Who needs loops? Regular, tail and mutual recursion in OCaml

λ Discussion Group

Tazmilur Saad :: 29th August 2023

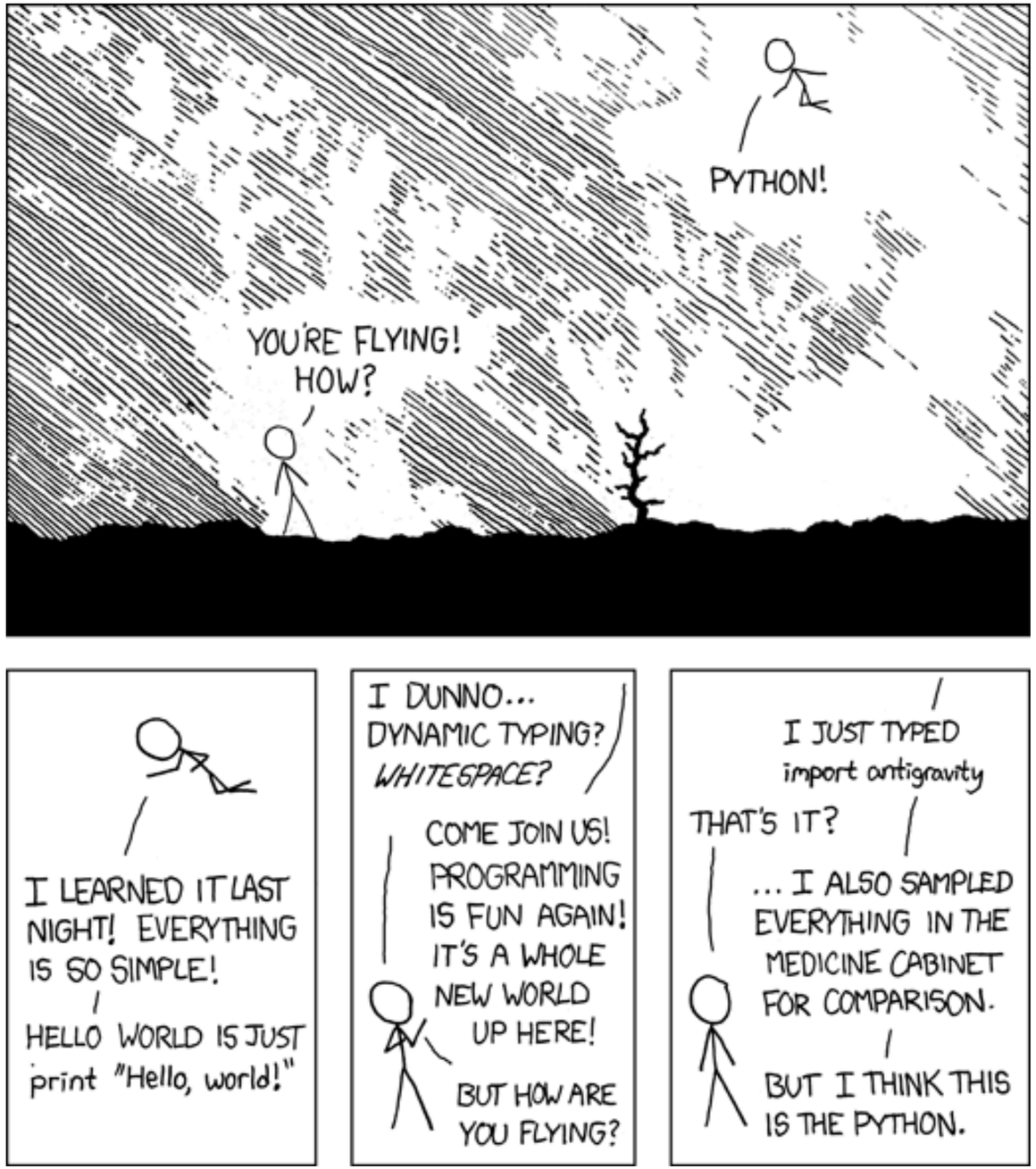
λ Discussion Group

λ Discussion Group

- Monthly show and tells about “functional” programming language features.
- Language agnostic. No experience required.
- A creative outlet for people to experiment with the weird and the academic and maybe learn something in the process!

Why should I learn another language?

