Foundations of Computer Science
Lecture 12:
Procedural Programming
& Recap

Anil Madhavapeddy & Amanda Prorok

2019-2020

References: ML Versus Conventional Languages

- We must write `!p` to get the *contents* of `p`
- We write just `p` for the address of `p`

- We can store *private* reference cells in functions; simulating object oriented programming

- OCaml's assignment syntax is $V := E$ instead of $V = E$

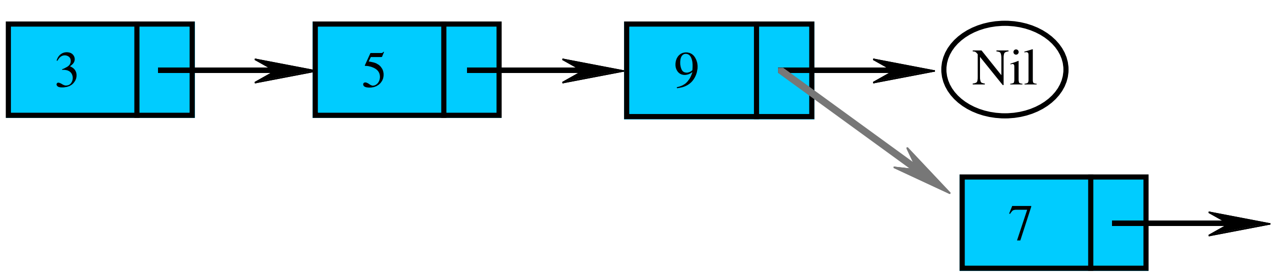
- OCaml has similar control structures: `while/done`, `for/done` and `match/with`

- OCaml has short syntax for updating arrays $x\,.\,(1)$ and the access is safe against buffer overflows

What More Is There to ML?

With references, we can now make mutable linked lists

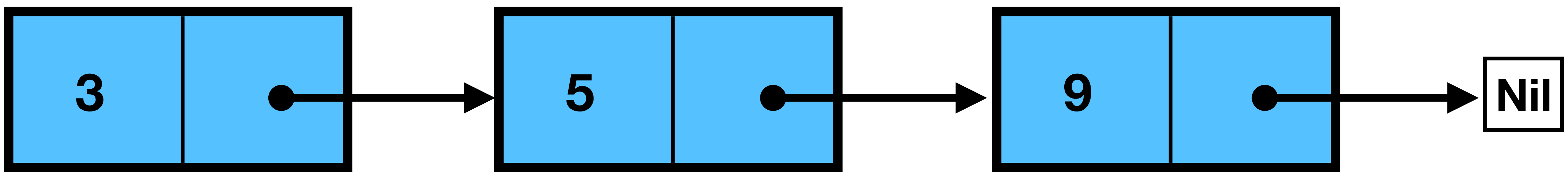
```
# type 'a mlist =
   | Nil
   | Cons of 'a * 'a mlist ref
type 'a mlist = Nil | Cons of 'a * 'a mlist ref
```



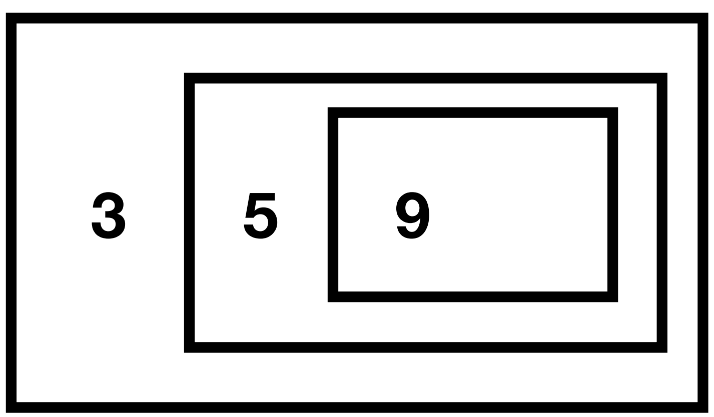
References to References

Two ways to visualize references to references:

(1) Using pointers:



(2) Using nested boxes:



Linked (Mutable) Lists

```
# type 'a mlist =
   | Nil
   | Cons of 'a * 'a mlist ref
```

*type 'a mlist = Nil | Cons of 'a * 'a mlist ref*

→ The tail can be redirected!

Linked (Mutable) Lists

```
# type 'a mlist =
   | Nil
   | Cons of 'a * 'a mlist ref
type 'a mlist = Nil | Cons of 'a * 'a mlist ref
```

→ The tail can be redirected!

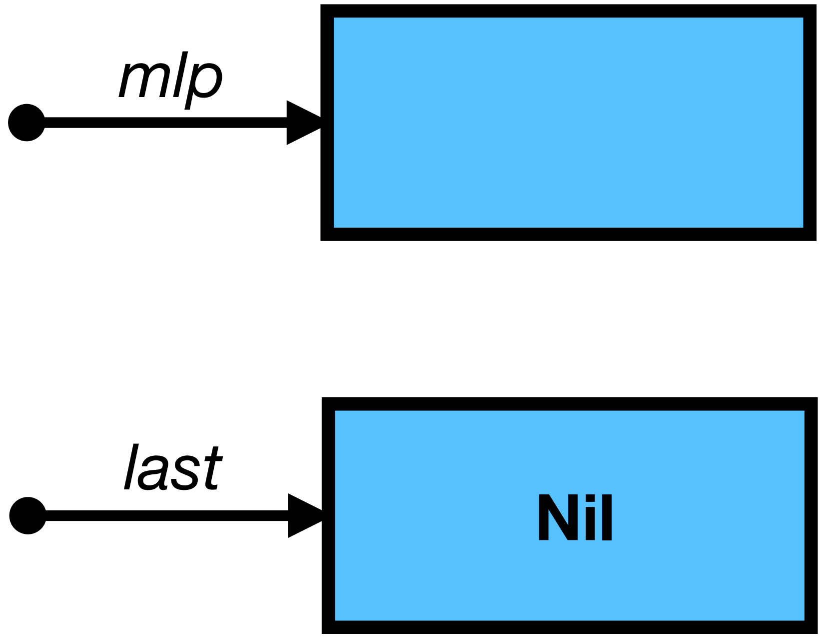
creates a new pointer to rest of mlist

```
# let rec mlistOf = function
   | [] -> Nil
   | x :: l -> Cons (x, ref (mlistOf l))
mlist : 'a list -> 'a mlist = <fun>
```

