Functional Programming

Dr. Liam O'Connor
University of Edinburgh LFCS
UNSW, Term 3 2020

Functional Programming

Many languages have been called functional over the years:

```
Lisp
(define (max-of list)
  (cond
    [(= (len
    [else (m
```

Functional Programming

Many languages have been called functional over the years:

Haskell

```
maxOf :: [Int] → Int
maxOf = foldr1 max
```

```
(define (m
  (cond
    [(= (length lst) 1) (first lst)]
    [else (max (first lst) (max-of (rest lst)))]
```

Functional Programming

Many languages have been called functional over the years:

Haskell

```
maxOf :: [Int] → Int
maxOf = foldr1 max
```

```
(define
  (cond
    [(= (length lst) 1) (first lst)]
    [else (max (first lst) (max-of (rest lst
```


JavaScript?

```
function maxOf(arr) {
  var max = arr.reduce(function(a, b) {
    return Math.max(a, b);
  }); }
```

Functional Programming

Many languages have been called functional over the years:

Haskell

```
maxOf :: [Int] → Int
maxOf = foldr1 max
```

```
(define
  (cond
    [(= (length lst) 1) (first lst)]
    [else (max (first lst) (max-of (rest lst
```


JavaScript?

```
function maxOf(arr) {
  var max = arr.reduce(function(a, b) {
    return Math.max(a, b);
  }); }
```

What do they have in common?

Definitions

Unlike imperative languages, functional programming languages are not very crisply defined.

Attempt at a Defi

A *functional pro* or
inspired by the
programming language.

The result? If it has λ in it, you can call it functional.

Definitions

Unlike imperative languages, functional programming languages are not very crisply defined.

Attempt at a Defi

A *functional pro* or
inspired by the
programming language.

The result? If it has λ in it, you can call it functional.

In this course, we'll consider *purely functional* languages, which have a much better definition.

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Functions as Values?

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Functions as Values?

Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Polymorphism?

Functions as Values?

Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Polymorphism?
ML, 1973

Functions as Values?
Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Type Inference?

Polymorphism?

ML, 1973

Functions as Values?

Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Type Inference?
ML, 1973

Polymorphism?
ML, 1973

Functions as Values?
Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Type Inference?
ML, 1973

g?

Polymorphism?
ML, 1973

Functions as Values?
Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Type Inference?
ML, 1973

g?

Polymorphism?
ML, 1973

Functions as Values?
Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Type Inference?
ML, 1973

g?

Polymorphism?
ML, 1973

Lazy Evaluation?

Functions as Values?
Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Type Inference?
ML, 1973

g?

Polymorphism?
ML, 1973

Lazy Evaluation?
Miranda, 1985

Functions as Values?
Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Monads?

Type Inference?

ML, 1973

g?

Polymorphism?

ML, 1973

Lazy Evaluation?

Miranda, 1985

Functions as Values?

Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Monads?

Has

Type Inference?

ML, 1973

g?

Polymorphism?

ML, 1973

Lazy Evaluation?

Miranda, 1985

Functions as Values?

Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages.

Monads?

Has

Type Inference?

ML, 1973

Software Tran

g?

Polymorphism?

ML, 1973

Lazy Evaluation?

Miranda, 1985

Functions as Values?

Lisp, 1958

Why Study FP Languages?

Think of a major innovation in the area of programming languages

Monads?

Has

Type Inference?

ML, 1973

Software Tran

GHC Haskell, 2005

g?

Polymorphism?

ML, 1973

Lazy Evaluation?

Miranda, 1985

Functions as Values?

Lisp, 1958

Purely Functional Programming Languages

The term *purely functional* has a very crisp definition.

Definition

A programming l on in general)
is actually a conflu
In other words, fun *side effects*

Purely Functional Programming Languages

The term *purely functional* has a very crisp definition.

Definition

A programming l on in general)
is actually a conflu
In other words, fun side effects

Consider what would happen if we allowed effects in a functional language:

$$count = 0;$$
$$f\ x = \{count := count +$$
$$m = (\lambda y.\ y + y)\ (f\ 3)$$

If we evaluate $f\ 3$ first, we will get $m = 6$, but if we β-reduce m first, we will get $m = 9$. $\Rightarrow$ not confluent.