- Because it’s fun!
- Because it’s a new perspective on the same problems.
- Because concepts from functional programming is useful even in imperative programs.



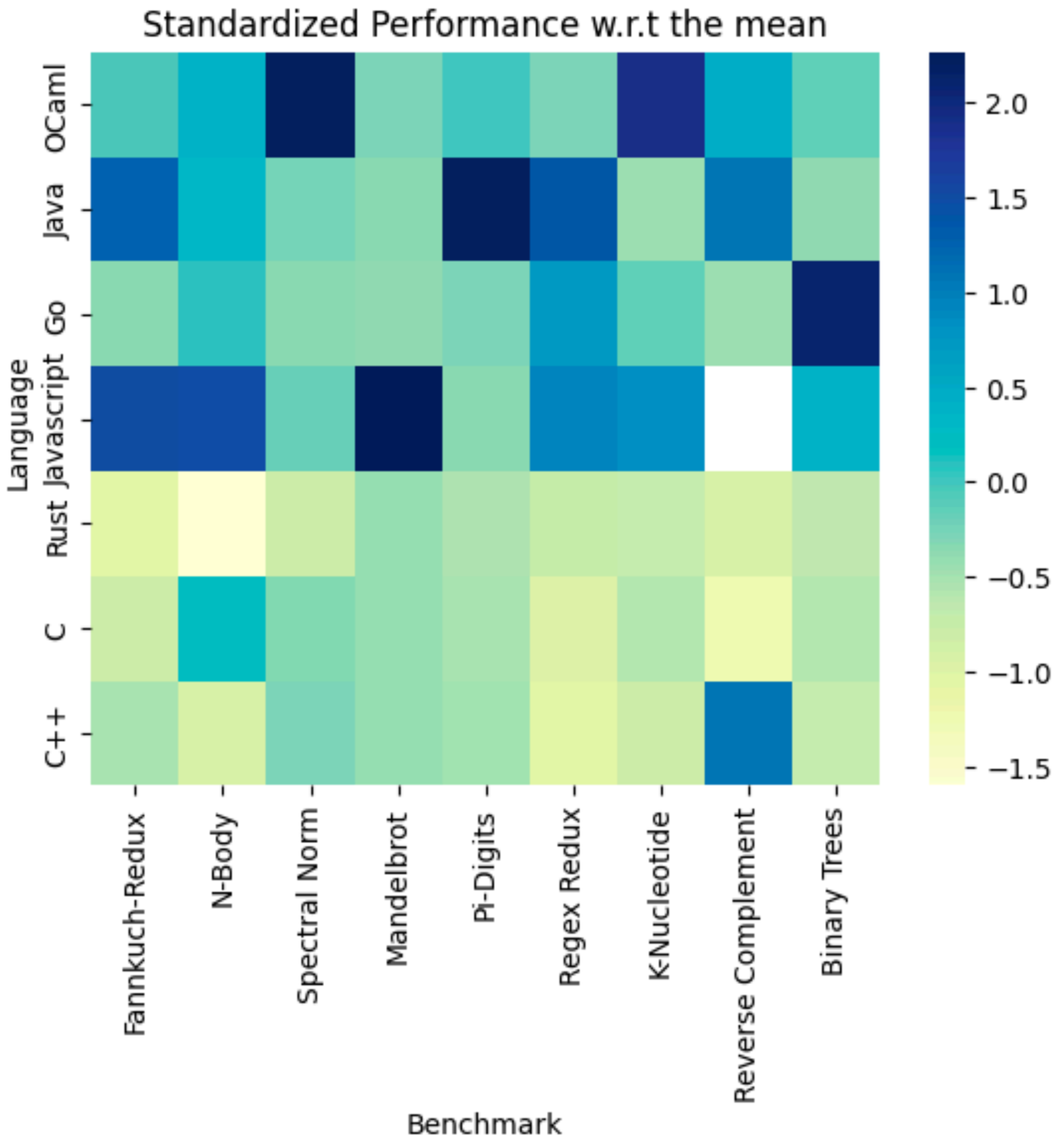
Aren’t functional languages super slow?

