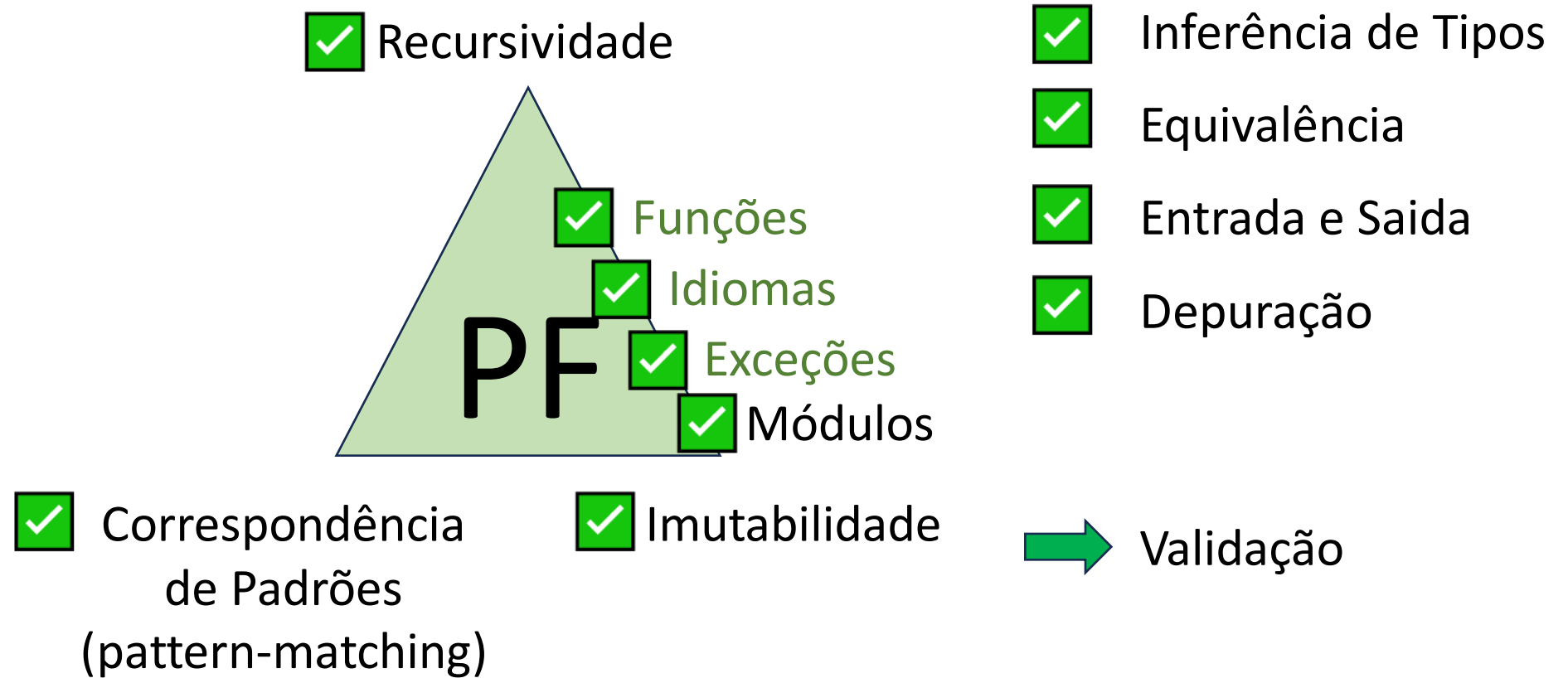


Aula 12: Testes Unitários em OCaml

UC: Programação Funcional
2023-2024

Até agora vimos



Bugs

- “bug” sugere que alguma coisa vagueou por aí

[IEEE 729]

- “Fault” resulta de erro humano no sistema informático
 - a implementação não corresponde à sua conceção, ou a sua conceção não corresponde aos requisitos
- “Failure” infringe um requisito
 - algo corre mal ao utilizador final

9/9

0800 Antan started
1000 " stopped - antan ✓

1300 (032) MP-MC 1.98264000
(033) PRO 2 2.130476415
conck 2.130676415

Relays 6-2 in 033 failed special speed test
in relay 11.00 test.

Relays changed

1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.

1545

Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

1630 Antan started.
1700 closed down.

Relay 3145
Relay 3371

Bugs

- **Depuração:** determina a causa de um comportamento não intencional
- **Programação defensiva:** implementa técnicas para facilitar a validação e a depuração
- **Validação:** o programa comporta-se como previsto ?
- **Teste:** um processo de validação

Defesas contra Bugs

De acordo com Rob Miller existem quatro defesas contra os bugs:

1. A primeira defesa contra os bugs é torná-los impossíveis

Classes inteiras de bugs podem ser erradicadas se optarmos por programar em linguagens que garantam a segurança da memória (“memory safety”) e segurança dos tipos (“type safety”).

