

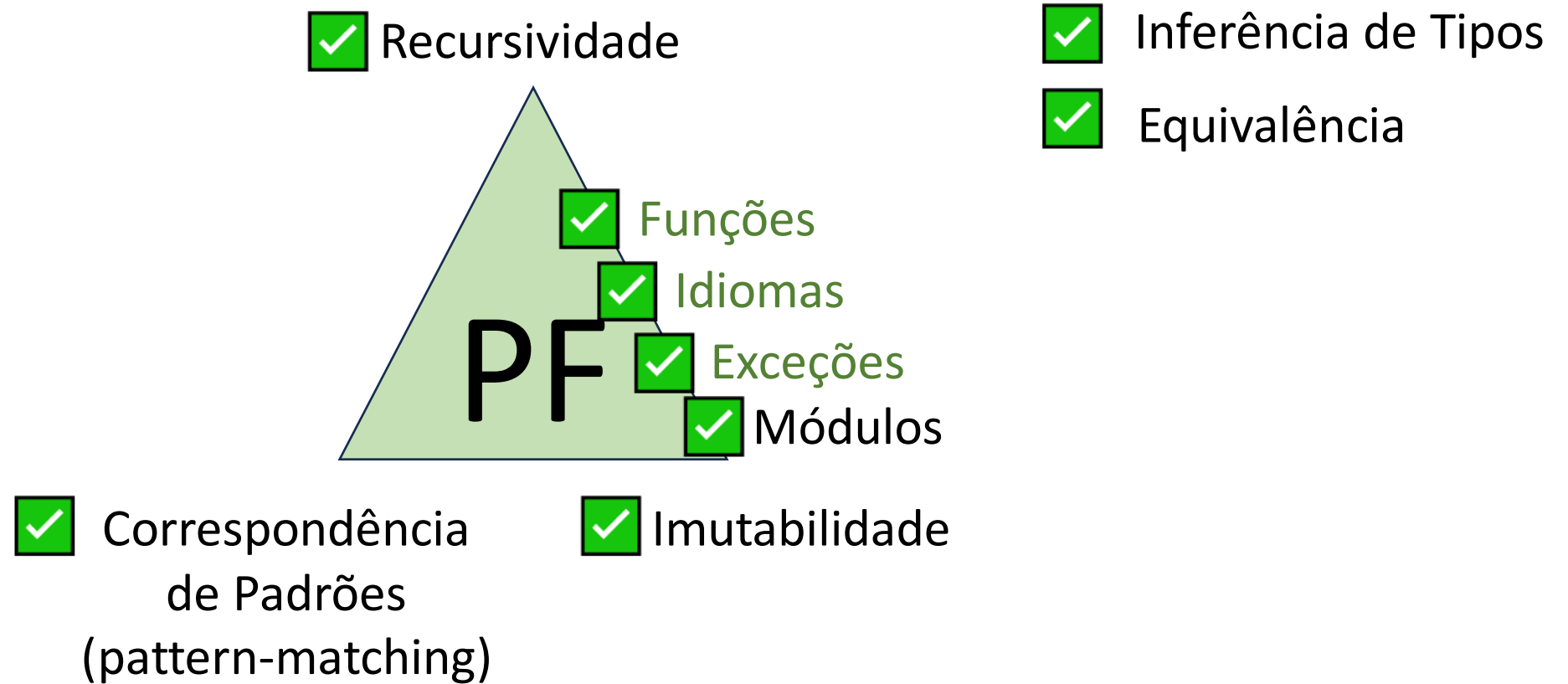
Aula 10: Motivação e História de PF

UC: Programação Funcional
2023-2024

Elegância e simplicidade em programação

- “Beauty is more important in computing than anywhere else in technology because software is so complicated. **Beauty is the ultimate defence against complexity.**” — David Gelernter
- “When I am working on a problem, I never think about beauty. I think only how to solve the problem. But when I have finished, **if the solution is not beautiful, I know it is wrong.**” — R. Buckminster Fuller
- “Complexity has nothing to do with intelligence, **simplicity does.**” — Larry Bossidy
- “**Simplicity** is prerequisite for **reliability.**” — Edsger W. Dijkstra
- “Sometimes, **the elegant implementation is a function.** Not a method, not a class, not a framework. Just a function.” — John Carmack
- “A language that doesn’t affect the way you think about programming is not worth knowing.” — Alan J. Perlis (1922-1990), primeiro galardoado com o Prémio Turing