OCaml vs GC'd languages

	Fannkuch-Redux	N-Body	Spectral Norm	Mandelbrot	Pi Digits	Regex-Redux	K-Nucleotide	Reverse Complement	Binary Trees
OCaml	8.76	6.81	5.32	7.56	2.12	2.47	24.86	2.33	3.62
Java	10.71	6.77	1.54	4.11	7.44	5.40	5.03	3.02	2.52
Go	8.25	6.36	1.42	3.73	1.37	4.14	7.48	1.33	14.69
Python	285.2	360	78.36	155.28	1.39	8.02	44.13	3.72	34.53
JS (Node)	11.08	8.41	1.66	130.8	1.26	4.51	15.72	—	6.37

OCaml vs languages w/o GC

	Fannkuch-Redux	N-Body	Spectral Norm	Mandelbrot	Pi Digits	Regex-Redux	K-Nucleotide	Reverse Complement	Binary Trees
OCaml	8.76	6.81	5.32	7.56	2.12	2.47	24.86	2.33	3.62
Rust	7.21	3.92	0.72	1.01	0.71	1.34	2.88	0.80	1.19
C	7.53	6.61	1.44	1.63	0.81	0.94	3.77	0.48	1.57
C++	7.99	4.88	1.52	2.34	0.89	0.80	1.96	3.03	0.94


Standardized Performance w.r.t the mean
Language
OCaml
Java
Go
Javascript
Rust
C
C++
Fannkuch-Redux
N-Body
Spectral Norm
Mandelbrot
Pi-Digits
Regex Redux
K-Nucleotide
Reverse Complement
Binary Trees
Benchmark
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5

Do you write code like this?

```
do
{
    var current = next;
    var v0 = Sse2.Subtract(countVector, ramp);
    var bits = BitOperations.TrailingZeroCount(Sse2.MoveMask(v0));
    v0 = Vector128.Create((byte)bits);
    var v1 = Sse2.AndNot(Sse2.CompareGreaterThan(v0.AsSByte(), ramp.AsSByte()).AsByte(), countVector);
    countVector = Sse2.Subtract(v1, Sse2.CompareEqual(v0, ramp));
    next = Ssse3.Shuffle(next, masks_shift[bits]);
    var first = Sse2.ConvertToInt32(current.AsInt32());
    {
        var flips = 0;
        var v3 = Ssse3.Shuffle(current, c0);
        while ((first & 0xff) != 0)
        {
            v0 = Sse2.Subtract(v3, ramp);
            v3 = Ssse3.Shuffle(current, v3);
            v0 = Sse41.BlendVariable(v0, ramp, v0);
            current = Ssse3.Shuffle(current, v0);
            flips++;
            first = Sse2.ConvertToInt32(v3.AsInt32());
        }

        checksum += flips;
        if (flips > maxFlips) maxFlips = flips;
    }
```

Benchmarks

- Must keep confounding factors low. No option but numerical algorithms.
- Every language needs to implement algorithms the same way. Language idioms cannot be translated.
- Benchmarks are written by people who know the language in and out.
- So why not just try the language out?

Functions!

```
let mult a b = a * b

let rec fact n = match n with
  | 0 | 1 -> 1
  | _ -> n * fact (n - 1)
```

Functions!

```
let mult a b = a * b

let rec fact n = match n with
  | 0 | 1 -> 1
  | _ -> n * fact (n - 1)

val fact : int -> int = <fun>

val g : int -> int -> int = <fun>
```

Functions!

```
val mysteryf : ('a -> 'b) -> 'a list -> 'b list

val mysteryg : ('a -> bool) -> 'a list -> 'a list
```

Functions!

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

`map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.

```
val filter : ('a -> bool) -> 'a list -> 'a list
```

`filter f l` returns all the elements of the list `l` that satisfy the predicate `f`. The order of the elements in the input list is preserved.

Problems brewing…

```
let mult a b = a * b

let rec fact n = match n with
  | 0 | 1 -> 1
  | _ -> n * fact (n - 1)
```

Problems brewing…

```
fact 5
= 5 * fact 4
= 5 * (4 * fact 3)
= 5 * (4 * (3 * fact 2))
= 5 * (4 * (3 * (2 * fact 1)))
= 5 * (4 * (3 * (2 * 1)))
= 5 * (4 * (3 * 2))
= 5 * (4 * 6)
= 5 * 24
= 120
```

Tail Recursion

```
let rec fact n acc = match n with
  | 0 | 1 -> acc
  | _ -> fact (n - 1) (n * acc)
```

Tail Recursion

```
fact 5 1
= fact 4 5
= fact 3 20
= fact 2 60
= fact 1 120
= 120
```

- Tail call optimization can work if the caller does nothing with the callee's return value but return it.