- Garantimos que nenhuma parte da memória pode ser acedida exceto através de um ponteiro (ou referência) que seja válido para essa região da memória
- Garantimos que nenhum valor pode ser utilizado de forma inconsistente com o seu tipo.

O sistema de tipos do OCaml, por exemplo, evita que os programas tenham “buffer overflows” e operações sem sentido (como adicionar um booleano a um float), enquanto o sistema de tipos do C não o faz.

Defesas contra Bugs

2. Utilizar ferramentas que os encontrem

Existem ferramentas automatizadas de análise de código-fonte, como o **FindBugs**, que pode encontrar muitos tipos comuns de erros em programas Java, e o **SLAM**, que é utilizado para encontrar erros em controladores de dispositivos (essencialmente no Windows).

- Os métodos formais ajudam a usar a matemática para especificar e verificar programas, ou seja, como provar que os programas não têm bugs.
- Os métodos sociais, como as revisões de código e a programação em pares, também são ferramentas úteis para encontrar bugs.
 - Num estudo (Jones, 1991), a inspeção de código encontrou 65% dos erros de codificação conhecidos e 25% dos erros de documentação conhecidos, enquanto os testes encontraram apenas 20% dos erros de codificação e nenhum dos erros de documentação.

Defesas contra Bugs

3. Torna-los imediatamente visíveis

Quanto mais cedo um erro aparecer, mais fácil será o seu diagnóstico e correção. Se, em vez disso, a computação prosseguir para além do ponto do erro, então essa computação adicional pode ocultar onde a falha realmente ocorreu.

As asserções no código-fonte fazem com que os programas “falhem rapidamente” e “falhem em voz alta”, para que os erros apareçam imediatamente e o programador saiba exatamente onde os procurar no código-fonte.

Defesas contra Bugs

4. Testes exaustivos

Como é que se pode saber se uma parte do código tem um determinado erro?

- Escrevemos testes que exponham o erro e depois confirmamos que o código não falha nesses testes.
 - Produzimos os testes unitários para um bloco de código relativamente pequeno, como uma função ou módulo, ao mesmo tempo que desenvolvemos esse código.

A execução destes testes deve ser automatizada para que, se falharem, possamos descobrir o bug o mais rapidamente possível.

Depois de todas essas defesas terem falhado, um programador é forçado a recorrer à depuração.

Depuração

- **Depuração:** determina a causa de um comportamento não intencional
- **Programação defensiva:** implementa técnicas para facilitar a validação e a depuração
- **Validação:** o programa comporta-se como previsto ?
- **Teste:** um processo de validação



Depuração

Existem essencialmente três formas de como podemos depurar (fazer debug) de programas em OCaml.

1. Instruções de impressão

Insira uma instrução de impressão para verificar o valor de uma variável. Suponha que quer saber qual é o valor de `x` na seguinte função:

```
let inc x = x + 1
```

Basta adicionar a linha abaixo para imprimir esse valor:

```
let inc x =  
    let () = print_int x in  
    x + 1
```

Depuração

2. Traços de funções

Suponha que quer ver o traço de chamadas recursivas e retornos de uma função. Utilize a diretiva **#trace** :

```
# let rec fib x = if x <= 1 then 1 else fib (x - 1) + fib (x - 2);;  
# #trace fib;;
```

Se avaliar **fib 2**, verá o seguinte resultado:

```
fib <-- 2  
fib <-- 0  
fib --> 1  
fib <-- 1  
fib --> 1  
fib --> 2
```

Para parar o rastreo, utilize a diretiva **#untrace**.

Depuração

3. Depurador (debugger)

O OCaml tem uma ferramenta de depuração **ocamldebug**.

Podemos encontrar um [tutorial](#) no site do OCaml.

ocamldebug funciona apenas em programas compilados em bytecode pelo **ocamlc** (não funciona em executáveis de código nativo).

A menos que esteja a usar o VSCode como editor, provavelmente achará inicialmente esta ferramenta mais difícil de usar do que apenas inserir instruções de impressão.

Alguns concelhos de depuração

- O “bug” está provavelmente onde vocês não pensam que está...
 - Perguntem a vocês mesmos onde é que ele não pode estar.
- Perguntem a alguém com mais experiência para vos ajudar...
- Se tudo falhar, duvidem da vossa sanidade mental
 - Têm o compilador correto? E o código fonte correto?
- Não depurem código quando estiverem exaustos ou furiosos
 - Façam uma pausa, e regressem de novo.
- Analisem cuidadosamente a correção
 - A correção de um “bug” conduz frequentemente a novos “bugs”.

