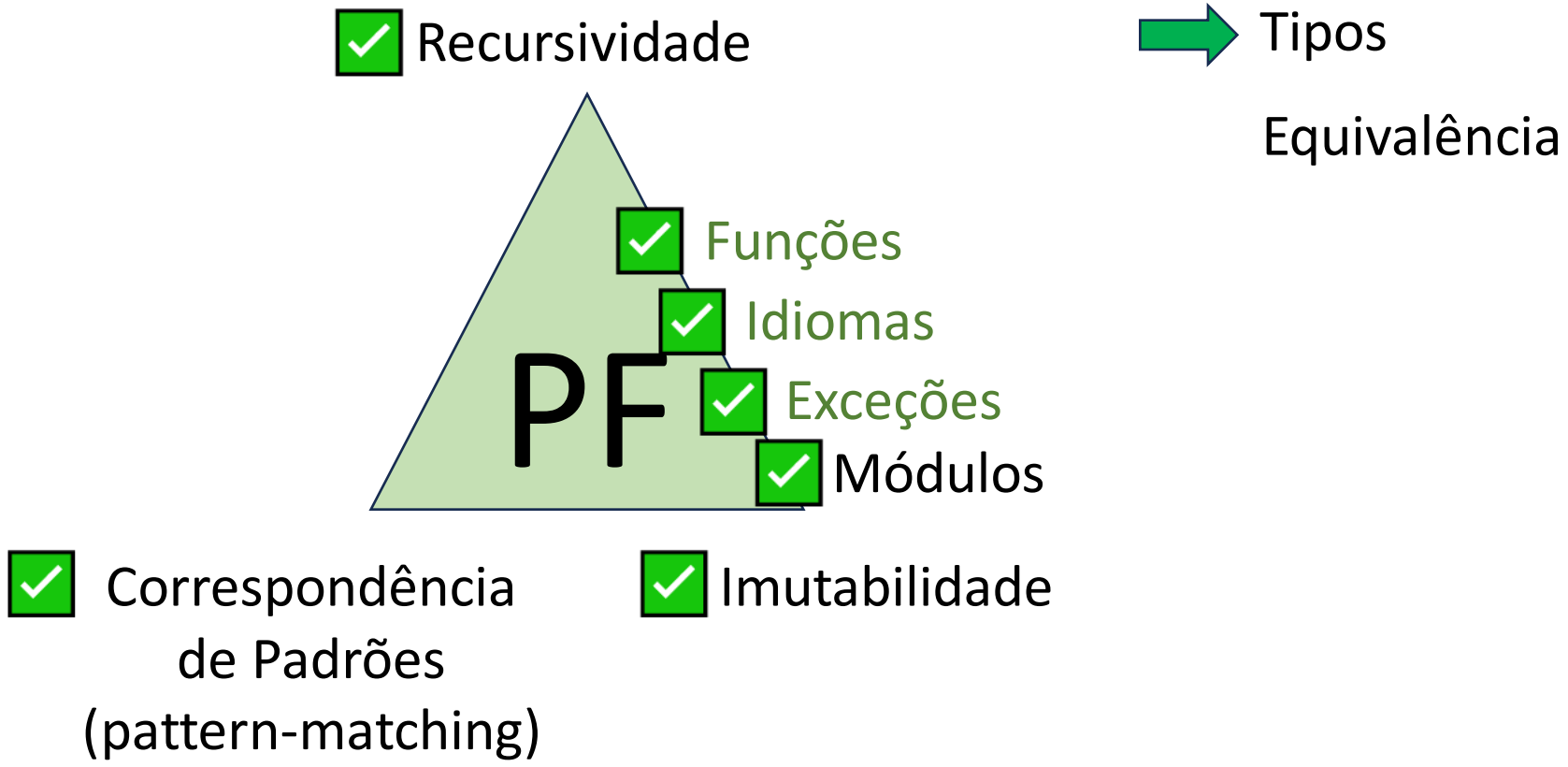
# Aula 9: Inferência de Tipos e Equivalência em OCaml

UC: Programação Funcional

2023-2024

# Até agora vimos



# Verificação de tipo (type-checking)

- A **verificação** (estática) **de tipos** pode rejeitar um programa antes de ser executado para evitar a possibilidade de alguns erros
  - Uma caraterística das **linguagens estaticamente tipificadas**
- As **linguagens tipificadas dinamicamente** não fazem esse controlo
  - Por isso, podem tentar tratar um número como uma função em tempo de execução
- Estudaremos as vantagens relativas entre as linguagens
  - Python, Javascript são dinamicamente tipificadas
- OCaml (Scala, Haskell) são estaticamente tipificadas (mas Java, C#, C, C++ também)
  - Cada ligação/binding tem um tipo, determinado antes da execução do programa

