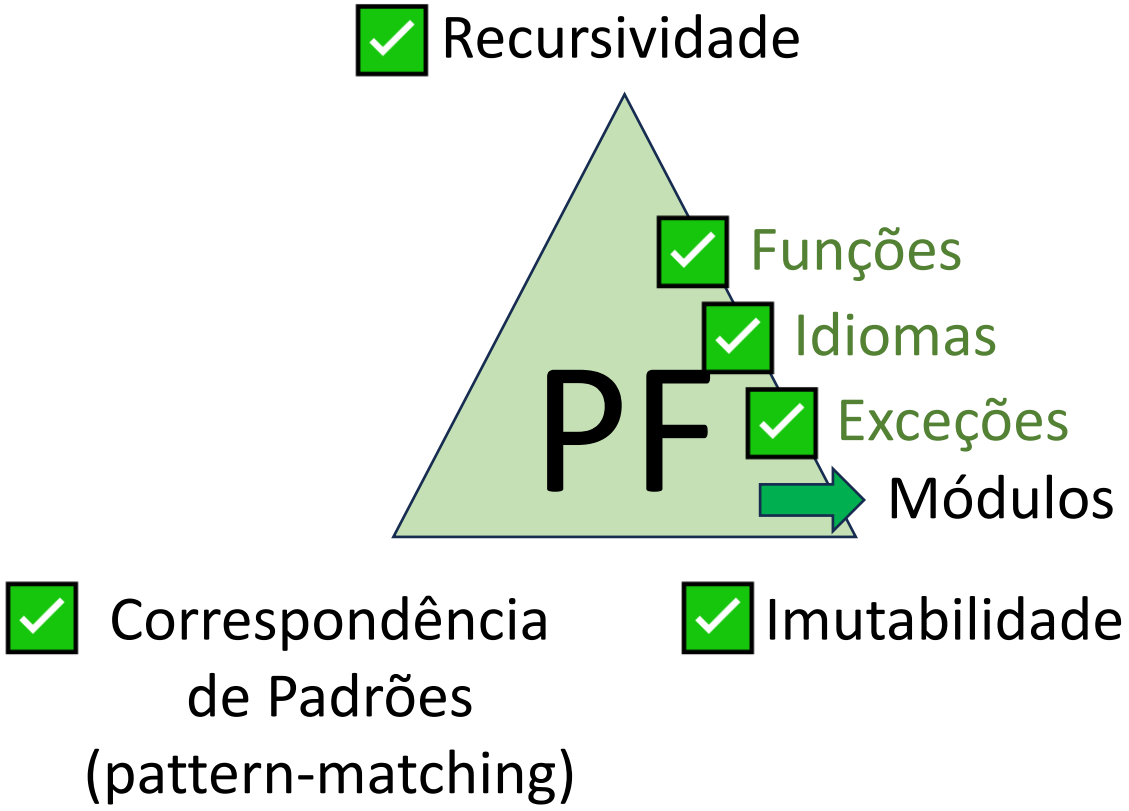
Aula 8: Módulos em OCaml

UC: Programação Funcional

2023-2024

Até agora vimos



Módulos

Para programas maiores, uma sequência de “bindings top-level” é insuficiente

- Um “binding” pode *utilizar todas* os “bindings” anteriores que não tenham sido sobrepostos

Assim, em OCaml temos *estruturas* para definir *módulos*

```
module MyModule = struct bindings end
```

- Dentro de um módulo, podemos utilizar as ligações anteriores como habitualmente
  - Pode ter qualquer tipo de ligação (let, type, exception, ...)
- Fora de um módulo, consulte as ligações de módulos anteriores através de **`ModuleName.bindingName`**
  - Tal como **`List.fold_left`**, etc. agora podemos definir os nossos próprios módulos

Exemplo

```
module MyMathLib = struct
  let rec fact x =
    if x = 0
    then 1
    else x * fact (x - 1)
  let half_pi = Float.pi /. 2.0
  let doubler x = x *. 2
end
```

Módulos implícitos através de ficheiros

Em OCaml temos uma conveniência incorporada:

- Um ficheiro `foo.ml` tem um módulo

  ```
  module Foo = struct … end
  ```

  implícito "à sua volta"

- Assim, os diferentes ficheiros não escondem os "bindings" uns dos outros, mas precisam de aceder a ligações de outros ficheiros com a sintaxe **`Foo.bar`**

Abertura de Módulos: Open

Podemos usar **`open ModuleName`** para ter acesso “direto” aos “bindings” do módulo