Bugs

- **Depuração:** determina a causa de um comportamento não intencional
- **Programação defensiva:** implementa técnicas para facilitar a validação e a depuração
- **Validação:** o programa comporta-se como previsto ?
- **Teste:** um processo de validação



Programação defensiva

- Uma das defesas contra bugs é tornar quaisquer bugs (ou erros) imediatamente visíveis (**defesa 3**).

Exemplo:

- A função **random_int bound** deve devolver um número inteiro aleatório entre **0** (inclusive) e **bound** (exclusive)
- O argumento **bound** deve ser maior do que **0** e menor do que 2^{30}

Ao utilizarmos a função **random_int** podemos passar um valor em **bound** que viole a regra, tal como **-1**, a implementação de **random_int** é livre de fazer o que quer que seja (na maioria das vezes fica indefinida e insegura).

Programação defensiva

```
(* possibilidade 1 *)  
let random_int bound =  
  assert (bound > 0 && bound < 1 lsl 30);  
  (* prosseguir com a implementação da função *)  
  
(* possibilidade 2 *)  
let random_int bound =  
  if not (bound > 0 && bound < 1 lsl 30)  
  then invalid_arg "bound";  
  (* prosseguir com a implementação da função *)  
  
(* possibilidade 3 *)  
let random_int bound =  
  if not (bound > 0 && bound < 1 lsl 30)  
  then failwith "bound";  
  (* prosseguir com a implementação da função *)
```


Programação defensiva ou depuração proactiva


A programação defensiva é dispendiosa ?

É o que nos parece.

Para o programador: o código defensivo que é escrito tende a compensar mais tarde através das falhas que são detetadas.

Para o desempenho: as falhas detetadas em produção devem poupar mais custos do que o tempo de execução das verificações.

Bugs

- **Depuração:** determina a causa de um comportamento não intencional
- **Programação defensiva:** implementa técnicas para facilitar a validação e a depuração
- **Validação:** o programa comporta-se como previsto ? 
- **Teste:** um processo de validação

Abordagens de Validação

- **Social**

- Revisão de código

- Programação por pares

- **Metodológica**

- Desenvolvimento orientado para o teste

- Controlo de versões

- Rastreio de bugs (“bug tracking”)

- **Tecnológica**

- Análise estática

- Testes Fuzz

- **Matemática**

- Sistemas de tipos

- Verificação formal



Menos formais

Estas técnicas podem não detetar problemas em programas

Todos estes métodos devem ser usados

Mesmo os métodos mais formais podem ter problemas:

- Será que provamos a coisa certa?
- Será que as nossas suposições correspondem à realidade?

Mais formais

Estas técnicas eliminam com certeza o maior número possível de problemas em programas

Teste Vs. Verificação


- Teste

- Rentável
- Garante que um programa é correto para as entradas de teste nos ambientes de teste

- Verificação

- Dispendiosa
- Garante que um programa é correto para todas as entradas e em todos os ambientes

Bugs

- **Depuração:** determina a causa de um comportamento não intencional
- **Programação defensiva:** implementa técnicas para facilitar a validação e a depuração
- **Validação:** o programa comporta-se como previsto ?
- **Teste:** um processo de validação 

Testes

- Testes revelam falhas no programa
- Depuração revela a causa dessas falhas
- A depuração leva mais tempo do que a programação
 - Por isso tentem acertar à primeira!
 - Tentem compreender exatamente por que razão consideram que o código funciona antes de o depurar!

Testes unitários com o OUnit

- **Teste unitário:** testa uma pequena parte da funcionalidade de um programa, como uma função individual.
- Usar o Toplevel para testar funções só funcionará para programas muito pequenos.
- Programas maiores precisam de conjuntos de testes que contenham vários testes unitários e que possam ser executados novamente sempre que atualizarmos nossa base de código (codebase).
- O fluxo de trabalho básico para usar OUnit é o seguinte:
 - Escrever uma função num ficheiro **f.ml**. Também pode haver muitas outras funções nesse ficheiro.
 - Escrever os testes unitários para essa função num ficheiro separado **teste.ml**. O nome exato não é essencial.
 - Construir e executar o binário **teste** para correr os testes unitários.
 - A documentação do OUnit está disponível no [GitHub](#).