Extending a List to the Rear

pointing to a 'box'

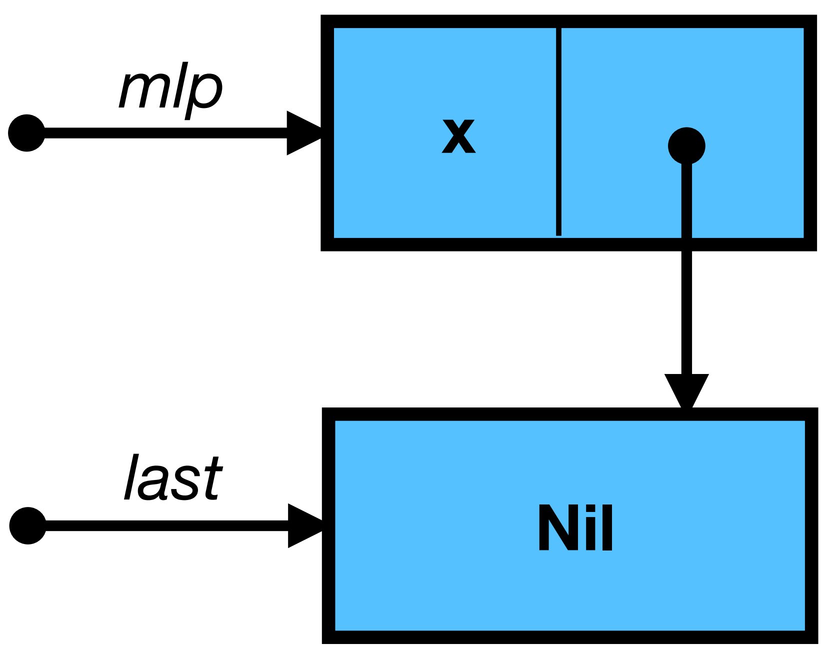
```
# let extend mlp x =
    let last = ref Nil in
    mlp := Cons (x, last);
    last
> val extend = fn : 'a mlist ref * 'a -> 'a mlist ref
```



Extending a List to the Rear

pointing to a 'box'

```
# let extend mlp x =
   let last = ref Nil in
   mlp := Cons (x, last);
   last
> val extend = fn : 'a mlist ref * 'a -> 'a mlist ref
```