Product Type

- A product type is just a struct: a collection of named data.
- In OCaml they are called records.

```
type person = {name : string; age : int}
```

Sum Types

- An ascended form of an enum.
- Also known as a variant, tagged union, discriminated union, disjoint union, etc.

```
type msg = Incr | Decr

let count = 0

let update c msg = match msg with
  | Incr -> c + 1
  | Decr -> c - 1
```

Sum Types

- More powerful than enums because they can contain data.
- And they can be recursive :)

```
type 'a list = Nil | Cons of 'a * 'a list

let rec length lst acc = match lst with
  | Nil -> acc
  | Cons (_, tl) -> length tl (acc + 1)

let rec sum lst acc = match lst with
  | Nil -> acc
  | Cons (v, tl) -> sum tl (acc + v)
```

Folds

```
let rec foldl f acc lst = match lst with
  | Nil -> acc
  | Cons (v, tl) -> foldl f (f v acc) tl

let sum lst = foldl ( + ) 0 lst

let length lst = foldl (fun _ acc -> acc + 1) 0 lst
```

```
type 'a t = 'a list =
| []
| (::) of 'a * 'a list
```

An alias for the type of lists.

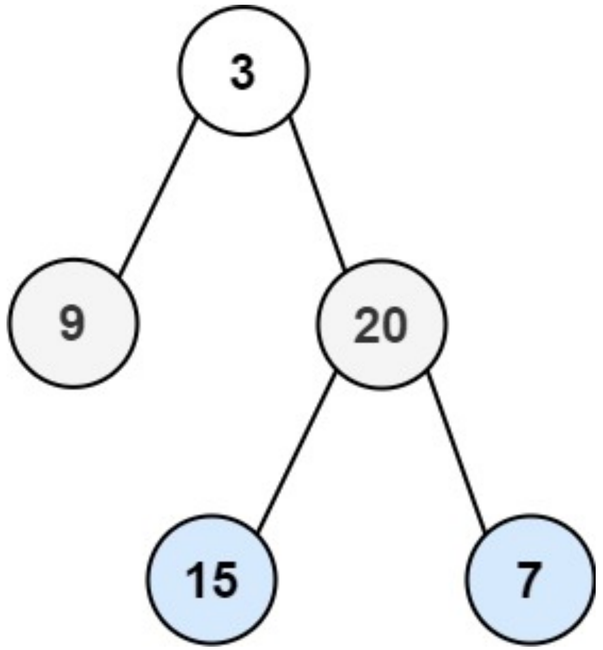
```
1 :: [] = [1]

1 :: 2 :: [] = [1;2]

1 :: [2; 3] = [1;2;3]

[1; 2] @ [3] = [1;2;3]
```

LeetCode #102. Given the root of a binary tree, return the level order traversal of its nodes' values.

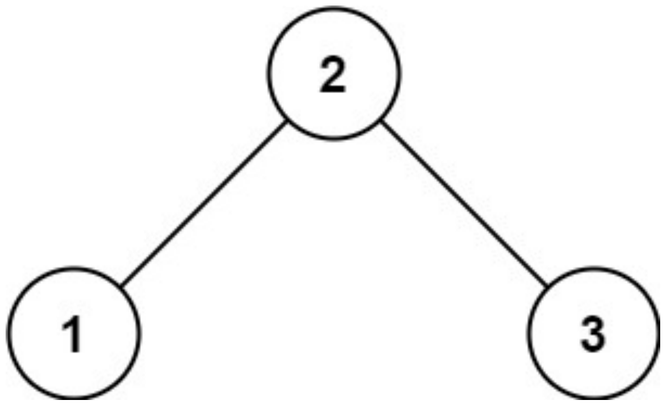



```
type 'a tree = Leaf | Tree of 'a * 'a tree * 'a tree

let rec fold_tree f acc = function
  | Leaf -> acc
  | Tree (v, l, r) -> f v (fold_tree f acc l)
                          (fold_tree f acc r)

let lorder t = fold_tree (fun v l r -> (v :: l) @ r) [] t
```