- Nunca é necessário; é apenas uma conveniência; muitas vezes é um mau estilo
- Muitas vezes é melhor criar “val-bindings” locais apenas para as ligações que utilizamos muito, por exemplo, **`let map = List.map`**
  - Mas não funciona com padrões
  - E o **`open`** pode ser útil, e.g., para testar código

Gestão do Namespace

Até agora, isto é apenas gestão de “namespaces”

Atribuímos uma hierarquia aos nomes para evitar sobrepô-los

- Sim, podem ter módulos dentro de módulos
- Permite ter diferentes módulos que reutilizem nomes, e.g., **`map`**
- Muito importante, mas não muito interessante

Tipos Módulo (assinaturas)

- Uma *assinatura* é um tipo para um módulo
  - Indica-nos que ligações tem e quais são os seus tipos
- Pode definir uma assinatura e *atribuí-la* aos módulos

```
module type MATHLIB = sig
  val fact : int -> int
  val half_pi : float
  val doubler : int -> int
end
module MyMathLib : MATHLIB = struct
  let rec fact x = ...
  let half_pi = Float.pi /. 2.0
  let doubler x = x *. 2
end
```

Em geral

- Assinaturas
  - Podem incluir variáveis, tipos, e exceções definidas no módulo

```
module type NomeDaAssinatura = sig
  tipos-para-as-ligações
end
```

- Atribuímos uma assinatura a um módulo

```
module NomeDoModulo : NomeDaAssinatura = struct
  ligações
end
```

  - O módulo *não efetua a verificação de tipo* a menos que *corresponda* à assinatura, o que significa que tem todas as ligações nos tipos corretos

Escondendo “coisas”

O verdadeiro valor das assinaturas é *esconder* ligações e definições de tipos

- Até agora, apenas serve para documentar e verificar os tipos

Ocultar os pormenores de implementação é a estratégia mais importante para escrever software correto, robusto e reutilizável

Assim, primeiro lembremo-nos que as funções já funcionam bastante bem para algumas formas de ocultação...

Escondendo “coisas” com funções

Estas três funções são totalmente equivalentes: nenhum utilizador pode saber qual delas estamos a utilizar (pelo que podemos alterar a nossa escolha mais tarde).

```
let double x = x*2
let double x = x+x
let y = 2
let double x = x*y
```

A definição de funções auxiliares localmente também é poderosa

- Pode alterar/remover funções mais tarde e saber que isso não afeta outro código

Também seria bom ter funções “privadas” de top-level

- Assim, duas funções podem partilhar uma função auxiliar
- O OCaml faz isso através de assinaturas que omitem as ligações...

Exemplo

Fora do módulo, **`MyMathLib.doubler`** está simplesmente desligada

- Por isso, não podemos utiliza-la [diretamente]
- Uma ideia bastante poderosa e muito simples

```
module type MATHLIB = sig
  val fact : int -> int
  val half_pi : float
end
module MyMathLib : MYMATHLIB = struct
  let rec fact x = ...
  let half_pi = Float.pi /. 2.0
  let doubler x = x *. 2
end
```

Na prática

- Relembrem-se de que **`foo.ml`** implicitamente define o módulo **`Foo`** sem nenhum **`module Foo = struct … end`** explicito
- Similarmente, se **`foo.mli`** existir, ele descreve a assinatura para **`Foo`** sem qualquer modulo explicito **`module type NomeTipo= sig … end`**
  - **`foo.ml`** é verificada da mesma forma que o módulo **`module Foo : NomeTipo = …`**
  - Outros módulos tipados com **`foo.mli`** “escondem coisas”
- Se não existir nenhum **`foo.mli`** então nada é “escondido”

Na prática (simplificado)

- **`foo.ml`** define o modulo **`Foo`**
- **`Bar`** utiliza a variável **`x`**, tipo **`t`**, construtor **`C`** em **`Foo`** através de **`Foo.x`**, **`Foo.t`**, **`Foo.C`**
  - Pode-se abrir um módulo utilizando **`open`**, no entanto devemos utilizar com moderação
- **`foo.mli`** define a assinatura para o módulo **`Foo`**
  - Ou “tudo é publico” se não existir **`foo.mli`**
- A ordem importa (depende da ordem na linha de comandos)
  - Sem referências futuras (longa história)
  - Ordem de avaliação do programa
- Ver manual de referência para ficheiros **`.cm[i,o]`**, flag **`-c`**, etc.