Making a Functional Language

We're going to make a language called **MinHS**.

1. Three type

Making a Functional Language

We're going to make a language called **MinHS**.

1. Three type
2. Static type c

Making a Functional Language

We're going to make a language called **MinHS**.

1. Three type
2. Static type c
3. Purely func

Making a Functional Language

We're going to make a language called **MinHS**.

1. Three type
2. Static type c
3. Purely func
4. Call-by-value (strict evaluation)

Making a Functional Language

We're going to make a language called **MinHS**.

1. Three type
2. Static type
3. Purely func
4. Call-by-value (strict evaluation)

In your Assignment 1, you will be implementing an evaluator for a dialect of MinHS.

Syntax

Integers $n ::= \ldots$

Identifiers $f, x ::= \ldots$

$$\mid \texttt{if } e_1 \qquad 1 \qquad 2$$

Syntax

Integers	n	$::=$	$\ldots$
Identifiers	f, x	$::=$	$\ldots$
		$\mid$	$\texttt{if}\ e_1$
		$\mid$	$e_1\ e_2$

Syntax

Integers	n	$::=$	$\ldots$
Identifiers	f, x	$::=$	$\ldots$

$$\mid \texttt{if}\ e_1\ e_2\ e_3 \mid \texttt{fun}\ f\ x \rightarrow e \mid \texttt{rec}\ f\ x \rightarrow e$$

↑ Like λ, but with recursion.

As usual, this is ambiguous concrete syntax. But all the precedence and associativity rule apply as in Haskell. We assume a suitable parser.

Examples

Example (Stupid division by 5)

$$\textbf{recfun}\ \mathit{divBy5} :: (\mathrm{Int} \to \mathrm{Int})\ x =$$
$$\textbf{if}\ x < 5$$

Example (Average Function)

$$\textbf{recfun}\ \mathit{average} :: (\mathrm{Int} \to (\mathrm{Int} \to \mathrm{Int}))\ x =$$
$$\textbf{recfun}\ \mathit{avX} :: (\mathrm{Int} \to \mathrm{Int})\ y =$$
$$(x + y)\ /\ 2$$

As in Haskell, $(\mathit{average}\ 15\ 5) = ((\mathit{average}\ 15)\ 5)$.

We don't need no let

We don't need no let

This language is so can we do without them?

let x_1 t_1 x_2 t_2 $\equiv$ $(\lambda x_1\ x_2.\ e_2)\ e_1\ t_1\ t_2$

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num Int`).
2. Operators like $a + b$ become (`Plus` a b).

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num` *i*).
2. Operators like $a + b$ become (`Plus` *a b*).
3. **if** *c* **then**

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num Int`).
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then**
4. Function a

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num Int`).
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then**
4. Function a
5. **recfun** f $1 \rightarrow 2$ `Recfun 1 2`

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num Int`)
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then**
4. Function a
5. **recfun** f $1 \rightarrow 2$ `Recfun 1 2`
6. Variable usages are wrapped in a term (`Var`

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num Int`)
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then**
4. Function a
5. **recfun** f $1 \rightarrow 2$ `Recfun 1 2`
6. Variable usages are wrapped in a term (`Var`

What changes when we move to higher order abstract syntax?

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num Int`).
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then**
4. Function a
5. **recfun** f $1 \to 2$ `Recfun 1 2`
6. Variable usages are wrapped in a term (`Var`

What changes when we move to higher order abstract syntax?

1. `Var` terms go away – we use the meta-language's variables.
2. (`Recfun` τ_1 τ_2 f x e) now uses meta-language abstraction: (`Recfun` τ_1 τ_2 (f. x. e)).

Working Statically with HOAS

To Code

We're going to writ
Seeing as this requi
have to extend the AST with special "tag" values.

Static Semantics

To check if a MinHS program is well-formed, we need to check:

1. **Scoping** – all variables used must be well defined
2. **Typing** – all operations must be used on compatible types.

Static Semantics

To check if a MinHS program is well-formed, we need to check:

1. **Scoping** – all variables used must be well defined
2. **Typing** – all operations must be used on compatible types.

