# Defunctionalization

Li-yao Xia - PLClub - UPenn - April 24, 2020

# Defunctionalization

- Often viewed as a (subpar) compilation technique
- But quietly pulls its weight in day-to-day programming
- A general tool for understanding programs

# What is defunctionalization?

# General idea

Rewrite a program to remove higher-order functions.

Program

```
... (\x -> isRed x) ...

... (\x -> isYellow x) ...

... (\x -> x == y) ...

... (\x -> p x && q x) ...

... f z ...
```

# Famous saying

"All problems in computer science
can be solved by
another level of indirection.”

Replace all lambdas with fresh **symbols**.
Replace all function applications with calls to "**`apply`**".

Program

```
... (\x -> isRed x) ...

... (\x -> isYellow x) ...

... (\x -> x == y) ...

... (\x -> p x && q x) ...

... f z ...
```

Defunctionalized program, part 1

```
... IsRed ...

... IsYellow ...

... (Equals y) ...

... (And p q) ...

... apply(fsym, z) ...
```

# Free variables in lambdas get captured in the corresponding **symbol**.

Program

```
... (\x -> isRed x) ...

... (\x -> isYellow x) ...

... (\x -> x == y) ...

... (\x -> p x && q x) ...

... f z ...
```

Defunctionalized program, part 1

```
... IsRed ...

... IsYellow ...

... (Equals y) ...

... (And p q) ...

... apply(fsym, z) ...
```

The **symbols** are constructors of a *data type*.
**`apply`** is a *first-order function* defined by pattern-matching.

Program

```
... (\x -> isRed x) ...

... (\x -> isYellow x) ...

... (\x -> x == y) ...

... (\x -> p x && q x) ...

... f z ...
```

Defunctionalized program, part 2

```
data Fun
  = IsRed
  | IsYellow
  | Equals Color
  | And Fun Fun

apply(IsRed, x) = isRed x
...
apply(Equals c, d) = (c == d)
...
```

# Where does it come from?

# A bit of history

"In this paper, we will describe and classify several varieties of [definitional] interpreters."

John C. Reynolds,
in Definitional interpreters for higher-order languages (1972)

Already presented as a programming technique rather than a compilation technique (even though the two views are closely related).

# Defunctionalization for compilation

- Often overshadowed by **closure conversion**.

**Defunctionalized** functions
"are equivalent to (...) ***closures***."
John C. Reynolds (1972)

The same, but different?

# ???

# Defunctionalization vs closures

```
... (\x -> x == y) ...
```

```
... (Equals y) ...
```

A **constructor**, i.e., a "tag"

```
... (*, y) ...
```

A **pointer** to a closed function

Value(s) of the free variable(s) in the original lambda (now parameters of **`apply`**/the closed function)

```
apply(Equals y, x) = (x == y)
```

```
globalfun(y, x) = (x == y)
```

# Defunctionalization vs closures

`a -> b`

## Algebraic data type

```
data Fun
  = IsRed
  | IsYellow
  | Equals Color
  | And Fun Fun
```

Much more **Fun** to program with!

## Existential type

```
∃e. ((e × a) #-># b) × e
```

Only “global functions” (just a **pointer**).

# For compilers...

- **`apply`** adds an unnecessary level of indirection
- Defun. enumerates all lambdas
  → full program compilation, lack of compositionality
- But pointer + existential takes away all the ***Fun***!

And actually not true!
(Hint: ML modules)

*Defunctionalization as modular closure conversion*, Ulrich Schöpp, PPDP 2017

# For programmers...

- Data types + functions = programming 101
- Easy to readapt: customize **`apply`**, use multiple data types, e.g., distinguish by function type.

Title of a Compose 2019 talk, by James Koppel. Check it out!

# The best refactoring you've never heard of

- Summary: Replace functions with a concrete representation + interpreter (`apply`).
- Easy to do by hand, and to readapt!

Benefits include:

- **Serializability** (store and send functions!)
- **Performance** (fancy recursive algorithms (a priori slow) = fast state machines)

Slogan from the same Compose 2019 talk, by James Koppel

# Defunctionalize the continuation!

```
sum [] = 0
sum (x : xs)
  = x + sum xs
```

CPS →

```
sum' []       k = k 0
sum' (x : xs) k
  = sum' xs (\y -> k (x + y))
```

k is always `(\y -> x1 + (x2 + ... + y))`
`(\y -> acc + y)`

```
sum (1 : 2 : 3 : ...)
  = 1 + (2 + (3 + sum ...))
```

Linear space

↓ Defunctionalize

```
sum'' []       acc = acc
sum'' (x : xs) acc
  = sum'' xs (x + acc)
```

```
sum'' (1 : 2 : 3 : ...) 0
  = sum'' ... 6
```

Constant space

# Functional Programming in an Emergency

**Emergency** [*noun*]
Situation where a functional programming language is not used

# Emergency Functional Programming

How to solve any {PROBLEM}:

- “{PROBLEM} can be solved with higher-order functions…”
- “… but I’m using **{BAD_PL}**.”
- **✱** *Puts on **Defunctionalization** goggles* **✱**
- “Oh, **{BAD_PL}** has higher-order functions,
  it’s almost a **good PL**.”

