um A
// A eh um valor de i32
const Sentinel = [*:A]i32;

const CharStar = [*:0]u8;
```

Implementação (Polimorfismo Paramétrico)

- Especialização/*inline*:
  - Gerar várias funções (*name mangling*)
  - Tudo se resolve estaticamente
  - Tende a gerar binários maiores (uma função para cada tipo)

- `void *` e *meta-dados*
  - Gerar só uma função
  - Tudo se resolve dinamicamente
  - Tende a gerar funções mais lentas (tem que olhar os meta-dados)

Exemplo: Especialização i32

```
struct<T> Pilha {
    usize len;
    usize size;
    [*]T items;
}

Pilha<T> new<T>() {
    return
        Pilha<T> {
    .len = 0,
    .size = 0,
    .items = malloc(
        0*sizeof(T)
        ), };
}
```

```
struct Pilha_i32 {
    usize len;
    usize size;
    [*]i32 items;
}

Pilha_i32 new_i32() {
    return
        Pilha_i32 {
    .len = 0,
    .size = 0,
    .items = malloc(
        0*sizeof(i32)
        ), };
}
```

Exemplo: Especialização i32

```
void push<T>(Pilha<T> *p, T item) {
    if ( p.size + 1 >= p.len ) {
        p.len *= 2;
        realloc(&p.items, p.len*sizeof(T));
    }
    p.items[p.size] = item;
    p.size += 1;
}

void push_i32(Pilha_i32 *p, i32 item) {
    if ( p.size + 1 >= p.len ) {
        p.len *= 2;
        realloc(&p.items, p.len*sizeof i32);
    }
    p.items[p.size] = item;
    p.size += 1;
}
```

Exemplo: Especialização i32

```
?T pop<T>(Pilha<T> *p) {
    if ( p.size > 0 ) {
        p.size -= 1;
        return p.items[p.size];
    } else return null;
}

?i32 pop_i32(Pilha_i32 *p) {
    if ( p.size > 0 ) {
        p.size -= 1;
        return p.items[p.size];
    } else return null;
}
```

Exemplo: Generalização

```
struct<T> Pilha {
    usize len;
    usize size;
    [*]T items;
}

Pilha<T> new<T>() {
    return Pilha<T> {
    .len = 0,
    .size = 0,
    .items = malloc(
        0*sizeof(T)
        ), };
}
```

```
struct Pilha {
    usize len, size;
    Info info;
    [*]*void items;
}

Pilha new(Info i) {
    return Pilha {
    .len =0, .size=0,
    .info = i,
    .items = malloc(
    0*sizeof(*void)
        ), };
}
```

Exemplo: Generalização

```
void push<T>(Pilha<T> *p, T item) {
    if ( p.size + 1 >= p.len ) {
        p.len *= 2;
        realloc(&p.items, p.len*sizeof(T));
    } p.items[p.size] = item;
    p.size += 1;
}

void push(Pilha *p, void *item) {
    if ( p.size + 1 >= p.len ) {
        p.len *= 2;
        realloc(&p.items,
            p.len*sizeof(*void));
    }
    p.items[p.size] = item;
    p.size += 1;
}
```

Exemplo: Generalização

```
?T pop<T>(Pilha<T> *p) {
    if ( p.size > 0 ) {
        p.size -= 1;
        return p.items[p.size];
    } else return null;
}
```

```
?*void pop(Pilha *p) {
    if ( p.size > 0 ) {
        p.size -= 1;
        return p.items[p.size];
    } else return null;
}
```

Exemplo: Generalização (coisas extras)

```
struct Info {
    usize size;
    u8 align;
    ...
};

struct InfoPack {
    Info info;
    *void ptr;
};

InfoPack from_i32_to_Info(i32 val) { ... }

i32 from_Info_to_i32(void *ptr) { ... }
```

Classes de Tipos

Outros nomes:

- Ad hoc Polymorphism
- Concepts [C++20]
- Interface [Go, Java, Kotlin, TypeScript]
- Module System [Standard ML]
- Mixin (?) [Ruby]
- Protocol [Erlang, Elixir, Swift]
- Typeclasses [Haskell]
- TypeTraits (?) [C++/Boost]
- Type Constrains [D]
- Traits [Rust]
- Vtable/Function Pointers [C, C++, Zig]

“Concepts vs Typeclasses vs Traits vs Protocols”
youtu.be/E-2y1qHQvTg

Classes de Tipos

- Agrupa tipos com uma mesma propriedade(s)
- Define um conjunto de operações com alguma semântica (que não dependem de implementação)
- É um contrato!
- Em Orientação a Objeto: uma forma de "injetar depêndencia" (*dependency injection*)
- Notação usa "seta gorda":

$$\forall x \,.\, \mathit{Classe}\ x \Rightarrow x \to x \to x$$

Monoid