Exemplo 2: módulos e assinaturas

foo.ml

```
type t1 = X1 of int
        | X2 of int

let get_int t =
  match t with
    X1 i -> i
  | X2 i -> i

type even = int

let makeEven i = i*2
let isEven1 i = true
(* isEven2 é "privado" *)
let isEven2 i = (i mod 2)=0
```

foo.mli

```
(* escolhemos mostrar *)
type t1 = X1 of int
        | X2 of int

val get_int : t1->int

(* escolhemos esconder *)
type even

val makeEven : int->even
val isEven1 : even->bool
```

Exemplo 3: módulos e assinaturas

bar.ml

```
type t1 = X1 of int
        | X2 of int

let conv1 t =
  match t with
    X1 i -> Foo.X1 i
  | X2 i -> Foo.X2 i
let conv2 t =
  match t with
    Foo.X1 i -> X1 i
  | Foo.X2 i -> X2 i

let _ =
 Foo.get_int(conv1(X1 17));
 Foo.isEven1(Foo.makeEven 17)
 (* Foo.isEven1 34 *)
```

foo.mli

```
(* escolhemos mostrar *)
type t1 = X1 of int
        | X2 of int

val get_int : t1->int

(* escolhemos esconder *)
type even

val makeEven : int->even
val isEven1 : even->bool
```

Outro exemplo

Consideremos agora um módulo que define um tipo de dados abstrato (ADT)

- Essencialmente um tipo de dados com operações sobre ele

Este exemplo de números racionais contém **`add`** e **`string_of_rational`**

```
module Rational1 =  struct
  type rational = Whole of int | Frac of int*int
  exception BadFrac

  (* outras funções internas, e.g., gdc *)

  let rec make_frac (n, d) = …
  let rec add r1 r2 = …
  let string_of_rational r = …
end
```

Especificação da biblioteca e invariantes

Propriedades [garantias visíveis do exterior, até ao programador da biblioteca]

- Não permitir denominadores de 0
- Devolver strings na forma reduzida (“4” e não “4/1”, “3/2” e não “9/6”)
- Cálculos corretos
- Sem ciclos infinitos ou exceções

Invariantes [parte da implementação, não da especificação do módulo]

- Todos os denominadores são maiores que 0
- Um valor **`racional`** devolvido por funções é sempre reduzido

Mais invariantes

O nosso código *garante* os invariantes e *confia* neles

Garantias:

- **`make_frac`** não permite denominador 0, remove denominador negativo e reduz o resultado
- **`add`** chama **`reduce`** depois de multiplicar os denominadores

Confianças:

- **`gcd`** não funciona com argumentos negativos, mas nenhum denominador pode ser negativo
- **`add`** utiliza as propriedades do cálculo para evitar chamar **`reduce`**
- **`string_of_rational`** assume que os seus argumentos estão reduzidos

Primeira assinatura

Com o que sabemos até agora, esta assinatura faz sentido:

- **`gcd`** e **`reduce`** não são visíveis fora do módulo

```
module type RATIONAL_A = sig
  type rational = Whole of int | Frac of int * int
  exception BadFrac
  val make_frac : int * int -> rational
  val add : rational -> rational -> rational
  val string_of_rational : rational -> string
end

module Rational1 : RATIONAL_A = struct ...
```

O problema

Ao revelar a definição do tipo, permitimos que os utilizadores violem os nossos invariantes criando diretamente valores do tipo **`Rational1.rational`**

- Quanto muito um comentário "devemos usar **`Rational1.make_frac`**"

```
module type RATIONAL_A = sig
  type rational = Whole of int | Frac of int * int
  …
```

Qualquer uma das opções pode levar-nos a ciclos infinitos ou resultados errados, razão pela qual o código do módulo nunca os devolveria:

- **`Rational1.Frac (3,1)`**
- **`Rational1.Frac (1,0)`**
- **`Rational1.Frac (3,-2)`**
- **`Rational1.Frac (9,6)`**

Por isso, tentem esconder mais

Ideia: Um ADT deve ocultar a definição do tipo concreto para que os utilizadores não possam criar diretamente valores do tipo que invalidem o invariante

Infelizmente, esta tentativa não funciona porque a assinatura utiliza agora um tipo **`rational`** que não se sabe se existe:

```
module type RATIONAL_WRONG = sig
  exception BadFrac
  val make_frac : int * int -> rational
  val add : rational -> rational -> rational
  val string_of_rational : rational -> string
end

module Rational1 : RATIONAL_WRONG = struct ...
```

Tipos abstratos

Assim, em OCaml temos a funcionalidade exata para isto (uma situação comum!!):

Na assinatura: **`type foo`** significa que o tipo existe, mas que o utilizador não sabe a sua definição

```
module type RATIONAL_B = sig
  type rational
  exception BadFrac
  val make_frac : int * int -> rational
  val add : rational -> rational -> rational
  val string_of_rational : rational -> string
end

module Rational1 : RATIONAL_B = struct ...
```

E funciona!