Até agora vimos



Vamos ver

- Um pouco de História
- Origens da linguagem OCaml
- Quem utiliza OCaml
- A comunidade OCaml

Hilbert

David Hilbert, em 1928, confiante na reconstrução da matemática operada na altura, enunciou o desafio seguinte:

Problema da Decisão: Poderemos definir um **processo** que possa determinar num **número finito** de **operações** se uma determinada fórmula lógica (predicativa de primeira ordem) é verdade ou falsa?



Este desafio pertence ao conjunto dos atos fundadores da informática. Mais será dito em TC, mas em síntese o desafio pergunta se existe um mecanismo (automático) que possa revolver genericamente qualquer problema matemático

No fundo, **podemos resumir a matemática a algo que virá a chamar-se de informática?**

Hilbert

Problema da Decisão: Poderemos definir um **processo** que possa determinar num **número finito** de **operações** se uma determinada fórmula lógica é verdade ou falsa?



Podemos resumir a matemática a algo que virá a chamar-se de informática?

Rapidamente a resposta veio a ser conhecida, e não foi positiva. Foi **duplamente negativa**

Gödel

Propriedade de Correção: o que se estabelece, está sempre certo.

Propriedade de Completude: consegue-se sempre estabelecer uma conclusão.

Teoremas de Incompletude: nenhum sistema axiomático (e.g. lógico) não trivial pode ser simultaneamente completo e correto. Para garantir a correção (o que é fundamental) é preciso abdicar da completude.



Assim

a Matemática é um edifício em perpétua construção e remodelação

em jeito de curiosidade, a autorreferência (e.g. recursividade) e o paradoxo do mentiroso têm o seu papel nesta história.

Voltando ao problema da decisão

Percebemos que a procura da verdade absoluta é um caminho sem fim à vista

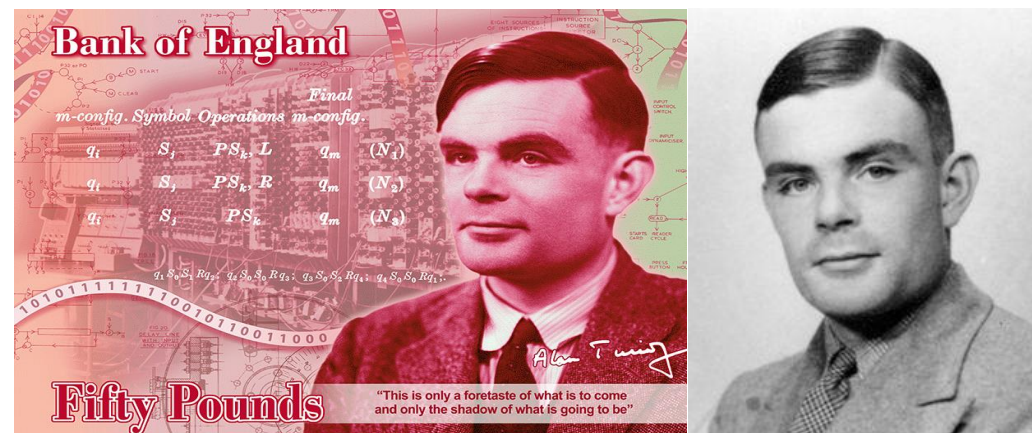
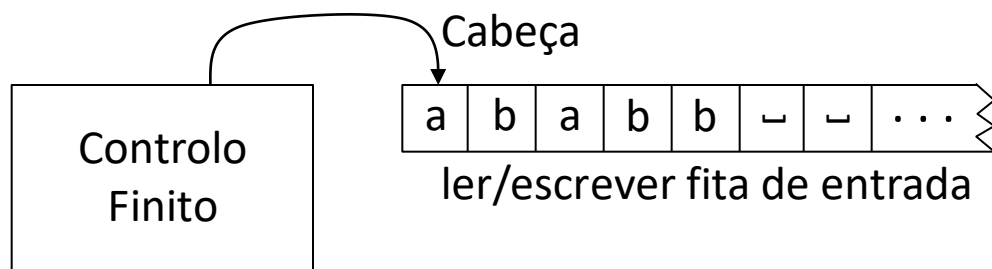
Mas se aceitarmos jogar em terrenos menos vastos do que a matemática na sua generalidade e considerarmos teorias corretas e completas (mas menos abrangentes) ou então abdicarmos da completude, será que podemos reduzir estas matemáticas a um processo mecânico automatizável?