# Implicitamente tipificada

- **OCaml é estaticamente tipificada**
- **OCaml é implicitamente tipificada**: raramente é necessário escrever tipos
  - Continua a ser de tipagem estática: “mas mais parecido com Java do que com Javascript!”

```
let f x = (* infer val f : int -> int *)
  if x > 3
  then 42
  else x * 2

let g x = (* report type error *)
  if x > 3
  then true
  else x * 2
```

# Inferência de tipos

- **Problema da inferência de tipos**: Atribuir a cada ligação/expressão um tipo tal que a verificação de tipo seja bem sucedida
  - Falha se e só se não existir solução
- Em princípio, poderia ser uma passagem antes do verificador de tipos (type-checker)
  - Na prática, muitas vezes implementados em conjunto
- A inferência de tipos pode ser fácil, difícil ou *impossível*
  - Depende dos pormenores da linguagem
  - Subtil, elegante e *não é mágico*: OCaml

# Visão geral

- Descreveremos o algoritmo de inferência de tipos OCaml através de vários exemplos
  - O algoritmo geral é um tópico um pouco mais avançado
  - O suporte de funções aninhadas também é um pouco mais avançado / “tricky”

- O suficiente para vos ajudar a “fazer a inferência de tipo mentalmente”
  - E compreenderem que não se trata de magia

# Passos Fundamentais

- Determinar os tipos das ligações / “bindings” por ordem
  - Não podemos utilizar ligações posteriormente: impossibilita a verificação de tipo
- Pata cada ligação **`let`** :
  - Analisar a definição para criar um conjunto de *restrições*
  - Exemplo: Se ver **`x > 0`**, então **`x`** deve ser do tipo **`int`**
  - Exemplo: Se observamos **`if true then y else z`**, então **`y`** e **`z`** devem ter o mesmo tipo
  - Erro de tipo se não for possível manter todos os factos (com restrições excessivas)
- Em seguida, utilizamos variáveis de tipo (e.g., **`'a`**) para quaisquer tipos sem restrições
  - Exemplo: Um argumento não utilizado pode ter qualquer tipo
- (Finalmente, aplicar a restrição de valor, falaremos mais tarde)

# Primeiro exemplo

Depois deste exemplo, veremos passo-a-passo

- Tal como o algoritmo automatizado faz

```
let x = 42 (* val x : int *)

let f y z w =
  if y (* y deve ser bool *)
  then z+x (* z deve ser int *)
  else 0 (* ambos os ramos têm o mesmo tipo *)
(* f deve devolver um int
   f deve receber um bool * int * ANYTHING
   ou seja, val f : bool -> int -> 'a -> int
 *)
```

# A relação com o polimorfismo

- Característica central da inferência de tipos em OCaml: pode inferir tipos com variáveis de tipo
  - Ótimo para reutilização de código e compreensão de funções

- Mas lembrem-se que estes são conceitos ortogonais
  - As linguagens podem ter inferência de tipo sem variáveis de tipo
  - As linguagens podem ter variáveis de tipo sem inferência de tipo

# Ideia-chave

- Recolher todas as restrições necessárias para a verificação de tipos (type-checking)
- Resolver restrições ou erro de tipo se não for possível resolver
- Exemplos (Ver ficheiro aula9.ml):
  - Dois exemplos sem variáveis de tipo
  - E um exemplo que não faz a verificação de tipo
  - Exemplos de funções polimórficas
    - Nada muda, apenas a restrição é menor: alguns tipos podem "ser qualquer coisa", mas podem ainda precisar de ser iguais a outros tipos

# Contexto Histórico

1. A história da inferência de tipo até agora é demasiado branda
   - As restrições de valores  limitam onde podem ocorrer tipos polimórficos
   - Vamos ver porquê e o que fazer

2. OCaml está num “ponto ideal”
   - A inferência de tipos é mais difícil sem polimorfismo
   - A inferência de tipos é mais difícil com subtipagem

Importante para “terminar a história”, mas estes dois tópicos são:

- Um pouco mais avançados
- Um pouco menos elegantes

# O Problema

Tal como descrito até agora, o sistema de tipos não é consistente!

