



Functional Programming Languages

Thomas Sewell
UNSW
Term 3 2025

Functional Programming

Many languages have been called **functional** over the years:

Haskell

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

Lisp

```
(define (max-of lst)  
  (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 Definition

A *functional programming language* is a programming language derived from or inspired by the λ -calculus, or derived from or inspired by another functional 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.

Monads?

Haskell, 1991

Type Inference?

ML, 1973

Garbage Collection?

Lisp, 1958

Software Transactional Memory?

GHC Haskell, 2005

Metaprogramming?

Lisp, 1958

Polymorphism?

ML, 1973

Functions as Values?

Lisp, 1958

Lazy Evaluation?

Miranda, 1985

Purely Functional Programming Languages

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

Definition

A programming language is *purely functional* if β -reduction (or evaluation in general) is actually a **confluence**.

In other words, functions have to be mathematical functions, and free of *side effects*.

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

```
count = 0;  
f x = {count := count + x; return 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 types of values: integers, booleans, and **functions**.
- 2 Static type checking (not inference)
- 3 Purely functional (no effects)
- 4 Call-by-value (strict evaluation)

Something not unlike this will appear in your **Assignment 1**.

Syntax

<i>Integers</i>	n	$::=$	\dots
<i>Identifiers</i>	f, x	$::=$	\dots
<i>Literals</i>	b	$::=$	True False
<i>Types</i>	τ	$::=$	Bool Int $\tau_1 \rightarrow \tau_2$
<i>Infix Operators</i>	\otimes	$::=$	* + == \dots
<i>Expressions</i>	e	$::=$	x n b (e) $e_1 \otimes e_2$ if e_1 then e_2 else e_3 $e_1 e_2$ $\text{recfun } f :: (\tau_1 \rightarrow \tau_2) x = e$ ↑ Like λ , but with recursion.

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

Examples

Example (Stupid division by 5)

```
recfun divBy5 :: (Int → Int) x =  
  if x < 5  
  then 0  
  else 1 + divBy5 (x - 5)
```

Example (Average Function)

```
recfun average :: (Int → (Int → Int)) x =  
  recfun avX :: (Int → Int) y =  
    (x + y) / 2
```

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

We don't need no let

This language is so minimal, it doesn't even need **let** expressions.
How can we do without them?

let $x :: \tau_1 = e_1$ **in** $e_2 :: \tau_2$ \equiv (**recfun** $f :: (\tau_1 \rightarrow \tau_2)$ $x = e_2$) e_1

Abstract Syntax

Moving to **first order** abstract syntax, we get:

Concrete Syntax	Abstract Syntax
n	(Num n)
b	(Lit n)
if c then t else e	(If c t e)
e_1 e_2	(Apply e_1 e_2)
recfun $f :: (\tau_1 \rightarrow \tau_2) x = e$	(Recfun τ_1 τ_2 f x e)
x	(Var x)

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 write code for an AST and pretty-printer for MinHS with HOAS.

Seeing as this requires us to **look under abstractions** without evaluating the term, we 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.

Our judgement is an extension of the scoping rules to include types:

Under this **context** of assumptions

$\Gamma \vdash e : \tau$

The expression is assigned this type

The **context** Γ includes **typing assumptions** for the variables:

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

Static Semantics

$$\frac{}{\Gamma \vdash (\text{Num } n) : \text{Int}} \quad \frac{}{\Gamma \vdash (\text{Lit } b) : \text{Bool}}$$
$$\frac{\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}}{\Gamma \vdash (\text{Plus } e_1 e_2) : \text{Int}}$$
$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\text{If } e_1 e_2 e_3) : \tau}$$
$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash (\text{Var } x) : \tau} \quad \frac{\Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2) \vdash e : \tau_2}{\Gamma \vdash (\text{Recfun } \tau_1 \tau_2 (f. x. e)) : \tau_1 \rightarrow \tau_2}$$
$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (\text{Apply } e_1 e_2) : \tau_2}$$

Let's implement a *type checker*.

Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states: All well typed expressions.

Final states: (Num n), (Lit b), **Recfun too!**

Evaluation of built-in operations:

$$\frac{e_1 \mapsto e'_1}{(\text{Plus } e_1 \ e_2) \mapsto (\text{Plus } e'_1 \ e_2)}$$

(and so on as per arithmetic expressions)

Specifying If

$$\frac{e_1 \mapsto e'_1}{(\text{If } e_1 \ e_2 \ e_3) \mapsto (\text{If } e'_1 \ e_2 \ e_3)}$$
$$\frac{}{(\text{If } (\text{Lit True}) \ e_2 \ e_3) \mapsto e_2}$$
$$\frac{}{(\text{If } (\text{Lit False}) \ e_2 \ e_3) \mapsto e_3}$$

How about Functions?

Recall that `Recfun` is a **final state** – we don't need to evaluate it unless it's applied to an argument.

Evaluating **function application** requires us to:

- 1 Evaluate the left expression to get a `Recfun`;
- 2 evaluate the right expression to get an argument value; and
- 3 evaluate the function's body, after supplying substitutions for the abstracted variables.

$$\frac{\frac{\frac{e_1 \mapsto e'_1}{(\text{Apply } e_1 \ e_2) \mapsto (\text{Apply } e'_1 \ e_2)}}{e_2 \mapsto e'_2}}{(\text{Apply } (\text{Recfun } \dots) \ e_2) \mapsto (\text{Apply } (\text{Recfun } \dots) \ e'_2)}}{v \in F} \frac{}{(\text{Apply } (\text{Recfun } \tau_1 \ \tau_2 \ (f.x. \ e)) \ v) \mapsto e[x := v, f := (\text{Recfun } \tau_1 \ \tau_2 \ (f.x. \ e))]}$$

MinHS To Come

We've now looked at MinHS (0.10: beta).

More MinHS content will come soon as part of Assignment 1 and Assignment 2.