Exemplo com o OUnit

- Coloquemos num determinado diretório o ficheiro **soma.ml** com a função

```
let rec sum = function
| [] -> 0
| x :: xs -> x + sum xs
```

- Agora, num segundo ficheiro **teste.ml** colocamos o código de teste

```
open OUnit2
open Soma
let tests = "test suite for sum" >::: [
  "empty" >:: (fun _ -> assert_equal 0 (sum []));
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]));
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]));
]
let _ = run_test_tt_main tests
```

- Dependendo do editor e da configuração, é provável que encontrem alguns erros, por exemplo, de “Unbound module” relativamente aos módulos OUnit2 e Soma

Exemplo com o OUnit

- Executamos o conjunto de testes e obtemos a saída

```
$ ./teste  
...  
Ran: 3 tests in: 0.10 seconds.  
OK  
$
```

- Suponhamos que modificamos `soma.ml` para introduzir um erro, alterando o código nele contido para o seguinte:

```
let rec sum = function  
| [] -> 1 (* bug *)  
| x :: xs -> x + sum xs
```

- Se recompilarmos e voltarmos a executar o conjunto de testes, todos os casos de teste falham

Testes unitários

- A saída diz-nos os nomes dos casos que falharam. Aqui está o início da saída com algumas simplificações:

```
$ ./teste
FFF
=====
Error: test suite for sum:1:singleton.

File "/aula12/teste/oUnit-test suite for localhost.log", line 2, characters 1-1:
Error: test suite for sum:1:singleton (in the log).

Raised at OUnitAssert.assert_failure in file "src/lib/ounit2/advanced/oUnitAssert.ml", line 45, characters 2-27
Called from OUnitRunner.run_one_test.(fun) in file "src/lib/ounit2/advanced/oUnitRunner.ml", line 83, characters 13-26

not equal
-----
(...)
-----

Ran: 3 tests in: 0.14 seconds.
FAILED: Cases: 3 Tried: 3 Errors: 0 Failures: 3 Skip: 0 Todo: 0 Timeouts: 0.
```

Testes unitários em detalhe

- `open OUnit2` inclui as várias funções em OUnit2 enquanto `open Soma` inclui o nosso módulo `soma.ml` (ficheiros são módulos, certo!).
- Construámos uma lista de casos de teste onde cada linha de código é um caso de teste separado.

```
[  
  "empty" >:: (fun _ -> assert_equal 0 (sum []));  
  "singleton" >:: (fun _ -> assert_equal 1 (sum [1]));  
  "two_elements" >:: (fun _ -> assert_equal 3 (sum [1; 2]));  
]
```

- Um caso de teste tem uma cadeia de caracteres que lhe dá um nome descritivo e uma função para ser executada como caso de teste. Entre o nome e a função, escrevemos `>::`, que é um operador personalizado definido pelo OUnit2.
- Também podemos testar exceções!

Testes unitários em detalhe

- Cada função de caso de teste recebe como entrada um parâmetro que OUnit2 chama de contexto de teste

```
fun _ -> assert_equal 0 (sum [])
```

- A função então chama **assert_equal**, que é uma função fornecida pelo OUnit2 e que verifica se seus dois argumentos são iguais. Se assim for, o caso de teste é bem sucedido. Caso contrário, o caso de teste falha.
- O operador **>:::** é outro operador personalizado do OUnit

```
let tests = "test suite for sum" >::: [ ]  
let _ = run_test_tt_main tests
```

Desenvolvimento orientado para o teste

- Os testes não têm de ser efetuados estritamente depois de se escrever o código.
- No desenvolvimento orientado para o teste, o teste vem em primeiro lugar!
- Este método enfatiza o desenvolvimento incremental do código: há sempre algo que pode ser testado.
- O teste não é algo que acontece após a implementação; em vez disso, o teste contínuo é utilizado para detectar erros numa fase inicial.
- Assim, é importante desenvolver testes unitários imediatamente quando o código é escrito.
- A automatização dos conjuntos de testes é crucial para que os testes contínuos não exijam praticamente nenhum esforço.

Desenvolvimento orientado para o teste

- Escolhemos deliberadamente uma função extremamente simples de implementar, para que o processo seja claro. Suponhamos que estamos a trabalhar com um tipo de dados para dias

```
type dia = Domingo | Segunda | Terça | Quarta | Quinta | Sexta | Sábado
```

- E queremos escrever uma função `dia_da_semana_seguinte : dia -> dia` que devolve o dia da semana seguinte dado um determinado dia como argumento.
- Começamos por escrever a versão mais básica dessa função

```
let dia_da_semana_seguinte = failwith "Não implementado!"
```

- A função incorpora o `failwith` que levanta uma exceção juntamente com a mensagem de erro `"Não implementado!"`.
- Escrevemos um teste unitário simples. Por exemplo, sabemos que o próximo dia da semana depois de segunda-feira é terça-feira

```
let testes = "suite de testes para dia_da_semana_seguinte" >::: [  
  "ter_depois_de_seg" >:: (fun _ -> assert_equal Terça (dia_da_semana_seguinte  
    Segunda));
```

Créditos para Michael Clarkson.