- Permite colocar um valor do tipo **`t1`** (e.g., **`int`**) onde se espera um valor do tipo **`t2`** (e.g., **`string`**)

A "culpa" é de uma combinação do polimorfismo e da mutação:

```
let r = ref None (* val r : 'a option ref *)
let _ = r := Some "hi"
let i = (Option.get (!r)) / 3
```

- A atribuição tipifica porque (infixo) **`:=`** tem tipo **`'a ref -> 'a -> unit`**, e instância com **`string option`**
- A desreferência tipifica porque **`!`** Tem tipo **`'a ref -> 'a`**, e instância com **`int option`**

# O que fazer

Para restaurar a consitência, é necessário um sistema de tipos mais rigoroso que rejeite pelo menos uma destas três linhas

```
let r = ref None (* val r : 'a option ref *)
let _ = r := Some "hi"
let i = (Option.get (!r)) / 3
```

- E não pode estabelecer regras especiais para tipos de referência porque o verificador de tipos não pode conhecer a definição de todos os sinónimos de tipos
  - Devido ao sistema de módulos

```
type 'a foo = 'a ref
let f : 'a -> 'a foo = ref
let r = f None
```

# A solução

```
let r = ref None (* val r : '_weak4 option ref *)
let _ = r := Some "hi"
let i = Option.get (!r) / 3
```

- Restrição de valores: uma var-ligação só pode ter um tipo polimórfico se a expressão for uma variável ou um valor
  - Chamadas de função com **`ref None`** não são
  - Nesse caso, o OCaml utiliza um “tipo fraco” para garantir que é preenchido com, no máximo, um tipo monomórfico baseado numa utilização posterior
- Não é óbvio que isto seja suficiente para tornar o sistema de tipos consistente, mas é

# A desvantagem

Como vimos, a restrição de valor pode causar problemas quando é desnecessária porque não estamos a utilizar a mutação

```
let pairWithOne = List.map (fun x -> (x,1))
(* does not get type 'a list -> ('a*int) list *)
```

O verificador de tipos não sabe que **`List.map`** não está a fazer uma referência mutável

Vimos uma solução alternativa sobre a aplicação parcial:

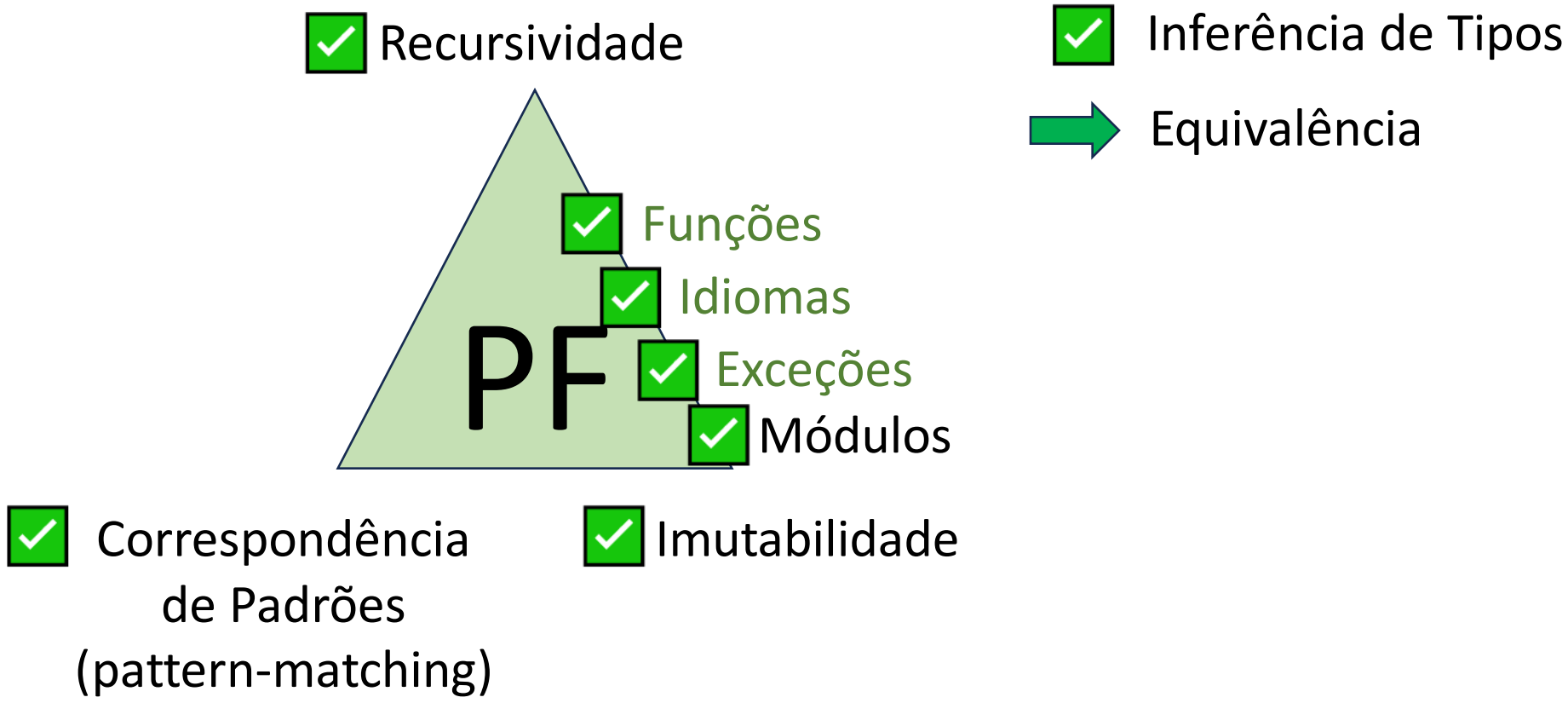
```
let pairWithOne xs = List.map (fun x -> (x,1)) xs
(* 'a list -> ('a*int) list *)
```