# Three examples of {BAD_PL}

1. OCaml
2. Haskell
3. Coq

# 1. {BAD_PL} = OCaml

# {BAD_PL} = OCaml

## *The Monad Problem*: not (*quite*) having monads.[1]

```haskell
-- Haskell
return :: Monad m => a -> m a
```

**One overloaded operation for all monads:** some operations can be defined for all monads, once and for all.

```ocaml
(* OCaml *)
val async_return : ‘a -> ‘a async_m
val    qc_return : ‘a -> ‘a    qc_m
val   etc_return : ‘a -> ‘a   etc_m
```

**Can’t generalize over monads *m*.**

[1] This might be an unfair exaggeration for comedic purposes.

# No higher-kinded types in OCaml

In OCaml, type variables (`'a`, `'b`, …) only range over ***types***…

```
'a list
```

… not ***type constructors*** ("type → type"; `list`, `option`, `_ * _`).

```
'a 'm          (* nonsense! *)
```

*See also: every PL more popular than Haskell.*

# ~~No~~ higher-kinded types in OCaml

In case of emergency, use **defunctionalization**!

Defunctionalize type constructors:

`m a` will be denoted by **`apply`**`(msym, a)`

`'a 'm` will be denoted by `('a, 'msym)` **`apply`**

*Lightweight higher-kinded polymorphism*, Jeremy Yallop, Leo White, FLP 2014

# ~~No~~ higher-kinded types in OCaml

```
(* Polymorphic return in OCaml *)
val return : ‘msym monad -> ‘a -> (‘a,’msym) apply

(*  return ::     Monad m =>  a -> m a   -- in Haskell *)
```

Some manual conversions are required, but at least it works:

```
val   wrap_list : ‘a list -> (‘a, listsym) apply
val unwrap_list : (‘a, listsym) apply -> ‘a list
```

*“Oh, **OCaml** has higher-kinded types, it’s almost a **good PL**.”*

# 2. {BAD_PL} = Haskell

# {BAD_PL} = Haskell

*"Haskell can't be **that** bad. It even has type families!"*

```
type family Map (f :: a -> b) (xs :: [a]) :: [b] where
  Map f [] = []
  Map f (x : xs) = f x : Map f xs
-- This is valid Haskell.
```

# No higher-order type families in Haskell

*“Wait a second…”*

```
Map :: (a -> b) -> [a] -> [b]  -- Looks pretty H-O...?
```

Try this:

```
type family Snd (xy :: (a, b)) :: b where
  Snd (x, y) = y
```

```
ghci> :kind! Map Snd [(1,”One”), (2, “Two”)]
<A WILD TYPE ERROR APPEARS>
```

# No higher-order type families in Haskell

```
type family Map (f :: a -> b) (xs :: [a]) :: [b] where
```

Only **type constructors** (`Maybe`, `[ ]`), not the same as **type families** (`Map`, `Snd`)..

```
type family Snd (xy :: (a, b)) :: b where
```

Key distinction: type families **cannot be** partially applied (always "`Snd something`")

```
Map Snd ...   -- Illegal
```

This limitation might disappear in the near future:
*Higher-order type-level programming in Haskell*, Csongor Kiss et al, ICFP 2019.

# ~~No~~ higher-order type families in Haskell

In case of emergency, use **defunctionalization**!

```
type a ~> b  -- Defunctionalized type families
type family Apply (fsym :: a ~> b) (x :: a) :: b


type family Map (fsym :: a ~> b) (xs :: [a]) :: [b] where
  Map fsym [] = []
  Map fsym (x : xs) = Apply fsym x : Map fsym xs
```

*Promoting functions to type families in Haskell*,
Richard A. Eisenberg, Jan Stolarek, Haskell Symposium 2014

# ~~No~~ higher-order type families in Haskell

```
data SndSym :: (a, b) ~> b  -- Defunctionalized!
type instance Apply SndSym (x, y) = y
```

```
ghci> :kind! Map SndSym [(1,"One"), (2, "Two")]
["One", "Two"] :: [Symbol]
```

Note: `Symbol` is the kind of type-level strings in Haskell (has nothing to do with defun. symbols).