LeetCode #98. Given the root of a binary tree, determine if it is a valid binary search tree (BST).




```
open List

let sorted lst = combine lst (tl lst)
                 |> for_all (fun (l, r) -> l < r)

let valid_bst t =
  fold_tree (fun v l r -> l @ (v :: r)) [] t |> sorted
```

Algebraic Datatypes

- They are called algebraic datatypes because you can combine sum and product types together.

```
type loc = NYK | Whippany
type employeee = {
    name : string;
    location : loc
    age : int;
}
```

Golang's Request Type

```
type Request struct {
    Method string
    URL *url.URL
    Proto string
    ProtoMajor int
    ProtoMinor int
    Header Header
    Body io.ReadCloser
    GetBody func() (io.ReadCloser, error)
    ContentLength int64
    TransferEncoding []string
    Host string
    Form url.Values
    PostForm url.Values
    ...
}
```

OCaml Request Type

```
type request = {
    m : string;
    url : url;
    proto : string;
    protoMajor: int;
    protoMinor : int;
    header : int;
    body : string option;
    getBody : unit -> (string * url) option;
    contentLength : int;
    transferEncoding : string list;
    host : string;
    form : url option;
    postForm : url option;
}
```

Golang Request Type

```
// URL specifies either the URI being requested (for server
// requests) or the URL to access (for client requests).
//
// For server requests, the URL is parsed from the URI
// supplied on the Request-Line as stored in RequestURI.  For
// most requests, fields other than Path and RawQuery will be
// empty. (See RFC 7230, Section 5.3)
//
// For client requests, the URL's Host specifies the server to
// connect to, while the Request's Host field optionally
// specifies the Host header value to send in the HTTP
// request.
URL *url.URL
```

Golang Request Type

```
// The protocol version for incoming server requests.
//
// For client requests, these fields are ignored. The HTTP
// client code always uses either HTTP/1.1 or HTTP/2.
// See the docs on Transport for details.
Proto      string // "HTTP/1.0"
ProtoMajor int    // 1
ProtoMinor int    // 0
```

Golang Request Type

```
// GetBody defines an optional func to return a new copy of
// Body. It is used for client requests when a redirect requires
// reading the body more than once. Use of GetBody still
// requires setting Body.
//
// For server requests, it is unused.
GetBody func() (io.ReadCloser, error)
```

Golang Request Type

```
// ... if this is a server req
r := http.Request{getBody: someFunc}
```

- What are the consequences in the best and worst case?

Golang Request Type

```
// ... if this is a server req
r := http.Request{getBody: someFunc}
```

- What are the consequences in the best and worst case?
- If we perform a straightforward translation from Go's type system we haven't achieved much.

Golang Request Type

- Every request can be categorized as either a server or client request.
- Some fields can only exist in one state or the other.
- Some fields are used in both states.

OCaml's Request Type

```
type protocol = HTTP1 | HTTP1'1 | HTTP2

let major = function
  | HTTP1 | HTTP1'1 -> 1
  | HTTP2 -> 2

let minor = function
  | HTTP1 | HTTP2 -> 0
  | HTTP1'1 -> 1
```

OCaml's Request Type

```
type result = Error of string | Success
type serverconn = { url : string; protocol : protocol;}
type clientconn = { url : string * string option;
        getBody : unit -> ((in_channel -> string option) * result)}
type connection = Server of serverconn | Client of clientconn
```

OCaml's Request Type

```
type request = {
  conntype : connection;
  transferEncoding: string list;
  host : string;
  body: string option;
}
```

Make illegal states impossible to represent.

Modules + Signatures

- A module is a structure that consists of type definitions and functions.
- Every file is automatically compiled to a module.
- Module signatures define a blueprint for a module to match.
- Similar to golang interfaces / Rust traits / Haskell typeclasses.

Modules + Signatures

- We can define a distance measure between two arbitrary types.
- The hamming distance is the number of edits needed to convert one sequence to another.
- If the hamming distance between two instances of the same type can be calculated, we'll call such a type Hammable.