Se produzíssemos uma referência mutável, teríamos uma nova referência a cada passo, o que não é mau!

# O ótimo local

- Apesar da restrição de valor, a inferência de tipos em OCaml é elegante e bastante fácil de entender
- Mais difícil *sem* o polimorfismo
  - Que tipo deve ter o comprimento-da-lista?
- Mais difícil *com* subtipagem
  - Suponhamos que os pares são “supertipos” de tuplos mais largos
  - Então `let (y,z) = x` restringe `x` a ter pelo menos dois campos, mas não exatamente dois campos
  - Dependendo dos pormenores, as linguagens podem suportar esta situação, mas os tipos são frequentemente mais difíceis de inferir e compreender

# Até agora vimos



# Equivalência

Um olhar mais cuidadoso sobre o significado de “dois blocos de código são equivalentes”

- Ideia fundamental da engenharia de software
- Mais fácil com
  - Abstração (“esconder coisas”)
  - Menos efeitos secundários

Não se trata de “novas formas de codificar algo”

# Equivalência

É preciso pensar “será que estes blocos de código são equivalentes” frequentemente

- Quanto mais precisamente se pensar nisso, melhor

- *Manutenção do código:* Posso simplificar este código?
- *Compatibilidade com versões anteriores:* Posso adicionar novas funcionalidades sem alterar o funcionamento das funcionalidades antigas?
- *Otimização:* Posso tornar este código mais rápido?
- *Abstração: Um utilizador externo pode saber que eu fiz uma determinada alteração?*

# Mais especificamente

Para centrar a discussão: Quando é que podemos dizer que duas funções são equivalentes, mesmo sem olhar para todas as chamadas para elas?

- Pode não conhecer todas as chamadas
- Por exemplo, editar uma biblioteca sem conhecer todos os utilizadores dela

# Definição

Duas funções são equivalentes se tiverem o mesmo “**comportamento observável**”, independentemente da forma como são utilizadas em qualquer parte de um programa

Com argumentos equivalentes, elas:

- Produzem resultados equivalentes
- Têm o mesmo comportamento de (não-)terminação
- Mutam a memória (não-local) da mesma forma
- Fazem a mesma entrada/saída
- Levantam as mesmas exceções

Não observável: tempo utilizado, espaço utilizado, memória local

# Expressividade negativa

Reparem que se *soubermos* que quem chama não pode "fazer" ou "ver" certas coisas, então há mais coisas equivalentes

- Facilita a manutenção
- Facilita a compatibilidade com as versões anteriores
- Facilita a otimização

Formas de garantir que quem chama não pode fazer ou ver coisas:

- Restringir o conjunto de argumentos possíveis da função
  - E.g., com um sistema de tipos e/ou abstração
- Utilizem dados imutáveis
- Evitem a entrada/saída
- Evitem exceções

# Exemplo

Como a pesquisa de variáveis em OCaml não tem efeitos colaterais, essas duas funções são equivalentes:

```
let f x = x + x
```