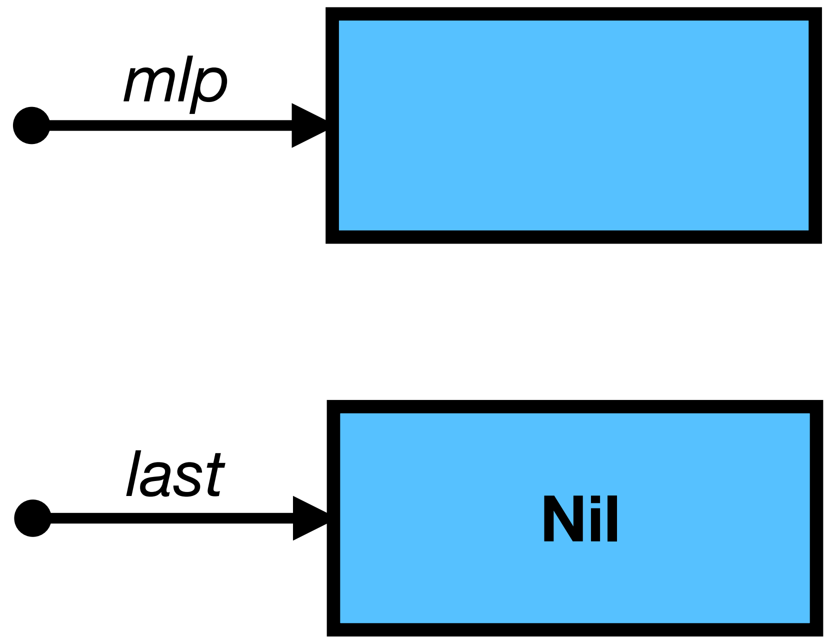


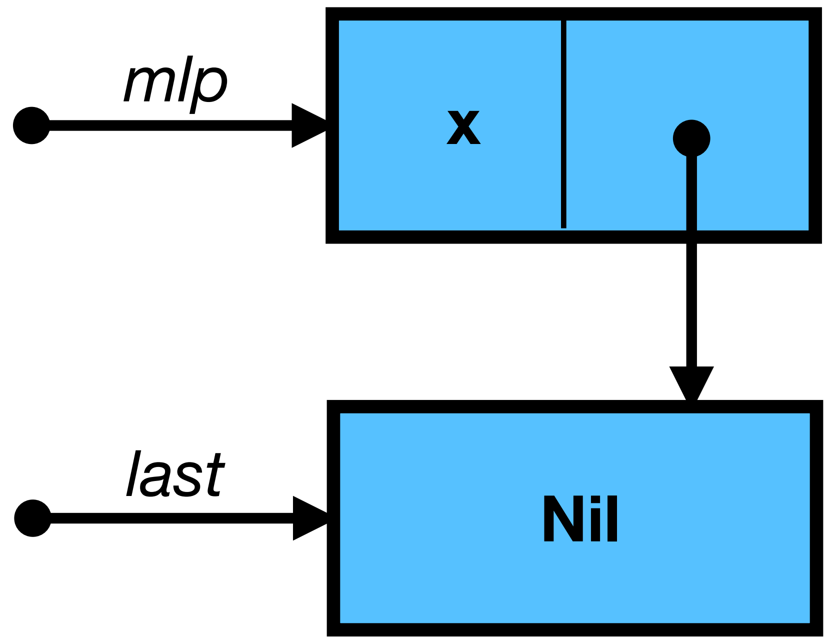
Example of Extending a List

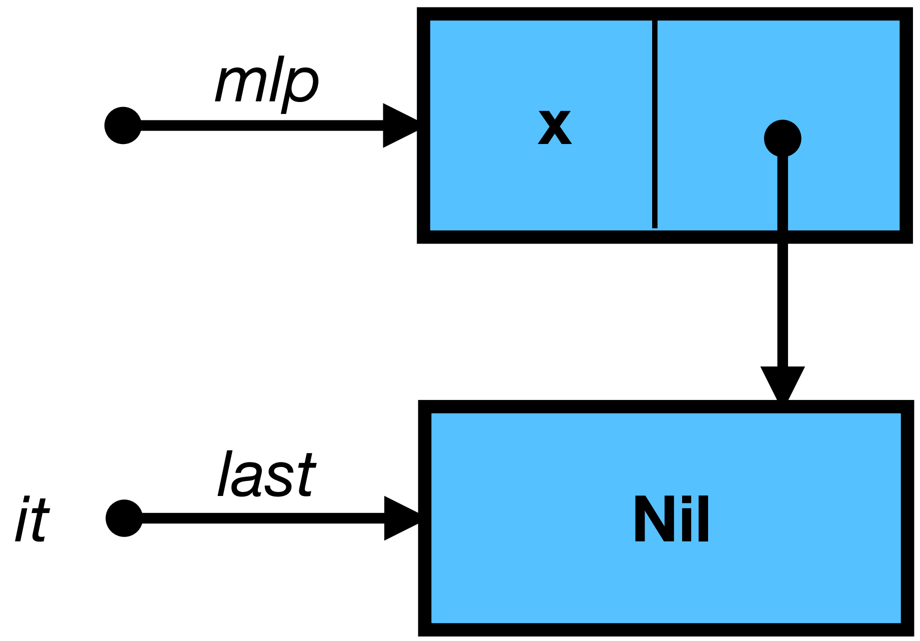
```
# let mlp = ref (Nil: string mlist);;
val mlp : string mlist ref = {contents = Nil}

# extend mlp "a";;
- : string mlist ref = {contents = Nil}
```