Para responder a este desafio é preciso definir com rigor o que entendemos por **operações** e **processo de cálculo** (o que é a informática?)

Turing e Church propuseram definições para estes conceitos e deram uma segunda resposta negativa ao desafio.

Turing

Inventou o conceito de Máquina de Turing



Apareceram em Inglaterra em homenagem a Turing em 2021

Estas máquinas dão um fundamento teórico às arquiteturas de computadores e à programação imperativa.

A fita tem o papel da **memória** (onde se encontram dados e programas) o autómato o do **processador**.

Máquinas de Turing e o paradigma imperativo

Um programa imperativo **lê, escreve, executa operações** e **“toma decisões”** com base no conteúdo de células de memória que contêm informação sobre as variáveis do programa, como **c, n, res** no seguinte programa em C (que calcula o fatorial).

```
int fatorial(int n)
{
    int res = 1;
    for (int c = 1; c <= n; c++)
        res = res * c;
    return res;
}
```

Church

Tal como Alan Turing, em 1936, Alonzo Church (orientador de Turing) dá uma definição alternativa de algoritmo e da execução de programas.

Responde pela negativa à questão de D. Hilbert.

A sua definição é de natureza bem diferente da das máquinas de Turing, embora se descubra cedo que ambas são equivalentes.

A. Church introduz o **cálculo lambda** (λ -calculus)

- Elemento de base: x , variável.
- Abstração: $\lambda x.M$, função anónima de um parâmetro formal, x e de corpo M
- Aplicação: $(M\ N)$, a função M aplicada ao parâmetro efetivo N

Computação: a regra da redução β : $(\lambda x.M)N \rightarrow_{\beta} M[x := N]$

Programa: um termo do cálculo lambda

Execução: basta a aplicação da regra β (nada de hardware complexo)



Cálculo lambda e a programação funcional

Num **programa funcional**, **definimos funções** (possivelmente recursivas), **compomo-las** e **aplicamo-las** para **calcular** os **resultados** esperados.

Exemplo:

```
let rec factorial =  
  fun n -> if n = 0 then 1 else n * (factorial (n-1))
```

Numa linguagem de programação funcional **pura**, as funções são cidadãs de primeira classe, como qualquer outro valor (inteiros, booleanos ...) e podem ser

- Atribuídas com um nome (ou não, anónimas)
- Avaliadas
- Passadas como argumento (de outras funções)
- Devolvidas como resultado de funções
- Usadas em qualquer local onde se espera uma expressão (já não é novidade para nós)

Cálculo lambda e a programação funcional

Usando da notação original de A. Church podemos reescrever a expressão

```
fun n -> if n=0 then 1 else n * (fact (n-1))
```

na forma

$$\lambda n. \text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fact } (n - 1))$$

Na verdade, são as famosas expressões **lambda** que se introduziram no Java 8 ou C++11.

Elementos basilares da programação funcional que as linguagens ditas *mainstream* agora integram.

A tese de Church-Turing

A. Turing, em 1937, demonstrou que o calculo lambda e as máquinas de Turing são formalismos com a mesma expressividade: uma função é **computável** por uma máquina de Turing **se e só se** é **computável** no cálculo- λ

A **Tese de Church-Turing** foi então formulada e afirma que uma função **computável** (um algoritmo) em **qualquer formalismo computacional** é também **computável por uma máquina de Turing**.

em termos mais gerais:

- todas as linguagens de programação são **computacionalmente equivalentes**
- **qualquer algoritmo** pode ser expresso usando um destes formalismos (que são equivalentes)