Modules + Signatures

```
module type Hammable = sig
  type t
  val hdist : t -> t -> int list
end

(* hdist "abc" "abb" = 001 *)
```

Modules + Signatures

```
module S : (Hammable with type t = string) = struct
  type t = string
  let hdist a b =
    let x = String.to_seq a in
    let y = String.to_seq b in
    Seq.zip x y
    |> List.of_seq
    |> List.map (fun (l, r) -> if l = r then 0 else 1)
end
```

Locality Sensitive Hashing

- Generally, we want our hash functions to minimize collisions.
- If two instances of the same type are within some d distance of each other, then they should hash to the same bucket.
- Hash based search that is not that sensitive to small differences in the type :)

Locality Sensitive Hashing

- One simple implementation of LSH is to take the Hamming distance and generate a series of functions that randomly pick an element of the distance vector.
- $\mathcal{F} = \{h : \{0,1\}^d \to \{0,1\} \mid h(x) = x_i \text{ for some } i \in \{1,\ldots,d\}\}$
- The vector of functions together can be used to define nearest neighbor search.
- We already have a Hammable module signature, can we automatically generate an LSHable module?

Functors

- Functions that take modules to modules.
- Use the definition of one interface to define another interface.
- For example — if you have one module that defines a Hashable signature for some type t you can use it to create a Hashmap module constrained on t.

Functors

```
module LSH (M : Hammable) = struct

  let () = Random.init 1

  let family (a : M.t) (b : M.t) : (unit -> int) list =
    let d = M.hdist a b in
    let l = List.length d in
    List.init l (fun _ -> fun () -> List.nth d (Random.int l))

  let hash (a : M.t) (b : M.t) : int list =
    family a b |> List.map (fun x -> x ())
end
```

What is a functional language?

- Functions are convenient. They are easy to create, can be used as values, are implemented efficiently, can be polymorphic, can be stored in containers like lists or maps, can capture their environment, etc.

```
func mult(a, b int) int {
    return a * b
}

func pmult(a int) func(int) int {
    return func(b int) int {
        return a * b
    }
}

f := pmult(10)
r := f(10) // 100
```

```
let mult a b = a * b in
let pmult a = mult a in
(pmult 10) 10
```

What is a functional language?

- Functions are the primary form of abstraction. Streams, generators, coroutines, effects, states, etc are implemented as composition of functions on simple data types.

```python
def count(start):
    n = start
    while True:
        yield n
        n += 1
```

```fsharp
let rec count n =
  Seq.Cons(n, fun () -> count (n + 1))

(*
type 'a node =
    | Nil
    | Cons of 'a * (unit -> 'a node)
*)
```

What is a functional language?

- Enforce a “pure” style of programming + immutability.
- A “pure" function returns the same output for the same input. If your function depends on some (external) state, then at the very least this state should be encoded as part of your function.

```
import “math/rand”

rand.Seed(1)

func f() int {
    return rand.Intn(10)
}

// sometime later

func g() int {
    return rand.Intn(100)
}
```

```
import “math/rand"

p := rand.New(rand.NewSource(1))
q := rand.New(rand.NewSource(2))

func f(r *rand.Rand) int {
    return r.Intn(10)
}

// sometime later

func g(r *rand.Rand) int {
    return r.Intn(100)
}
```

```
open Random.State

let p = make [|1|]
let q = make [|2|]

let f s = int s 10
let g s = int s 100

(* let f (s : State.t) = int s 10 *)
```

Purity

- There is a bug here! In some ideal world the output might be the same :)

```
def f(a = []):
    a += [1]
    return a

one = f()
two = f()

print(one)
print(two)
```

Purity

- There is a bug here! In some ideal world the output might be the same :)
- The default variable points to the same place in memory and persists between calls.

```
def f(a = []):
    a += [1]
    return a

one = f()
two = f()

print(one)
print(two)
```

What about data structures?

- How do we use a map if we cannot modify it?
- Updating a map returns a new map with the updated binding instead of mutating the existing map!

```
val add : key -> 'a -> 'a t -> 'a t
```

`add key data m` returns a map containing the same bindings as `m`, plus a binding of `key` to `data`. If `key` was already bound in `m` to a value that is physically equal to `data`, `m` is returned unchanged (the result of the function is then physically equal to `m`). Otherwise, the previous binding of `key` in `m` disappears.

What about data structures?