Our judgement is a

Unde

$\vdash$

The ex

The context Γ includes typing assumptions for the variables:

$$x : \mathrm{Int}, y : \mathrm{Int} \vdash (\texttt{Plus}\ x\ y) : \mathrm{Int}$$

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

Let's implement a *type checker*.

Static Semantics

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}$$

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma,}{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.\ x.\ e)) :}$$

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1, \quad}{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.\ x.\ e)) :}$$

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1, \ldots}{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.\ x.\ e)) :}$$

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1, f : (}{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.\ x.\ e)) : \tau_1 \to \tau_2}$$

$$\frac{}{\Gamma \vdash (\texttt{Apply}\ e_1\ e_2) :}$$

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1, f : (}{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.\ x.\ e)) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2}{\Gamma \vdash (\texttt{Apply}\ e_1\ e_2) :}$$

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1, f : (}{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.\ x.\ e)) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (\texttt{Apply}\ e_1\ e_2) :}$$

Let's implement a *type checker*.

Static Semantics

$$\frac{}{\Gamma \vdash (\texttt{Num}\ n) : \texttt{Int}} \qquad \frac{}{\Gamma \vdash (\texttt{Lit}\ b) : \texttt{Bool}}$$

$$\frac{}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e}$$

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1, f : (\quad}{\Gamma \vdash (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.\ x.\ e)) : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (\texttt{Apply}\ e_1\ e_2) : \tau_2}$$

Let's implement a *type checker*.

Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states:

Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states: All well typed expressions.

Final states:

Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states: All well typed expressions.

Final states:

Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states: All well typed expressions.

Final states:

Evaluation of bu

$$\frac{e_1 \mapsto e_1'}{(\texttt{Plus}\ e_1\ e_2) \mapsto (\texttt{P}}$$

(and so on as per arithmetic expre

Specifying If

$$\frac{}{(\texttt{If (Lit True)}\ e_2} \qquad \frac{}{(\texttt{If (Lit False)}\ e_2}$$

How about Functions?

Recall that `Recfun` is a final state – we don't need to evaluate it when it's alone.

Evaluating function application requires us to:

1. Evaluate the left expression to get the function being applied
2. Evaluate th
3. Evaluate th variables.

$$\frac{e_1 \mapsto e_1'}{(\texttt{Apply}\ e_1\ e_2) \mapsto (\texttt{Ap}}$$

$$\frac{e_2 \mapsto e_2'}{(\texttt{Apply}\ (\texttt{Recfun}\ldots)\ e_2) \mapsto (\texttt{Apply}\ (\texttt{Recfun}\ldots)\ e_2')}$$

How about Functions?

Recall that `Recfun` is a final state – we don't need to evaluate it when it's alone.

Evaluating function application requires us to:

1. Evaluate the left expression to get the function being applied
2. Evaluate th
3. Evaluate th variables.

$$\frac{e_1 \mapsto e_1'}{(\texttt{Apply}\ e_1\ e_2) \mapsto (\texttt{Ap}}$$

$$\frac{e_2 \mapsto e_2'}{(\texttt{Apply}\ (\texttt{Recfun}\ldots)\ e_2) \mapsto (\texttt{Apply}\ (\texttt{Recfun}\ldots)\ e_2')}$$

How about Functions?

Recall that `Recfun` is a final state – we don't need to evaluate it when it's alone.

Evaluating function application requires us to:

1. Evaluate the left expression to get the function being applied
2. Evaluate th
3. Evaluate th variables.

$$\frac{e_1 \mapsto e_1'}{(\texttt{Apply}\ e_1\ e_2) \mapsto (\texttt{Ap}}$$

$$\frac{e_2 \mapsto e_2'}{(\texttt{Apply}\ (\texttt{Recfun}\ldots)\ e_2) \mapsto (\texttt{Apply}\ (\texttt{Recfun}\ldots)\ e_2')}$$

$$\frac{v \in F}{(\texttt{Apply}\ (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.x.\ e))\ v) \mapsto e[x := v, f := (\texttt{Recfun}\ \tau_1\ \tau_2\ (f.x.\ e))]}$$