$\equiv$

```
let y = 2
let f x = y * x
```

Regra geral, estes dois não são equivalentes: dependem do que é passado em **f**

- São equivalente *se* o argumento **f** não tiver "side-effects"

```
let g f x =
  (f x) + (f x)
```

$\not\equiv$

```
let y = 2
let g f x = y * (f x)
```

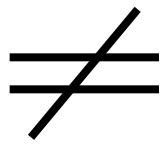
- Exemplo: `g (fun i -> print_string "hi" ; i) 7`
- Uma óptima razão para efetuarmos programação funcional "*pura*"

# Outro exemplo

Estes blocos são equivalentes *apenas se* as funções ligadas a **g** e **h** não levantarem exceções ou tiverem "side-effects" (imprimirem, atualizarem o estado, etc.)

- Mais uma vez: as funções puras tornam mais coisas equivalentes

```
let f x =
  let y = g x in
  let z = h x in
  (y,z)
```



```
let f x =
  let z = h x in
  let y = g x in
  (y,z)
```

- Exemplo: **g** divide por **0** e **h** muta uma referência (top-level)
- Exemplo: **g** escreve numa referência que **h** vai ler

# Outro exemplo que importa

Mais uma vez, transformar da “esquerda para a direita” é ótimo, mas só se as funções forem puras:

```
map f (map g xs)
```

```
map (f % g) xs
```

# Açúcar sintático

Usar ou não usar, o açúcar sintático é sempre equivalente

- Por definição, senão não é açúcar sintático

Exemplo:

```
let f x =
  x && g x
```

$=$

```
let f x =
  if x
  then g x
  else false
```

Mas tenham cuidado com a ordem de avaliação

```
let f x =
  x && g x
```

$\neq$

```
let f x =
  if g x
  then x
  else false
```

# Três equivalências “standard”

Existem três equivalências padrão para funções

- Bem estudadas
- “Devem funcionar” em qualquer linguagem [decente?]
- Mas têm alguma subtileza quando são aplicáveis

Vejamos cada uma delas...

# Equivalência (1)

Renomear consistentemente variáveis ligadas e suas utilizações

```
let y = 14
let f x = x+y+x
```

$=$

```
let y = 14
let f z = z+y+z
```

Mas notem que não podem utilizar um nome de variável já utilizado no corpo da função para se referir a outra coisa

```
let y = 14
let f x = x+y+x
```

$\neq$

```
let y = 14
let f y = y+y+y
```

```
let f x =
  let y = 3 in
   x+y
```

$\neq$

```
let f y =
  let y = 3 in
   y+y
```

# Equivalência (2)

Utilizar uma função auxiliar ou não

```
let y = 14
let g z = (z+y+z)+z
```

$\equiv$

```
let y = 14
let f x = x+y+x
let g z = (f z)+z
```

Mas é preciso ter cuidado com os ambientes

```
let y = 14
let y = 7
let g z = (z+y+z)+z
```

$\not\equiv$

```
let y = 14
let f x = x+y+x
let y = 7
let g z = (f z)+z
```

# Equivalência (3)

Encapsulamento de funções desnecessário

```
let f x = x+x
let g y = f y
```

$\equiv$

```
let f x = x+x
let g = f
```

Mas reparem que se computarem a função a chamar e *essa computação* tiver "sife-effects", têm de ter cuidado

```
let f x = x+x
let h () =
  (print_string "hi"; f)
let g y = (h()) y
```

$\not\equiv$

```
let f x = x+x
let h () =
  (print_string "hi"; f)
let g = (h())
```

# Ainda uma outra

Se ignorarmos os tipos, então os `let` do OCaml podem ser um açúcar sintático para chamar uma função anónima:

```
let x = e1   in e2
```

```
(fun x -> e2) e1
```

- Ambos avaliam **`e1`** para **`v1`**, então avalia **`e2`** num ambiente estendido com o mapeamento de **`x`** para **`v1`**
- Portanto, *exatamente* a mesma avaliação de expressões e resultado

Mas em OCaml, há uma diferença no sistema de tipos:

- **`x`** à esquerda pode ter um tipo polimórfico, mas não à direita
- Pode sempre ir da direita para a esquerda
- Se **`x`** não precisar de ser polimórfico, pode ir da esquerda para a direita

# E quanto ao desempenho

De acordo com a nossa definição de equivalência, estas duas funções são equivalentes, mas aprendemos que uma é horrível