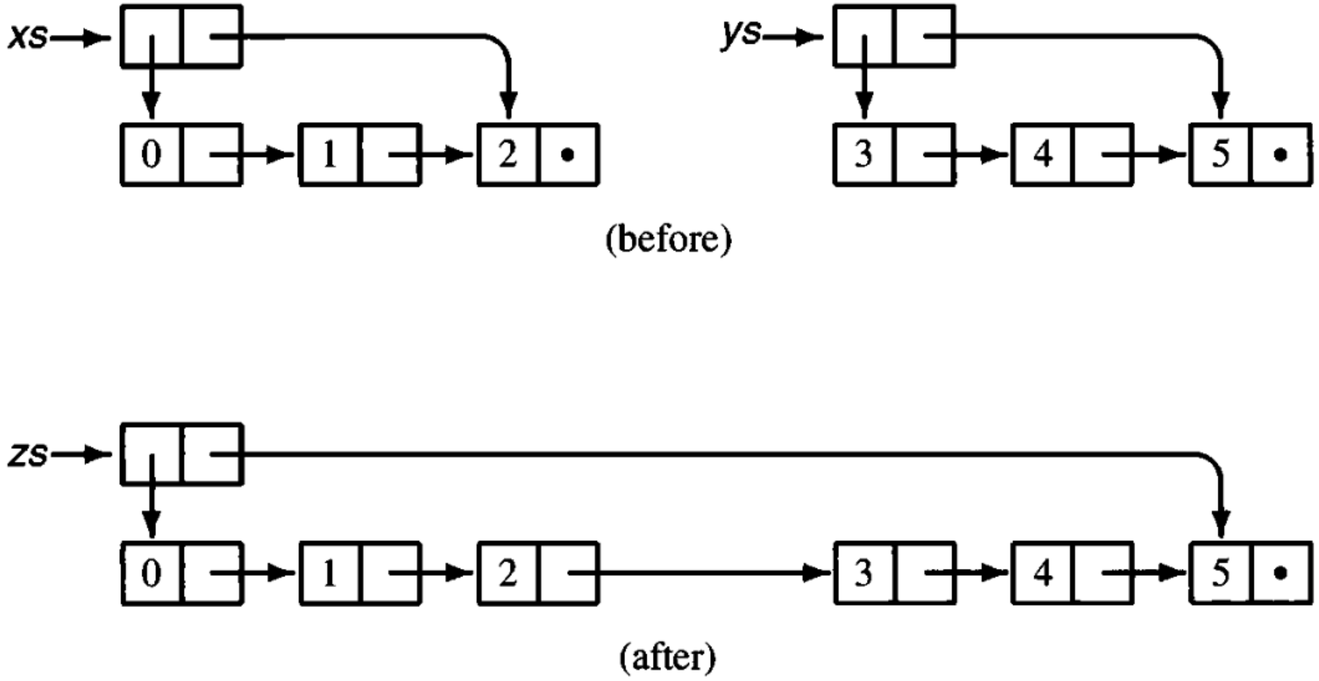


Figure 2.4. Executing *zs* = *xs* ++ *ys* in an imperative setting. Note that this operation destroys the argument lists, *xs* and *ys*.

What about data structures?