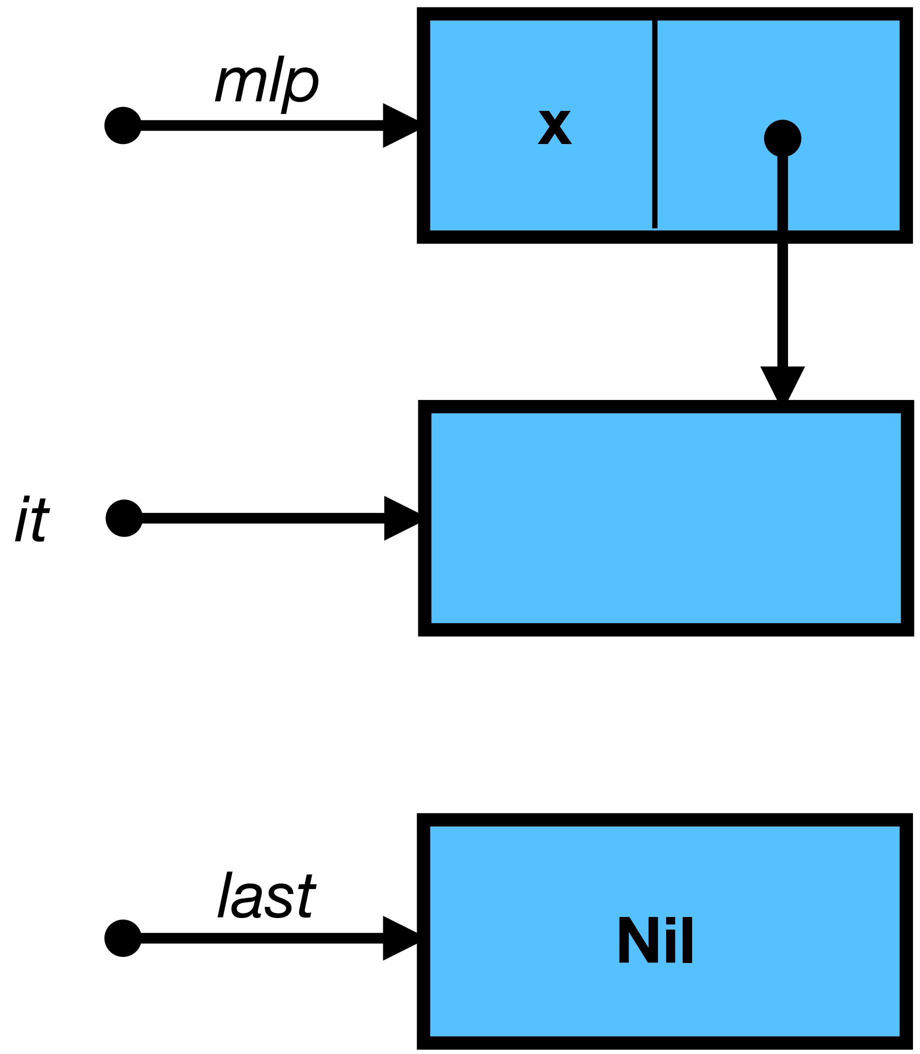
Example of Extending a List

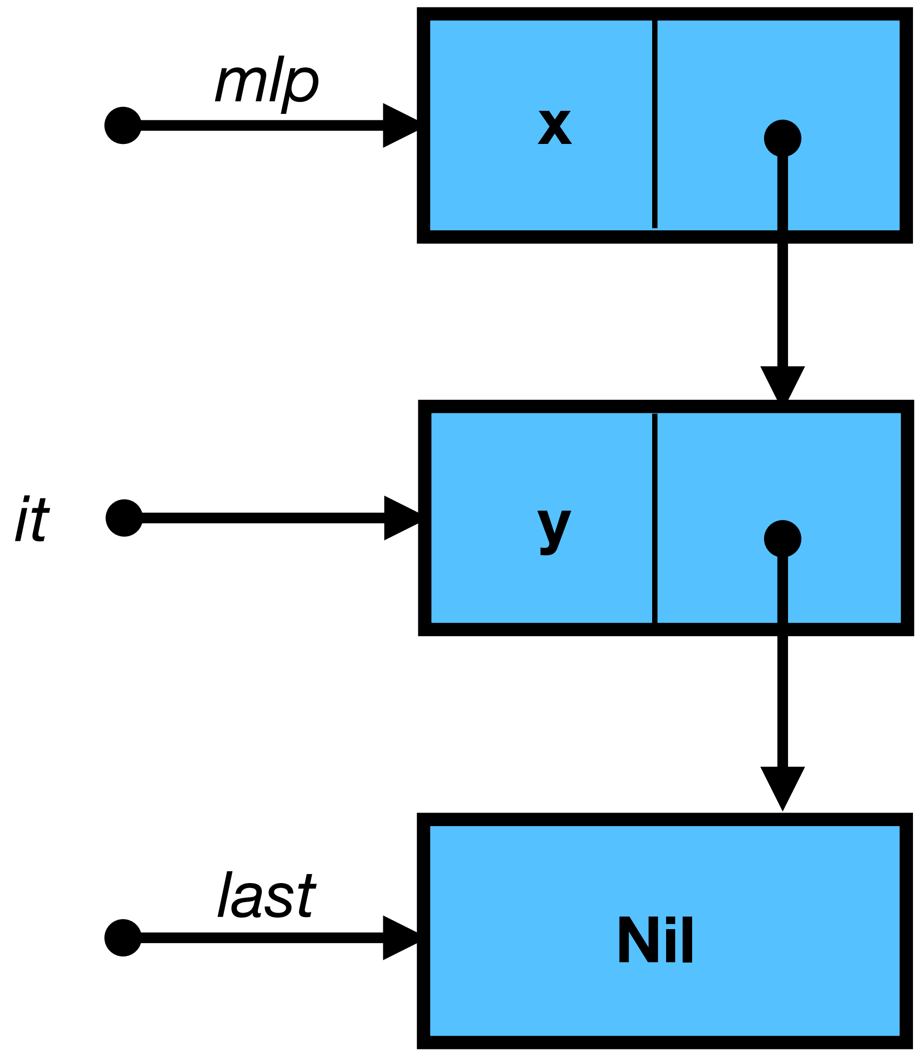
```
# let mlp = ref (Nil: string mlist);;
val mlp : string mlist ref = {contents = Nil}

# extend mlp "a";;
- : string mlist ref = {contents = Nil}

# let mlp = ref (Nil : string mlist);;
val mlp : string mlist ref = {contents = Nil}

# let it = extend mlp "a" ;;
val it : string mlist ref = {contents = Nil}

# let it = extend it "b" ;;
- : string mlist ref = {contents = Nil}

# mlp ;;
- : string mlist ref =
{contents = Cons ("a",
  {contents = Cons ("b", {contents = Nil})})}
```