Não há nada que um utilizador possa fazer para invalidar invariantes e propriedades:

- A única maneira de construir um racional é usar **`Rational1.make_frac`**
- Depois podemos apenas usar **`Rational1.make_frac`**, **`Rational1.add`**, e **`Rational1.string_of_rational`**
- Escondemos os construtores e os padrões – não sabemos se um **`Rational1.rational`** é um tipo variante (poderá não ser!)
- Mas os utilizadores podem continuar a passar as frações de qualquer forma

Devemos utilizar o sistema de módulos para aplicar abstrações

Restrições chave

Assim, temos duas formas poderosas de utilizar assinaturas para esconder detalhes:

1. Negar a existência de ligações (variáveis, funções, ...)

2. Tornar os tipos abstratos (para que os utilizadores não possam criar diretamente valores dos mesmos nem aceder diretamente às suas partes)

(Mais tarde veremos que uma assinatura também pode tornar o tipo de uma ligação mais específico do que é dentro do módulo, o que é interessante mas menos importante)

E quanto ao construtor `whole`

Deve raciocinar cuidadosamente sobre cada ligação pública para garantir que não quebra nenhuma abstração

A função **`whole`** acaba por estar bem

De facto, não deve haver problema se o construtor **`Whole`** for exposto, uma vez que o único problema é o **`Frac`**

- Em OCaml isso não é permitido; o construtor **`Whole`** não é tratado como uma função, por isso definimos um *wrapper* **`whole`**
- Outras linguagens da família ML, SML, permitem expor construtores como funções

Correspondência da assinatura

Até agora, temo-nos baseado numa noção informal de “um módulo verifica o tipo dado por uma assinatura?” Como de costume, existem regras precisas...

`module Foo : BAR` é permitido se:

- Todos os tipos não abstratos de **`BAR`** são providenciados em **`Foo`**, como especificado
- Todos os tipos abstratos de **`BAR`** são providenciados em **`Foo`**
  - Pode ser um tipo variante ou um tipo sinonimo
- Todas as val-ligações de **`BAR`** são providenciadas em **`Foo`**, possivelmente com um tipo interno mais geral e/ou menos abstrato
- Todas as exceções em **`BAR`** são providencias em **`Foo`**

Notem que **`Foo`** pode ter mais ligações/ “bindings” (*implícitas* nas regras anteriores)

Implementações equivalentes

O propósito chave da abstração é permitir *diferentes implementações* serem *equivalentes*

- Nenhum utilizador pode dizer qual está a utilizar
- Possibilita melhorar/substituir/escolher implementações mais tarde
- Mais fácil de fazer se *começarmos* com assinaturas mais abstratas (revelamos apenas o que é necessário)

Agora: uma segunda estrutura que também pode ter assinatura **`RATIONAL_A`**, **`RATIONAL_B`**, ou **`RATIONAL_C`**

- Mas apenas *equivalente* entre o **`RATIONAL_B`** ou o **`RATIONAL_C`** (ignorando a possibilidade de “overflow”)

Implementações equivalentes

Exemplo (ver código `aula8.ml`):

- **`structure Rational2`** não mantém os racionais na forma reduzida, reduzindo-os "no último momento" em **`string_of_rational`**
  - Também utiliza as funções locais **`gcd`** e **`reduce`**
- Não é equivalente ao **`RATIONAL_A`**
  - **`Rational1.string_of_rational(Rational1.Frac(9,6)) = "9/6"`**
  - **`Rational2.string_of_rational(Rational2.Frac(9,6)) = "3/2"`**
- Equivalente ao **`RATIONAL_B`** ou **`RATIONAL_C`**
  - Invariantes diferentes, mas as mesmas propriedades
  - Essencial que o tipo **`rational`** seja abstrato

Um exemplo mais interessante

Dada uma assinatura com um tipo abstrato, podem ser criadas diferentes estruturas:

- Ter a mesma assinatura
- Mas implementar o tipo abstrato de forma diferente

Estas estruturas podem ou não ser equivalentes

Exemplo (ver código aula8.ml):

- **`type rational = int * int`**
- *Não tem* a assinatura **`RATIONAL_A`**
- *Equivalente* a ambas as estruturas anteriores tanto a **`RATIONAL_B`** como a **`RATIONAL_C`**