```
class Semigroup a where
    (<>) :: a -> a -> a

class Semigroup a => Monoid a where
    mempty :: a -> a -> a
    mconcat :: [a] -> a
    mconcat = foldr (<>) mempty
```

- Associatividade (Semigroup):

```
(x <> y) <> z == x <> (y <> z)
```

- Identidades:

```
x <> mempty == x
mempty <> x == x
```

Functor

```
class Functor a where
    fmap :: (a -> b) -> f a -> f b
    (<$) :: a -> f b -> f a
    (<$) = fmap . const
```

- Identidade:

  ```
  fmap id == id
  ```

- Composição:

  ```
  fmap (f . g) == fmap f . fmap g
  ```

Collection (Java 8)

```
interface Collection<E> extends Iterable<E>
{   boolean add(E e);
    boolean addAll(Collection
        <? extends E> c);
    void clear();
    boolean contains(Object o);
    boolean containsAll(Collection<?> c);
    boolean equals(Object o);
    int hashCode();
    boolean isEmpty();
    Iterator<E> iterator();
    default Stream<E> parallelStream();
    boolean remove(Object o);
    ...
}
```

Collection (Java 8) - Part 2

```
interface Collection<E> extends Iterable<E>
{   ...
    boolean removeAll(Collection
        <? extends E> c);
    default boolean removeIf(Predicate
        <? super E> filter);
    boolean retainAll(Collection
        <? extends E> c);
    int size();
    default Spliterator<E> spliterator();
    default Stream<E> stream();
    Object[] toArray();
   <T> T[] toArray(T[] a);
}
```

docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Monad

Implementação (Classe de Tipos)

Outros Tipos

Existem mais tipos!

Mas infelizmente não dá tempo de falar sobre todos eles :(

Vou passar rapidinho por eles.

Tipos Dependentes

Permite criar *tipos* que dependem de *valores*

Útil para dar tipos para: `arrays`, *provas*, ...

Linguagens de programação:

- Agda
- C++ [Template]
- Idris
- Zig [Comptime, First Class Types]

Assistentes de Prova:

- ALF
- Coq
- Lean

Mais exemplos: en.wikipedia.org/wiki/Category:Dependently_typed_languages

Tipos Sub-estruturais

Como assim, você está usando um ovo duas vezes???
Você não pode fazer dois bolos e só usar um ovo!!!

Tipos Sub-estruturais

	Exchange	Weakening	Contraction
Ordered	Não	Não	Não
Linear	Sim	Não	Não
Affine	Sim	Sim	Não
Relevant	Sim	Não	Sim
Normal	Sim	Sim	Sim

Nota: *Existe uma lógica para cada linha da tabela.*

Usos:

- Ordered: Alocação em Stack
- Linear: Uso de recursos (exatamente uma vez)
- Affine: Uso de recursos (zero ou uma vez)
- Relevant: Uso de recursos (uma ou mais vezes)

Ref: `en.wikipedia.org/wiki/Substructural_type_system`

Tipos Sub-estruturais

	Exchange	Weakening	Contraction
Ordered	Não	Não	Não
Linear	Sim	Não	Não
Affine	Sim	Sim	Não
Relevant	Sim	Não	Sim
Normal	Sim	Sim	Sim

Nota: *Existe uma lógica para cada linha da tabela.*

Linguagens de programação (Linear/Affine):

- Idris
- Linear ML
- Rust [Borrow-checker, Copy Trait]
- Nim

Ref: en.wikipedia.org/wiki/Substructural_type_system

Efeitos Algébricos (*novo*)

São `exceptions` (`exceções`), porém mais *gerais* (lembram um pouquinho `interfaces`).

Os `handlers` (algo tipo os blocos `catch`) podem continuar a função que gerou a exceção e modificar a saída.

Linguagens de programação (de pesquisa):

- Eff (`eff-lang.org`)
- Effekt (`effekt-lang.org`)
- Koka (`koka-lang.github.io`)
- Unison (`unison-lang.org/`)

Ref: `en.wikipedia.org/wiki/Effect_system`

Row/Variant Polymorphism

A ideia é: se eu tenho uma função `f` que recebe uma *struct genérica*, mas só acessa o campo *.nome*; deveria ser possível passar as duas structs: `struct struct1 { nome : Str, num : Int }`, `struct struct2 { nome : Str, flag : Bool }`.

Isso lembra *duck typing ("If it walks like a duck, and it quacks like a duck, then it must be a duck!")*

Nota: *Row Polymorphism e Variant Polymorphism são duas coisas diferentes, mas duais.*

Linguagens de programação:

- Roc (em desenvolvimento: `roc-lang.org/`)

Mais Tipos!

Tipos “que existem” mas “que eu não sei muito bem o que são”:

- Variant/Contravariant/Invariant Types
- Subtypes/Supertypes
- ...

FIM

Fim!