*"Oh, **Haskell** has higher-order type families, it's almost a **good PL**."*

# 3. {BAD_PL} = Coq

# {BAD_PL} = Coq

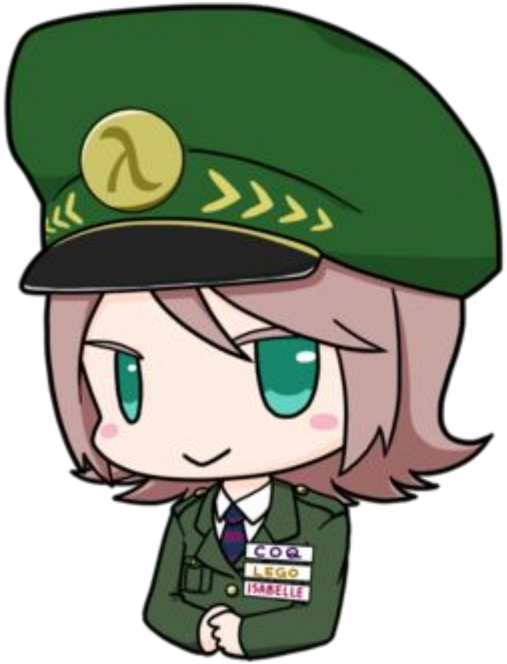
Coq is a total language: all functions terminate.

Restrictions on recursive definitions.

**Cofixpoints** must be **productive**.

```
CoFixpoint ones := Cons 1 ones.
CoFixpoint bad := bad.   (* Rejected *)
```

# No general recursion in a total language




We have **Proof General**,
but this has nothing to do with the topic.

# No general recursion in a total language

```
(* Fixed point of a function f. Solve for nu:  nu = f nu *)
(* e.g., f ones := Cons 1 ones  leads to  nu = ones  from the pv. slide *)
CoFixpoint mfix (f : Stream a -> Stream a) : Stream a :=
  f (mfix f).  (* Rejected *)
```

f might inspect the very stream we are in the middle of constructing!

The expression `mfix f`, although it has type `Stream a`, must be used according to very restrictive rules: it is **not** truly **a first-class value**.

# ~~No~~ general recursion in a total language

In case of emergency, use **defunctionalization**!

```
(* Defunctionalized Streams (instead of functions) *)
Inductive StrSym a : Type := NuSym | ...
```

Defun-zed (`mfix f`)

```
Definition apply : StrSym a -> Stream a -> Stream a := ...

CoFixpoint mfix (f : StrSym a -> StrSym a) : Stream a :=
  apply (f NuSym) (mfix f).
```

`apply` guaranteed to not inspect `mfix f`,
just places it wherever `apply` finds `NuSym`

*"Oh, **Coq** has general recursion, it's almost a **good PL**."*

The details get hairy very quickly; see references in last slide and presenter notes for more.

# Defunctionalization

1. *Higher-kinded types for* ***OCaml***
2. *Higher-order type families for* ***Haskell***
3. *General recursion for* ***Coq***

# References (Defun. and closures)

The appearance of defunctionalization:
- *Definitional interpreters for higher-order programming languages*, John C. Reynolds, ACM 1972 (this link is actually a reprint of the original version)

Among the earliest references (if not *the* one) for closures and closure conversion:
- *The mechanical evaluation of expressions*, Peter Landin, The Computer Journal, 1964

Closure conversion + existential types:
- *Typed closure conversion*, Minamide, Morrisett, Harper, POPL 1996.

For the claim that defunctionalization can be made modular, using ML modules:
- *Defunctionalization as modular closure conversion*, Ulrich Schöpp, PPDP 2017, (<- in this and related papers: game semantics!)
I heard of this thanks to an online comment by Neel Krishnaswamy on a SIGPLAN blogpost by James Koppel (see next slide)

# References (Defun. in practice)

A very fun introduction to defunctionalization:
- *The best refactoring you've never heard of*, James Koppel, Compose 2019 talk & transcript
- *Defunctionalization: everybody does it, nobody talks about it*, James Koppel, SIGPLAN blog, 2019 (condensed version of the talk)

- *Defunctionalization at work*, Olivier Danvy, Lasse R. Nielsen, extended version of a PPDP 2001 paper.
- *Refunctionalization at work* (2009) also seems like a good follow-up

cf. title slide near the middle

# References ("Defun. in an Emergency of {BAD_PL}")

OCaml, encoding higher-kinded types:


- *Lightweight higher-kinded polymorphism*, Jeremy Yallop, Leo White, FLP 2014


Haskell, encoding higher-order type families (or actually adding them to the language, see ICFP 2019):


- *Higher-order type-level programming in Haskell*, Csongor Kiss et al, ICFP 2019
- *Promoting functions to type families in Haskell*, Richard A. Eisenberg, Jan Stolarek, Haskell Symposium 2014
- *Defunctionalization for the win*, Richard A. Eisenberg, blog


Coq (and siblings), encoding general recursion:


- *Turing Completeness Totally Free*, Conor McBride, unpublished manuscript (?)
- *Compositional coinductive recursion in Coq*, Gregory Malecha, blog
- *Interaction trees*, Li-yao Xia et al., POPL 2020