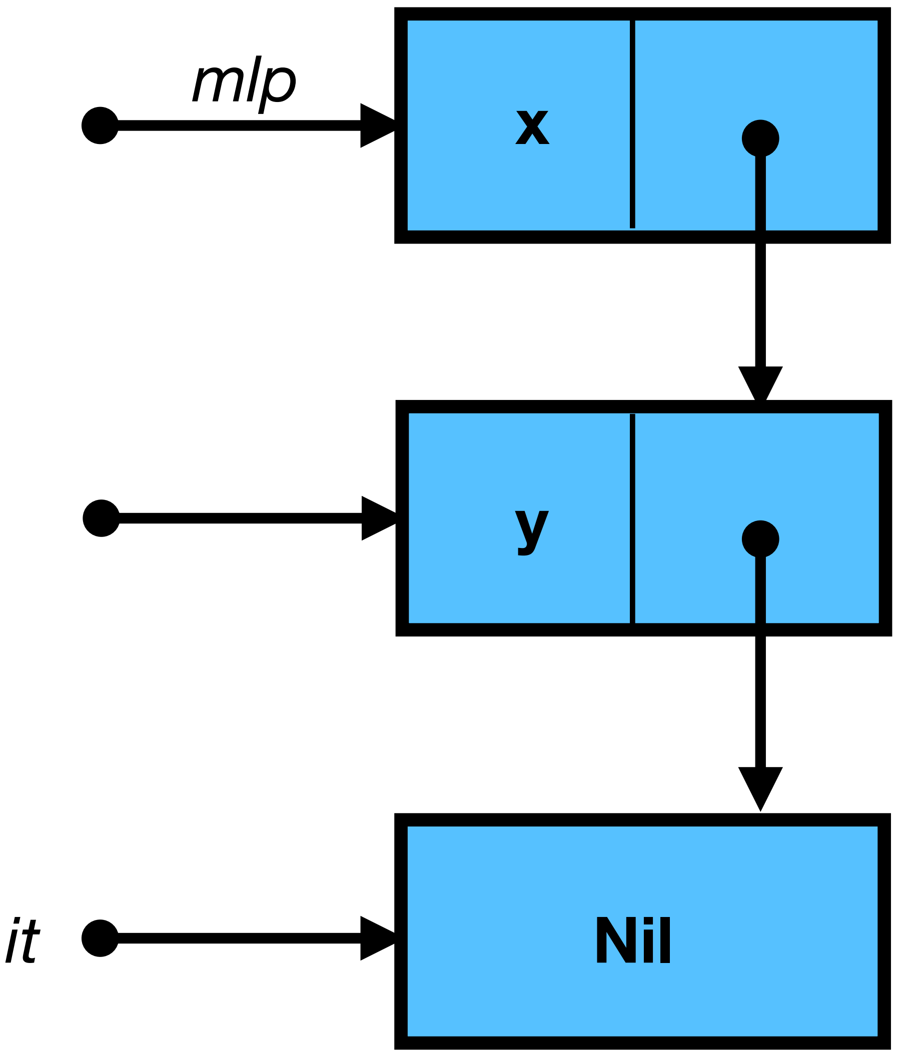

mlp
last
Nil

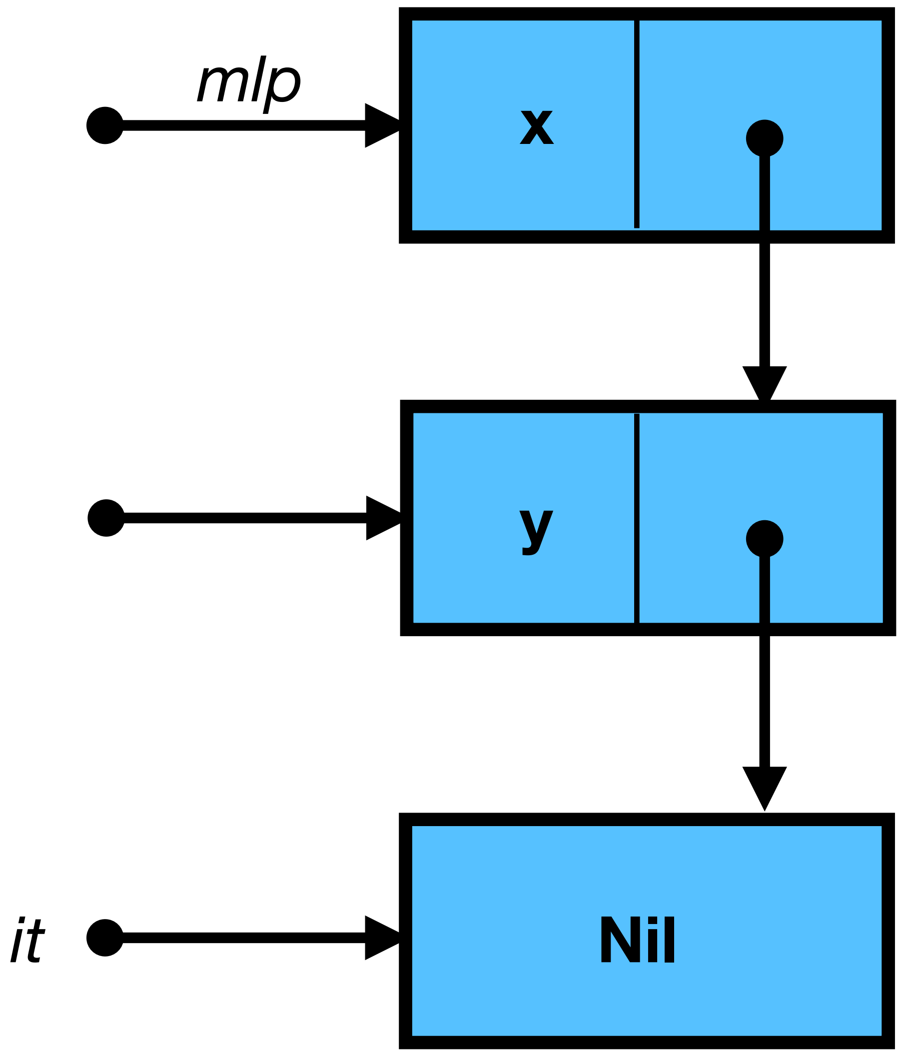

mlp
x
last
Nil


mlp
x
last
it
Nil


mlp
x
it
last
Nil


mlp
x
it
y
last
Nil


mlp
x
y
it
Nil



ref (Cons (x, ref (Cons (y, ref Nil))))

Destructive Concatenation

pointing to a 'box'   contents of a 'box'

```
# let rec joining mlp ml2 =
  match !mlp with
  | Nil -> mlp := ml2
  | Cons (_, mlp1) -> joining mlp1 ml2
val joining : 'a mlist ref * 'a mlist -> unit = <fun>

# let join ml1 ml2 =
  let mlp = ref ml1 in
  joining mlp ml2;
  !mlp
val join : 'a mlist -> 'a mlist -> 'a mlist = <fun>
```

Side-Effects

```
# let ml1 = mlistOf ["a"];;
val ml1 : string mlist = Cons ("a", {contents = Nil})
# let ml2 = mlistOf ["b";"c"];;
val ml2 : string mlist =
  Cons ("b", {contents = Cons ("c", {contents = Nil})})
# join ml1 ml2 ;;
```

What does this return?

Side-Effects

```
# let ml1 = mlistOf ["a"];;
val ml1 : string mlist = Cons ("a", {contents = Nil})
# let ml2 = mlistOf ["b";"c"];;
val ml2 : string mlist =
  Cons ("b", {contents = Cons ("c", {contents = Nil})})
# join ml1 ml2 ;;
```

What does this return?

```
- : string mlist =
Cons ("a",
 {contents = Cons ("b",
   {contents = Cons ("c", {contents = Nil})})})
```

Functional Programming

Let's Recap

Goals of Programming

- to **describe a computation** so that it can be done *mechanically*:
  - *expressions* compute *values*
  - *commands* cause *effects*
- to do so **efficiently and correctly**, giving right answers *quickly*
- to allow **easy modification** as our needs change
  - through an orderly *structure* based on *abstraction* principles
  - programmer should be able to predict effects of changes

Why Program in OCaml?

- It is **interactive**.
- It has a flexible notion of **data type**.
- It hides the underlying hardware: **no crashes**.
- Programs can easily be **understood mathematically**.
- It **distinguishes naming** from updating memory.
- It **manages storage** in memory for us.

Language

- Static type checking
- Parametric Polymorphism
- Type Inference
- Algebraic Data Types
- Pattern Matching
- First Class Functions

Abstraction

Language

- Static type checking
- Parametric Polymorphism
- Type Inference
- Algebraic Data Types
- Pattern Matching
- First Class Functions

```
# let x = "1" + 1 ;;
Error: This expression has type string but
an expression was expected of type int
```

Language

Static type checking

Parametric Polymorphism

Type Inference

Algebraic Data Types

Pattern Matching

First Class Functions

```
# let x = "1" + 1 ;;
Error: This expression has type string but
an expression was expected of type int
```

1A Object Oriented Programming

Dr Andrew Rice

Language

Static type checking

Parametric Polymorphism

Type Inference

Algebraic Data Types

Pattern Matching

First Class Functions

```
# type 'a tree =
    | Lf
    | Br of 'a * 'a tree * 'a tree
```

Language

Static type checking

Parametric Polymorphism

Type Inference

Algebraic Data Types

Pattern Matching

First Class Functions

```
# let fn l = List.map (fun (a,b) ->
        string_of_int a ^ b) l;;

val fn : (int * string) list -> string list
= <fun>
```

Language

Static type checking

Parametric Polymorphism

Type Inference

Algebraic Data Types

Pattern Matching

First Class Functions

1B Concepts in Programming Languages

1B Further Java

II Types

Language

Static type checking

Parametric Polymorphism

Type Inference

Algebraic Data Types

Pattern Matching

First Class Functions

```
# type vehicle =
    | Car of bool
    | Motorbike of int
    | Bicycle
```