Mas as linguagens de programação **não nascem todas iguais**

Por norma, computacionalmente equivalentes, mas têm comodidades e expressividades diferentes. A procura das construções programáticas mais expressivas ou mais cómodas é uma luta sem fim que leva a diferentes

- Formas de representação dos dados
- Modelos de execução
- Mecanismos de abstração

E muitas outras características desejáveis entram em conta no momento do desenho de novas linguagens de programação:

- Segurança da execução
- Eficiência
- Modularidade e capacidade de manutenção

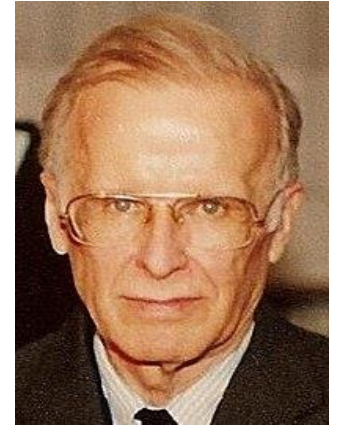
Dependendo do problema em causa, algumas linguagens de programação são mais adequadas do que outras.

Uma opinião do criador da linguagem FORTRAN

John Backus foi pioneiro na programação funcional.

“Functional programs deal with structured data, ... do not name their arguments, and do not require the complex machinery of procedure declarations ...”

“Can programming be liberated from the von Neumann style?” — John Backus, Turing lecture, 1977



Porque a programação funcional está para ficar?

- Uma reflexão sobre o ensino introdutório em informática na CMU (<http://www.cs.cmu.edu/~bryant/pubdir/cmu-cs-10-140.pdf>) destaca tendências emergentes das quais a
 - Necessidade de uma maior fiabilidade do software;
 - Os programas funcionais (puros) são mais fáceis de provar que estão correctos do que os imperativos;
 - Aproveitam o poder da computação paralela;
 - Um conjunto cuidadosamente escolhido de funções de ordem superior permite escrever programas que são facilmente paralelizáveis.
- Um exemplo conhecido que advém diretamente da programação funcional é o paradigma da “Cloud” **MapReduce**.

A onnipresença funcional

A **programação funcional** é longe de ser um exclusivo das linguagens de programação ditas funcionais, **pratica-se em todas as linguagens**.

As suas vantagens estão largamente reconhecidas clareza, elegância, simplicidade e expressividade:

- A **recursividade** é uma componente habitual de qualquer linguagem de programação
- Java 5 e C / .NET introduziram os **genéricos** (polimorfismo de tipos, nas linguagens funcionais tipificadas)
- Java 8 e C++ (v. 11) introduziram as expressões **lambda** (as funções puras, como já as descrevemos) e as construções relacionadas (fluxos)

A onnipresença funcional

A programação funcional em OCaml, Haskell ou Racket é a programação “virada para o futuro”.

Estamos a programar hoje utilizando funcionalidades que outras linguagens terão amanhã.

A onnipresença funcional

A programação funcional não é exclusiva das linguagens de programação funcionais; pode ser praticada **em todas as linguagens**.

As linguagens de programação recentes **assimilaram** nas suas construções as **boas práticas** que as linguagens de programação funcionais estabeleceram.

Linguagens como **Rust, Swift, Scala, Python, Typescript, Go, Closure** (sob a influência do paradigma funcional) têm a ver com a programação funcional.

A onnipresença funcional

Independentemente das vossas linguagens de programação favoritas, compreender os princípios da programação funcional é uma competência transversal importante que, sem dúvida, vos permitirá programar de forma mais elegante.

Exemplos de linguagens funcionais: **Haskell, Racket, Scheme, SML**, etc.

Mas começámos esta viagem pelas linguagens de programação funcionais com **OCaml**.