Alguns pormenores interessantes para o `Rational3`

- Internamente **`make_frac`** tem tipo **`int * int -> int * int`**, mas externamente **`int * int -> rational`**
  - O utilizador não pode dizer se devolve-mos um argumento inalterado
  - Poderá dar o tipo **`rational -> rational`** na assinatura, mas isto é horrível: torna todo o módulo *inutilizável* - porquê?
- Internamente **`whole`** tem tipo **`'a -> 'a * int`** mas externamente **`int -> rational`**
  - Corresponde porque podemos especializar **`'a`** para **`int`** e então abstrair **`int * int`** para **`rational`**
  - **`whole`** não pode ter tipos **`'a -> int * int`** or **`'a -> rational`** (deve especializar todos os usos de **`'a`**)
  - O verificador de tipo (type-checker) infere tudo isto por nós

Não é possível misturar-e-combinar ligações de módulos

Módulos com a *mesma assinatura* ainda definem *diferentes tipos*

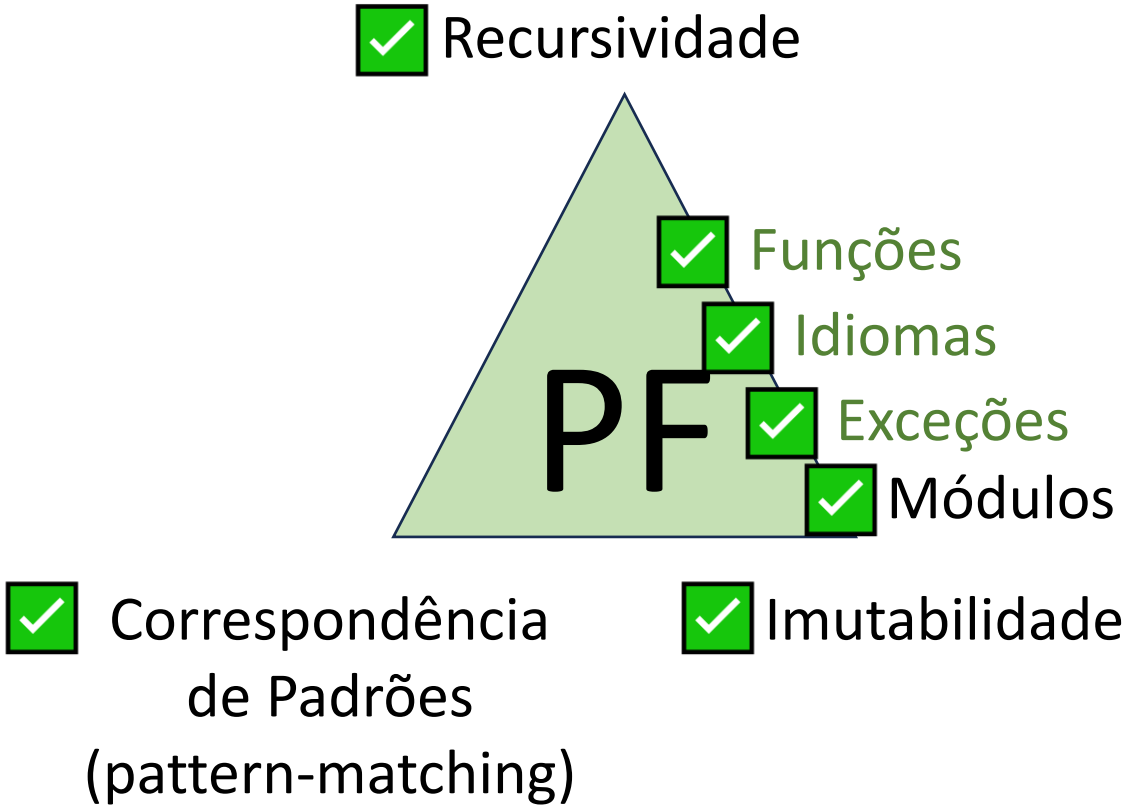
Por isso, coisas como esta não tipificam (ainda bem!):

- **`Rational1.string_of_rational(Rational2.make_frac(9,6))`**
- **`Rational3.string_of_rational(Rational2.make_frac(9,6))`**

Este é um comportamento crucial para o sistema de tipos e propriedades de módulos:

- Módulos diferentes têm invariantes internas diferentes!
- De facto, têm definições de tipo diferentes
  - **`Rational1.rational`** parece-se com **`Rational2.rational`**, mas o utilizador e o type-checker não sabem isso
  - **`Rational3.rational`** é **`int*int`** e não é um tipo variante!

Até agora vimos



Créditos para Dan Grossman.