Language

Static type checking

Parametric Polymorphism

Type Inference

Algebraic Data Types

Pattern Matching

First Class Functions

```
# type vehicle =
    | Car of bool
    | Motorbike of int
    | Bicycle

# match v with
  | Car false -> "car"
  | Car true  -> "reliant robin"
  ...
```

Language

Static type checking

Parametric Polymorphism

Type Inference

Algebraic Data Types

Pattern Matching

First Class Functions

1B Semantics of Programming Languages

```
# type vehicle =
    | Car of bool
    | Motorbike of int
    | Bicycle

# match v with
  | Car false -> "car"
  | Car true  -> "reliant robin"
  ...
```

Language

Static type checking

Parametric Polymorphism

Type Inference

Algebraic Data Types

Pattern Matching

First Class Functions

Abstraction

Runtime

Fast Foreign Functions

Static Linking

Garbage Collection

Fast Native Code

Multiarchitecture

Portable Bytecode

Execution

Runtime

Fast Foreign Functions

Static Linking

Garbage Collection

Fast Native Code

Multiarchitecture

Portable Bytecode

Upcoming Courses:

1A Operating Systems

1B Compiler Construction

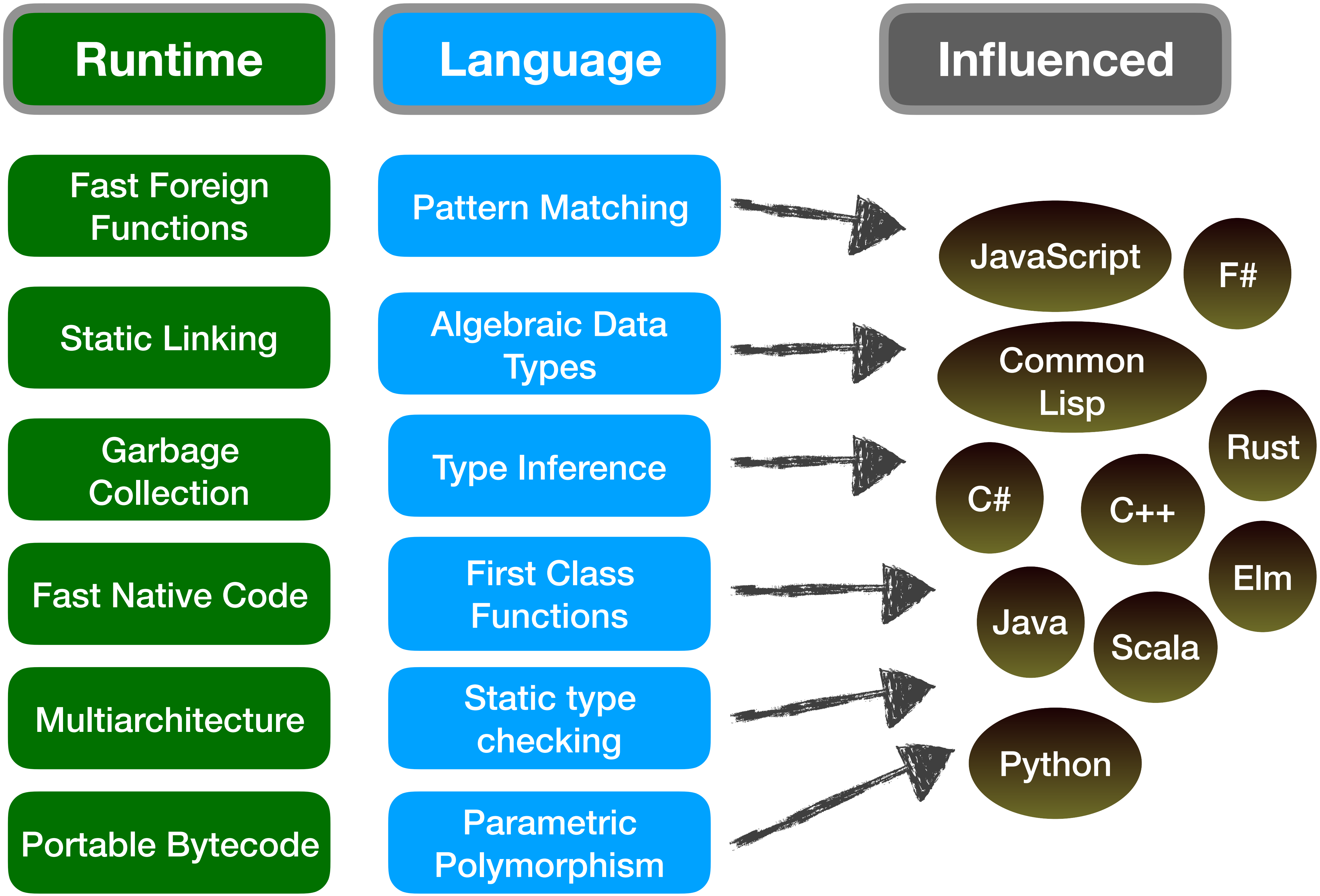
1B Programming in C/C++

1B Concurrent & Distributed Systems

OCaml: a system

Runtime	Language
Fast Foreign Functions	Pattern Matching
Static Linking	Algebraic Data Types
Garbage Collection	Type Inference
Fast Native Code	First Class Functions
Multiarchitecture	Static type checking
Portable Bytecode	Parametric Polymorphism

OCaml (& ML): Influences



OCaml: Applications

Runtime

- Fast Foreign Functions
- Static Linking
- Garbage Collection
- Fast Native Code
- Multiarchitecture
- Portable Bytecode

Language

- Pattern Matching
- Algebraic Data Types
- Type Inference
- First Class Functions
- Static type checking
- Parametric Polymorphism

Flexibility

- JavaScript
- Wasm
- FPGAs
- Microcontrollers
- Unix
- Mobile
- Unikernels
- Containers
- Proof Assistants
- Static Analysis

OCaml: Web Programming

Runtime	Language
Fast Foreign Functions	Pattern Matching
Static Linking	Algebraic Data Types
Garbage Collection	Type Inference
Fast Native Code	First Class Functions
Multiarchitecture	Static type checking
Portable Bytecode	Parametric Polymorphism

Flexibility

JavaScript

Wasm

REASON

https://reasonml.github.io

Reason lets you write simple, fast and quality type safe code while leveraging both the JavaScript & OCaml ecosystems.