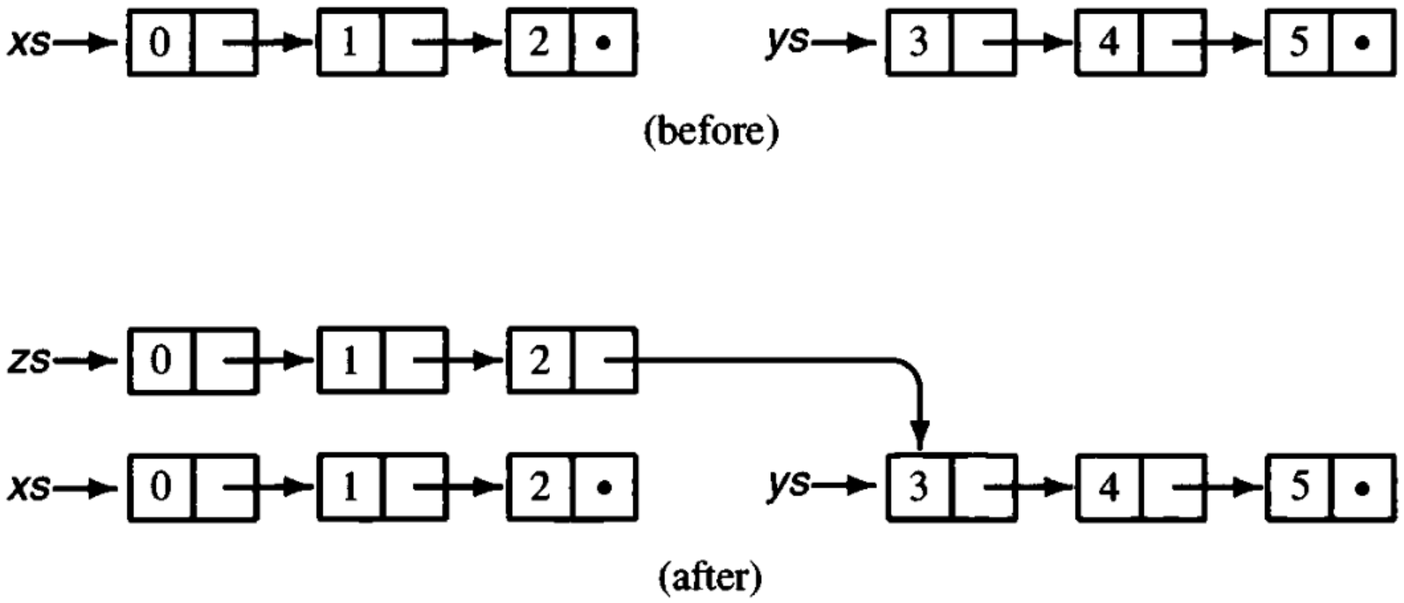


Figure 2.5. Executing *zs* = *xs* ++ *ys* in a functional setting. Notice that the argument lists, *xs* and *ys*, are unaffected by the operation.

What about data structures?

- How do we use a map if we cannot modify it?
- Updating a map returns a new map with the updated binding instead of mutating the existing map!
- Compiler optimizations such as path copying and structural sharing tries to reduce memory usage as much as possible.
- If everything is mutable and you have to defensively copy everything anyways then you are just using a worse version of immutability :)

What is a functional language?

- Have a strong and static type system and utilize type driven development. Lisp and its derivatives are generally the only exception.
- Type inference makes your programs look like scripts and the soundness of the system means you avoid type errors without the boilerplate and verbosity.
- Type driven development with pattern matching ensures that the compiler can rule out as many possible bugs as possible.

```
type msg = Get of int | Post of int * int | Delete of int
type resp = Success | Error of string

let update msg = match msg with
  | Post (k, v) -> (* call post handler *)
  | Delete k -> (* call delete handler *)
  | Get _ -> Error "invalid path"
```

```
type msg = Get of int | Post of int * int | Delete of int
type resp = Success | Error of string

let update msg = match msg with
  | Post (k, v) -> (* call post handler *)
  | Delete k -> (* call delete handler *)
  | Get _ -> Error "invalid path"
```

```
type msg = Get of int | Post of int * int | Delete of int
            Update of int * int
type resp = Success | Error of string

let update msg = match msg with
  | Post (k, v) -> (* call post handler *)
  | Delete k -> (* call delete handler *)
  | Get _ -> Error "invalid path"

(* compile error *)
```

```
func f(i int) int {
    res := g(h(i))
    return res
}


// sometime later

func f(i int) int {
    var res int

    if i > 1 {
        res = g(h(i))
    }

    // mistake!

    return res
}
```

```
let f i = g (h i)


let f i = if i > 1 then g (h i)
(* does not compile *)

let f i = if i > 1 then
  Some (g (h i)) else None
(* forces you to fix everything *)
```

Tests are not equal to proofs!

- At best tests are a probabilistic measure of how correct your program is.
- Enumerating every natural number ad infinitum will never show for sure that there are infinitely many natural numbers, but a simple proof by contradiction will work.

Tests are not equal to proofs!

- Type theory came about as an alternative to logic and set theory.
- Types are isomorphic to logical propositions. The rules for typing correspond to rules for proving the proposition.
- Evaluation corresponds to simplification. Programs are isomorphic to proofs!
- Let the type checker do its magic :)

Next Month

- WebAssembly Text Format stack machine using monadic parser combinators and mutually recursive functions.

```
(func (param i32) (param f32) (local f64)
  local.get 0
  local.get 1
  local.get 2)
```

Thanks :)