A origem da família de linguagens ML

JOURNAL OF COMPUTER AND SYSTEM SCIENCES 17, 348–375 (1978)

A Theory of Type Polymorphism in Programming

ROBIN MILNER

Computer Science Department, University of Edinburgh, Edinburgh, Scotland

Received October 10, 1977; revised April 19, 1978

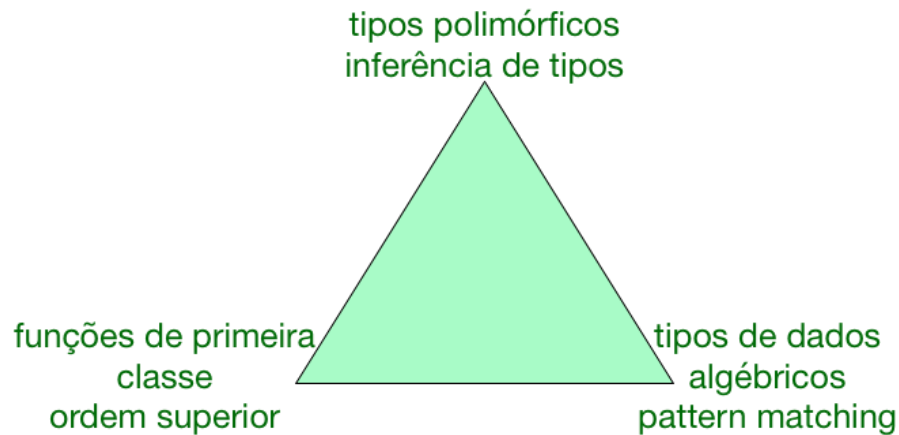
The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types, entails defining procedures which work well on objects of a wide variety. We present a formal type discipline for such polymorphic procedures in the context of a simple programming language, and a compile time type-checking algorithm \mathcal{W} which enforces the discipline. A Semantic Soundness Theorem (based on a formal semantics for the language) states that well-type programs cannot “go wrong” and a Syntactic Soundness Theorem states that if \mathcal{W} accepts a program then it is well typed. We also discuss extending these results to richer languages; a type-checking algorithm based on \mathcal{W} is in fact already implemented and working, for the metalanguage ML in the Edinburgh LCF system.

1. INTRODUCTION

The aim of this work is largely a practical one. A widely employed style of programming, particularly in structure-processing languages which impose no discipline of types (LISP is a perfect example), entails defining procedures which work well on objects of a wide variety (e.g., on lists of atoms, integers, or lists). Such flexibility is almost essential

- OCaml pertence à família das linguagens funcionais **fortemente tipificadas**, com **tipificação estática**.
- Esta família nasceu com o trabalho de Robin Milner (na linguagem **ML**).

As características da família de linguagens ML



Tipificação estática forte

Qualquer expressão da linguagem tem um tipo, rigorosamente verificado em tempo de compilação.

Inferência de tipos

Não precisamos de o definir, o compilador sabe como o computar.

Pattern-matching

é possível questionar a estrutura de um determinado dado e usá-la

História da linguagem OCaml

1980: o projeto Formal no INRIA, na direção de Gérard Huet

- trabalho pioneiro em mecanização da matemática
- utiliza originalmente a linguagem ML de Milner
- neste contexto faz contribuições importantes à linguagem (em particular o mecanismo de *pattern-matching*)
- resolve criar a sua própria versão de ML

Nascia o primeiro Caml

- 1985: Guy Cousineau, Pierre-Louis Curien e Michel Mauny criam a **Categorical Abstract Machine** (CAM)
- 1987: Ascender Suarez publica a primeira implementação de Caml
- 1988-1992: Michel Mauny e Pierre Weis alteram e estendam o Caml

Nesta fase a linguagem ganhou algum estatuto na comunidade académica mas era complexa e necessitava de computadores de alto desempenho (da época).

História da linguagem OCaml

- Início dos anos 90: a era do Caml-light
- Em 1990-1991 Xavier Leroy cria a **máquina abstracta Zinc**, Damien Doligez implementa um gestor de memória otimizado e estes esforços resultam no **Caml-light** (com alunos de 3º ano!)
 - Cabia numa disquete <1.44Mb (o binário era pequeno)
 - Era suportado num interpretador de bytecode (portável)
 - Podia ser executado em computadores pessoais (eficiente)
- Diz a lenda que eles queriam programar em casa, razão pela qual surgiu este projeto.