- Na verdade, já olhamos para estes dois blocos, lembram-se?

```
let rec max xs =
  match xs with
  | [] -> raise Empty
  | x::[] -> x
  | x::xs' ->
      if x > max xs'
      then x
      else max xs'
```

```
let rec max xs =
  match xs with
  | [] -> raise Empty
  | x::[] -> x
  | x::xs' ->
      let y = max xs' in
      if x > y
      then x
      else y
```

# Moral da história

A equivalência é um conceito essencial no desenvolvimento de software

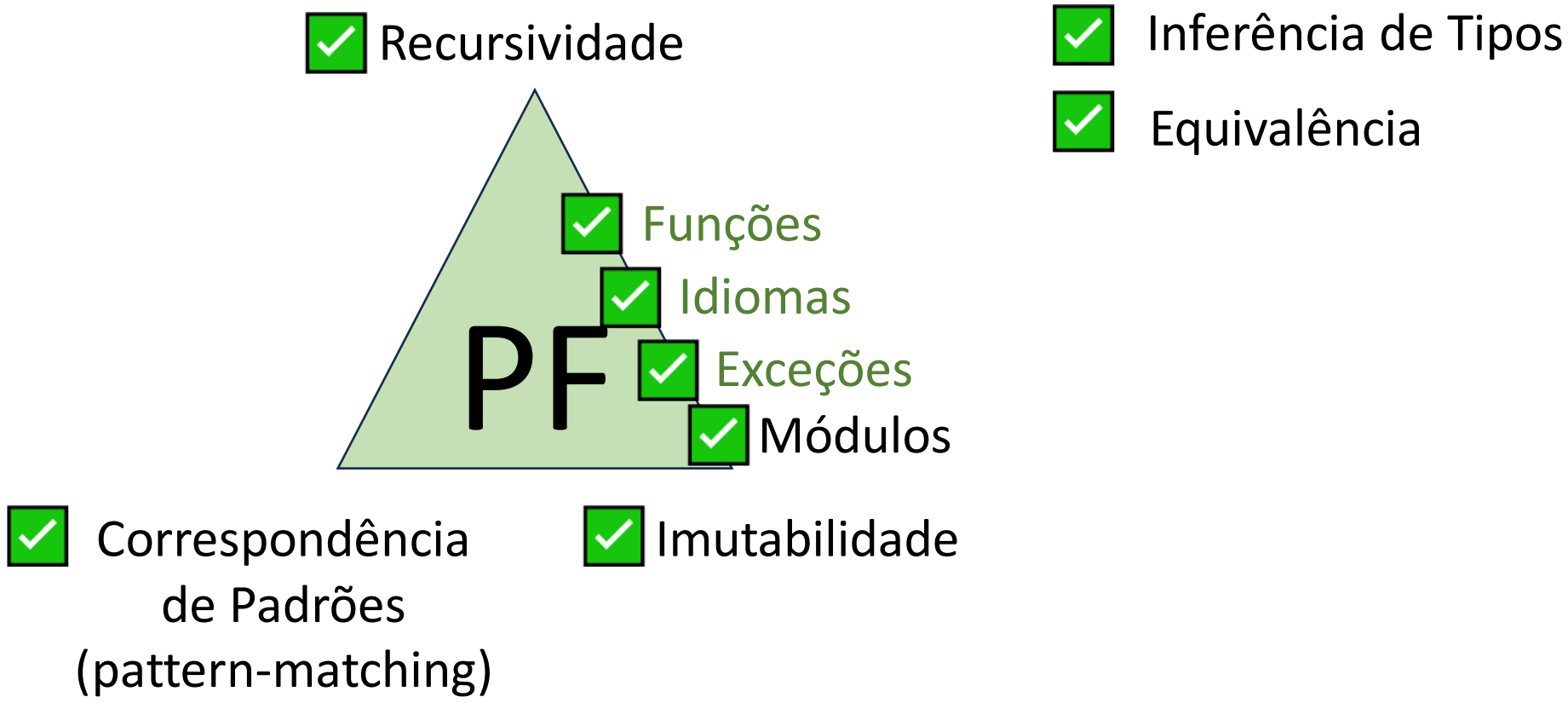
A equivalência é subtil, muitas vezes depende de:

- Possíveis efeitos secundários “side-effects”
- Âmbito de aplicação variável

A equivalência é subtil, depende sempre:

- O que se assume que os utilizadores podem “observar”

# Até agora vimos



Créditos para Dan Grossman.