OCaml: Building Hardware

Runtime | **Language** | **Flexibility**

OCaPIC: PIC microcontrollers programmed in OCaml

Static Linking

Algebraic Data Types

FPGAs

Microcontrollers

Garbage Collect

Type Inference

Fast Nativ

Multiarchit

Portable By

HardCaml is a structural hardware design DSL embedded in OCaml. The library can be used for front end design tasks up to the synthesis stage where a VHDL or Verilog netlist is generated. Libraries for fast simulation using LLVM, waveform viewing and co-simulation with Icarus Verilog are provided.

HardCaml-RiscV is a simple pipelined RV32I core, targetted towards a FPGA implementation and built with HardCaml.

OCaml: Operating Systems

Runtime

Language

Flexibility

MIRAGE OS Blog Docs API Canopy Community

A programming framework for building type-safe, modular systems

MirageOS is a library operating system that constructs unikernels for secure, high-performance network applications across a variety of cloud computing and mobile platforms. Code can be developed on a normal OS such as Linux or MacOS X, and then compiled into a fully-standalone, specialised unikernel that runs under a Xen or KVM

Recent Updates all

- *MirageOS running on the ESP32 embedded chip* (26 Jan 2018)
- *MirageOS Winter 2017 hack retreat roundup* (23 Dec 2017)

Collection

Fast Native Code

First Class Functions

docker

St

cl

Xen Project

Portable Bytecode

Parametric Polymorphism

Unix

Mobile

Unikernels

Containers

https://mirage.io

OCaml: Safety Critical

Runtime

Fast Foreign Functions

Static Linkin

Garbage Collection

Fast Native Co

Multiarchitecture

Portable Bytecode

Lang

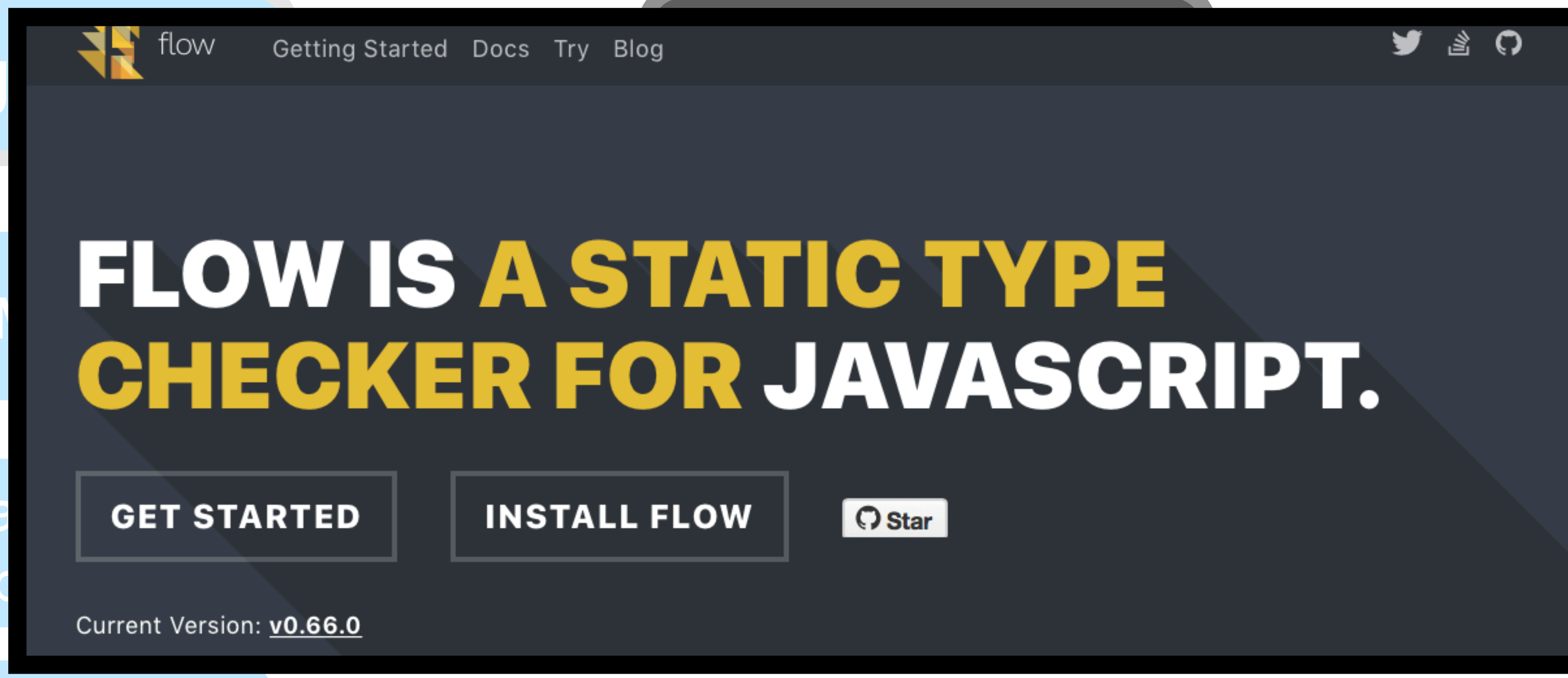
Pattern M

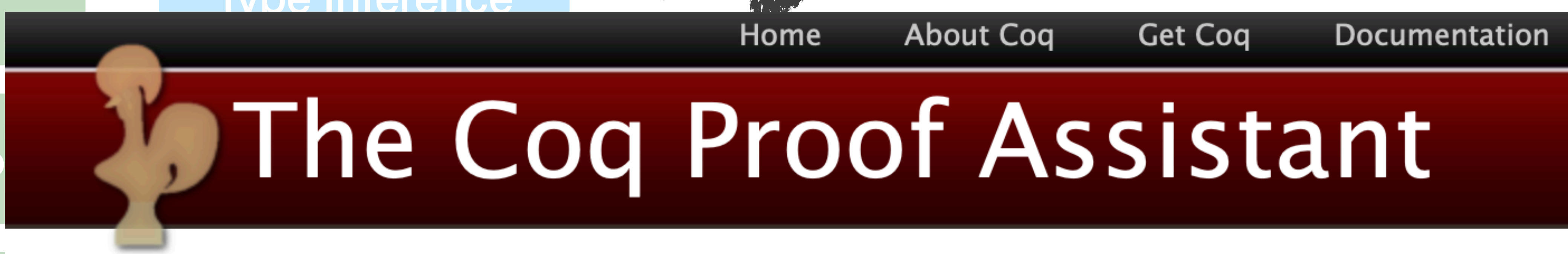
ebra Typ

Type Inference

Static type checking

Parametric Polymorphism





https://coq.inria.fr

Proof Assistants

Static Analysis

OCaml: Predictable Robots!