História da linguagem OCaml

A máquina Zinc era conceptualmente muito diferente da CAM, mas o nome ficou

- 1995: **Caml Special Light**: compilador nativo, sistema de módulos
- 1996: **Objective Caml**: camada objeto eficiente e elegante (Jérôme Vouillon e Didier Remy)
- 2000: Integração de métodos polimórficos, *labels*, argumentos opcionais, variantes polimórficos (Jacques Garrigue)
- 2011: o nome da linguagem estabiliza-se para **Ocaml**

Ao longo dos anos, o OCaml amadureceu e agora fornece um conjunto expressivo de mecanismos de programação.

Quem utiliza o OCaml no ensino

OCaml não é uma linguagem *mainstream*, mas é uma linguagem de programação com uma grande comunidade de utilizadores e é muito utilizada no meio académico.

- **Portugal:** UBI, FCT-UNL, U. Évora
- **França:** Univ. Paris Diderot, Pierre et Marie Curie, Paris Saclay, Rennes, École Normale Supérieure, École Polytechnique, etc.
- **Europa:** Univ. de Pisa, Bologna, Birmingham, Cambridge, Aarhus, Innsbruck, Varsóvia, etc.
- **Estados Unidos:** Cornell, Harvard, MIT, Pennsylvania, Princeton, CalTech, Chicago, etc.

E em alguns outros sítios ...

Quem utiliza o OCaml em projetos de Software e Hardware

Exemplos de projectos (de investigação):

- **Coq** – Sistema de prova (ACM Software System Award 2014)
- **Analizador estático Astrée** – Verificação de software (e.g., Airbus A380)
- **Plataforma de análise e verificação Frama-C** – Análise de programas em C
- **Ocsigen** – Framework para desenvolvimento de aplicações web (Javascript)
- **Alt-Ergo** – SMT solver
- **Mirage OS** – Unikernel ou Sistema Operativo Biblioteca
- **Flow / Hack** – *Type checkers* para PHP/Javascript
- **Infer** – Analizador estático de aplicações móveis (Android e iOS)
- **Hardcaml** – Hardcaml é uma biblioteca OCaml para projetar hardware
- E alguns outros

Quem utiliza o OCaml na Indústria

- Bloomberg – Finanças
- Citrix – Virtualização e Cloud
- Dassault – Aeroespacia
- Facebook
- JaneStreet Capital – Finanças
- LexiFi – Finanças
- Docker – Containers
- Google
- Microsoft
- RedHat – Distribuição Linux
- Cryptosense – Criptografia
- Trust-in-Soft – Segurança e fiabilidade
- Tezos – Finanças descentralizadas e blockchain
- etc.

Nos estudos relativos às profissões de engenharia de software, as ofertas que requerem o domínio e a especialização em programação funcional (em OCaml) revelam-se as mais bem remuneradas.

Eis alguns testemunhos.

Aplicações Web: Projeto Ocsigen

- O projeto Ocsigen tornou possível a escrita de aplicações web avançadas
- *OCaml's type system allows Ocsigen to **check statically advanced properties** of a Web application, like ensuring that a program will **never generate invalid HTML pages**, or that **a form has the expected fields**.*
- *The advantages of this powerful type system become obvious when **refactoring a large project**: the compiler points out every piece of code that needs to be modified, **saving days of testing and debugging**. — Vincent Balat, criador do Ocsigen (2015)*

Virtualização: Citrix, Xen

- Xen é um monitor de máquinas virtuais (*hypervisor*) utilizado na nuvem e algumas ferramentas de gestão e controlo das máquinas virtuais são desenvolvidas em OCaml.
- “OCaml has brought **significant productivity and efficiency benefits** to the project. OCaml has enabled our engineers to be more productive than they would have been had they adopted any of the mainstream languages.” — Richard Sharp, Citrix

Unikernel: Mirage OS

- A Citrix e a Universidade de Cambridge estão atualmente a desenvolver o sistema operativo de biblioteca Mirage OS, um unikernel para Xen inteiramente desenvolvido em OCaml
- “OCaml’s combination of static type safety and fast native code compilation has been essential to our MirageOS project, which re- builds operating system components (including TCP/IP and device drivers) in a safe, modular and flexible style.” — Anil Madhavapeddy, Universidade de Cambridge (2015)

