Functional Programming Languages

Johannes Åman Pohjola
UNSW
Term 3 2022
Functional Programming

Many languages have been called functional over the years:

**Haskell**

\[
\text{maxOf} :: \text{[Int]} \rightarrow \text{Int}
\]

\[
\text{maxOf} = \text{foldr1 max}
\]

**Lisp**

\[
\text{(define (max-of lst)}
\]

\[
\text{(cond}
\]

\[
[ (= \text{(length lst)} 1) \text{(first lst)}]
\]

\[
[\text{else (max (first lst) (max-of (rest lst)))}]\)
\]

**JavaScript?**

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

What do they have in common?
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 $\lambda$-calculus, or derived from or inspired by another functional programming language.

**The result?** If it has $\lambda$ 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
- Metaprogramming? Lisp, 1958
- Software Transactional Memory? GHC Haskell, 2005
- Polymorphism? ML, 1973
- Functions as Values? Lisp, 1958
The term purely functional has a very crisp definition.

**Definition**

A programming language is purely functional if \( \beta \)-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:

\[
\begin{align*}
\text{count} &= 0; \\
\text{f x} &= \{ \text{count} := \text{count} + x; \text{return count}\}; \\
\text{m} &= (\lambda y. y + y) (\text{f 3})
\end{align*}
\]

If we evaluate \( f \ 3 \) first, we will get \( m = 6 \), but if we \( \beta \)-reduce \( m \) first, we will get \( m = 9 \). ⇒ 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)

In your **Assignment 1**, you will be implementing an evaluator for a slightly less minimal dialect of MinHS.
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)

\[
\text{recfun } \text{divBy5} :: (\text{Int} \rightarrow \text{Int}) \ x = \\
\quad \text{if } x < 5 \\
\quad \quad \text{then } 0 \\
\quad \quad \text{else } 1 + \text{divBy5} (x - 5)
\]

Example (Average Function)

\[
\text{recfun } \text{average} :: (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})) \ x = \\
\text{recfun } \text{avgX} :: (\text{Int} \rightarrow \text{Int}) \ y = \\
\quad (x + y) / 2
\]

As in Haskell, \((\text{average } 15 \ 5) = ((\text{average } 15) \ 5)\).
We don’t need no let

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

\[
\text{let } x :: \tau_1 = e_1 \text{ in } e_2 :: \tau_2 \equiv (\text{recfun } f :: (\tau_1 \rightarrow \tau_2) \ x = e_2) \ e_1
\]
Abstract Syntax

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

<table>
<thead>
<tr>
<th>Concrete Syntax</th>
<th>Abstract Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$(\text{Num } n)$</td>
</tr>
<tr>
<td>$b$</td>
<td>$(\text{Lit } n)$</td>
</tr>
<tr>
<td><strong>if</strong> $c$ <strong>then</strong> $t$ <strong>else</strong> $e$</td>
<td>$(\text{If } c \ t \ e)$</td>
</tr>
<tr>
<td>$e_1 \ e_2$</td>
<td>$(\text{Apply } e_1 \ e_2)$</td>
</tr>
<tr>
<td><strong>recfun</strong> $f :: (\tau_1 \rightarrow \tau_2) ; x = e$</td>
<td>$(\text{Recfun } \tau_1 \ \tau_2 \ f \times e)$</td>
</tr>
<tr>
<td>$x$</td>
<td>$(\text{Var } x)$</td>
</tr>
</tbody>
</table>

What changes when we move to **higher order** abstract syntax?

1. **Var** terms go away – we use the meta-language’s variables.
2. $(\text{Recfun } \tau_1 \ \tau_2 \ f \times e)$ now uses meta-language abstraction: $(\text{Recfun } \tau_1 \ \tau_2 \ (f \times e))$. 
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 \( \Gamma \) includes typing assumptions for the variables:

\[ x : \text{Int}, y : \text{Int} \vdash (\text{Plus } x \ y) : \text{Int} \]
Let’s implement a type checker.
Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states: All well typed expressions.

Final states: $(\text{Num } n)$, $(\text{Lit } b)$, $\text{Recfun too!}$

Evaluation of built-in operations:

\[
e_1 \mapsto e'_1
\]

\[
(\text{Plus } e_1 e_2) \mapsto (\text{Plus } e'_1 e_2)
\]

(and so on as per arithmetic expressions)
Specifying If

\[
e_1 \mapsto e'_1
\]

\[
(\text{If } e_1 e_2 e_3) \mapsto (\text{If } e'_1 e_2 e_3)
\]

\[
(\text{If (Lit True) } e_2 e_3) \mapsto e_2
\]

\[
(\text{If (Lit False) } e_2 e_3) \mapsto e_3
\]
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.

\[
\begin{align*}
    e_1 \mapsto e'_1 \\
    \text{(Apply } e_1 \ e_2) \mapsto \text{(Apply } e'_1 \ e_2) \\
    e_2 \mapsto e'_2 \\
    \text{(Apply } (\text{Recfun} \ldots) \ e_2) \mapsto \text{(Apply } (\text{Recfun} \ldots) \ e'_2) \\
    v \in F \\
    \text{(Apply } (\text{Recfun } \tau_1 \ \tau_2 \ (f.\ x.\ e)) \ v) \mapsto e[x := v,\ f := (\text{Recfun } \tau_1 \ \tau_2 \ (f.\ x.\ e))] \end{align*}
\]