Ru

Fast
Fur

Stati

Ga
Co

Fast N

Creating safe robots with Imandra

Kostya Kanishev Follow
Jul 9, 2018 · 3 min read

From self-driving cars to medical surgeons, robots have become ubiquitous. Ensuring they operate safely and correctly is evermore important. The most popular middleware for robotics is the open-sourced Robot OS. We have begun work on developing an Imandra interface to Robot OS, opening up the world of robotics to the latest advancements in automated reasoning. In this post, we showcase our early results, discuss our roadmap and our submission for a talk at the upcoming ROSCon 2018 (Madrid, Spain).

Multiarchitecture

Portable Bytecode

Static type checking

Parametric Polymorphism

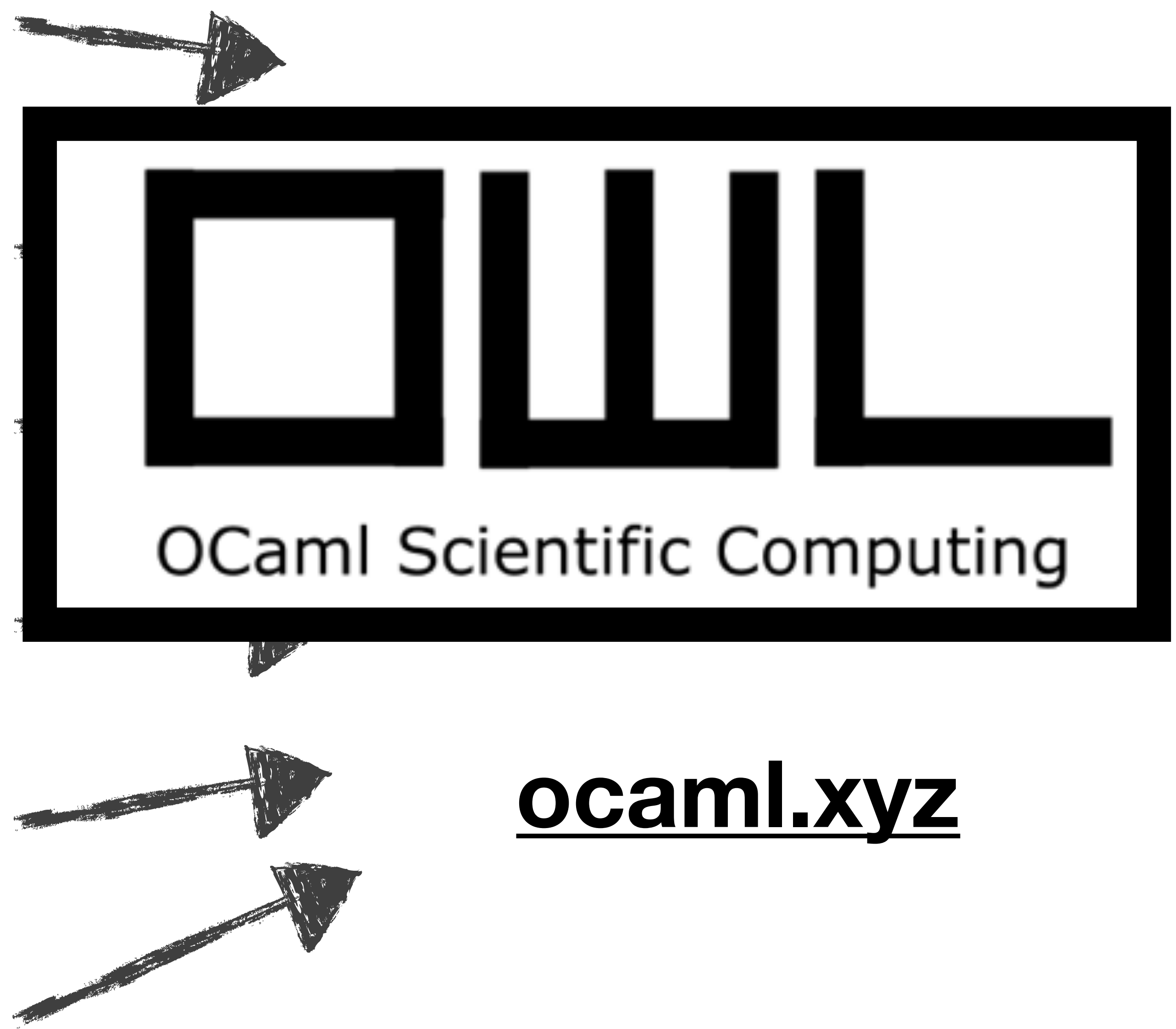
www.imandra.ai

Proof Assistants

Static Analysis

OCaml: Data Science

Runtime	Language	Flexibility
Fast Foreign Functions	Pattern Matching	
Static Linking	Algebraic Data Types	
Garbage Collection	Type Inference	
Fast Native Code	First Class Functions	
Multiarchitecture	Static type checking	
Portable Bytecode	Parametric Polymorphism	



ocaml.xyz

Goals of Programming

- to **describe a computation** so that it can be done *mechanically*:
  - *expressions* compute *values*
  - *commands* cause *effects*
- to do so **efficiently and correctly**, giving right answers *quickly*
- to allow **easy modification** as our needs change
  - through an orderly *structure* based on *abstraction* principles
  - programmer should be able to predict effects of changes