Cibersegurança: TrustInSoft

- TrustInSoft fornece soluções inovadoras de software de segurança (“*security*” e “*safety*”)
- “OCaml generates code that is **very efficient** compared to other languages with similar expressivity. **Expressivity** is needed when developing sophisticated static analyzers. **Efficiency** is necessary when working at the frontier of what is possible at all on today’s computers. **Static typing** saves clock cycles at execution time and, more importantly, human time during development.” — Pascal Cuoq, TrustInSoft (2015)

Criptografia: Cryptosense

- Cryptosense desenvolve soluções de software para a auditoria de segurança em criptografia.
- *“We see OCaml as a strategic advantage. It helps us to **rapidly** produce **high-quality readable, reusable code**, which is essential for a start-up.” — Graham Steel, Cryptosense (2015)*

Garantir a segurança de código em sistemas embebidos críticos: Astrée

- Astrée é o analisador estático usado pela Airbus para provar a ausência de bugs nos sistemas de comando e controlo do A380.
- “A **type-safe functional language** was the natural choice to implement the Astrée analyzer. OCaml’s robust design supported a scalable development process, from research to industry, and we appreciated its **high performance native code compiler**.” — Antoine Miné, Researcher at CNRS & ENS (2015)

Provas por computador: o sistema de prova Coq

- O sistema de prova Coq é uma sistema de prova de referência mundial utilizado em provas por computador complexas.
- Por exemplo, realizaram-se com o seu auxílio, provas de seguranças complexas (**smart cards**, sistemas criptográficos, etc.)
- “Amongst all the great features of OCaml, **pattern-matching** is crucial for Coq: without it, implementing complex symbolic computations would be a nightmare!” — The Coq development team (2015)

Finanças: JaneStreet

- JaneStreet utiliza OCaml para a construção de ferramentas de *financial trading* capaz de lidar *com transações de alta velocidade de mais de 10 mil milhões de dollars por dia*
- “Our experience with OCaml on the research side convinced us that we could build **smaller, simpler, easier-to-understand systems** in OCaml than we could in languages such as Java or C. **For an organization that valued readability, this was a huge win...**”
- “There is, a surprisingly wide swath of bugs against which **the type system is effective**, including **many bugs that are quite hard to get at through testing.**”
— Yaron Minsky. Em OCaml for the masses. Communications of the ACM, September 2011

Finanças: LexiFi

- LexiFi desenvolve software inovador na área da finança que permite a gestão de produtos financeiros complexos, combinando computação numérica com computação simbólica avançada.
- *“Safety, readability, expressivity and great performance are often cited as key benefits of OCaml. We also value the portability of the system, as our products are deployed on Unix, Windows and in the web browser. Parts of our codebase which were historically written in C, C++ or Javascript are now in OCaml. As one of the earliest industrial adopters of OCaml, we are delighted to observe the growing interest and activity around OCaml in the last years.” — Alain Frisch, LexiFi (2015)*

Framework de desenvolvimento de software: OCamlPro

- OCamlPro é especialista no desenvolvimento de software no ecossistema OCaml.
- *“I have tried many programming languages, but none of them could compete with OCaml. In OCaml, you just **define the type of your data**, and the compiler will gently **drive you towards your destination**, at highspeed on a highway. It’s just fascinating!”
— Fabrice Le Fessant, OCamlPro (2015)*

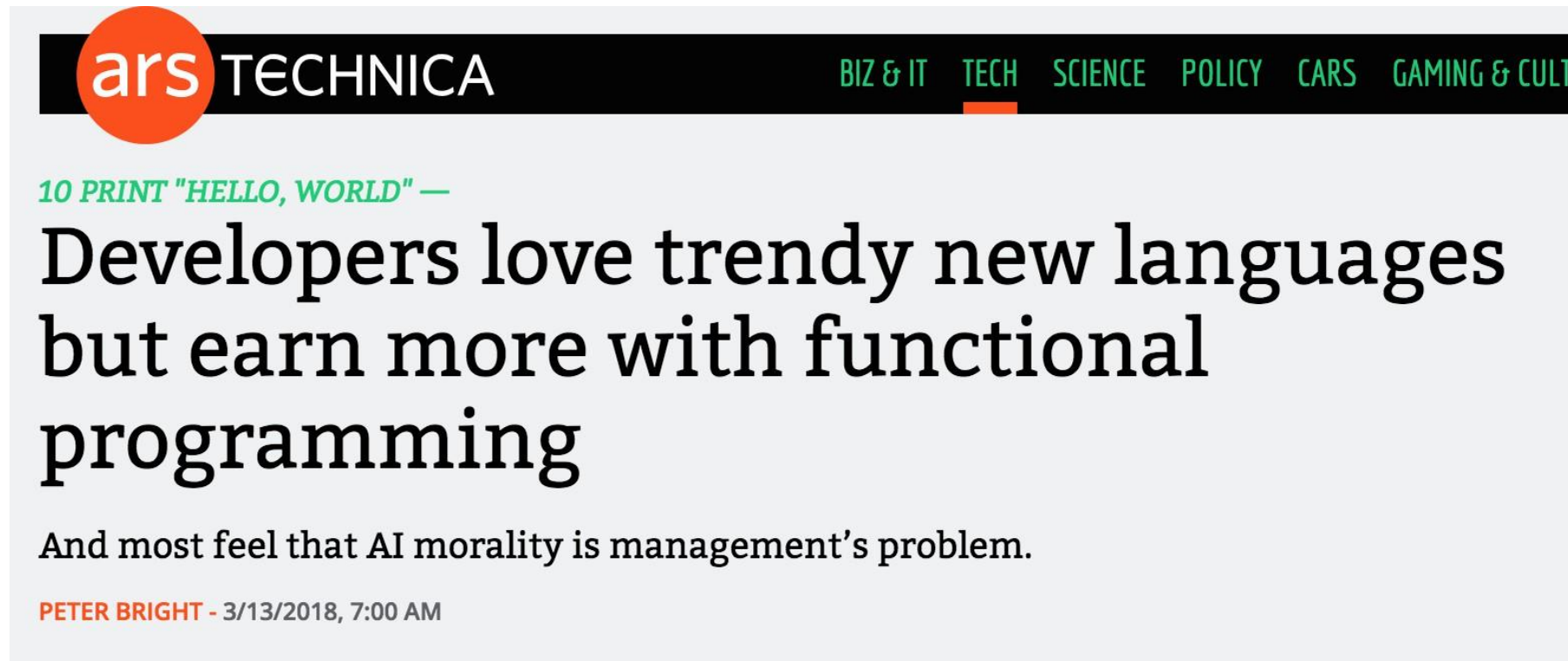
Em resumo, o que se diz sobre a utilização de OCaml

Existe uma variedade utilizações da linguagem OCaml.

De forma unânime os utilizadores valorizam:

- **Segurança**
 - Desde a tipificação estática forte até ao *pattern-matching*
- **Eficiência**
 - Compilador de alto desempenho
- **Expressividade**
 - Combinação de uma linguagem funcional com inferência de tipos e polimorfismo

OCaml, vale a pena em termos de carreira?

The image is a screenshot of the top portion of an Ars Technica article. At the top, there is a black navigation bar. On the left side of this bar is the Ars Technica logo, which consists of an orange circle with the word 'ars' in white lowercase letters, followed by the word 'TECHNICA' in white uppercase letters. To the right of the logo, the navigation bar contains several menu items in green uppercase letters: 'BIZ & IT', 'TECH', 'SCIENCE', 'POLICY', 'CARS', and 'GAMING & CULT'. The 'TECH' item is highlighted with a small orange rectangle underneath it. Below the navigation bar, the article title is displayed in a large, bold, black serif font. Above the title, there is a line of text in a smaller, green, italicized serif font. Below the title, there is a line of text in a smaller, black serif font. At the bottom of the article header, there is a line of text in a smaller, orange serif font, followed by a date and time in a smaller, black serif font.

ars TECHNICA

BIZ & IT TECH SCIENCE POLICY CARS GAMING & CULT

10 PRINT "HELLO, WORLD" —

Developers love trendy new languages but earn more with functional programming

And most feel that AI morality is management's problem.

PETER BRIGHT - 3/13/2018, 7:00 AM

[Ars Technica, 3/13/2018 \(link\)](#)

Créditos para Simão Melo de Sousa.