Functional Programming Languages

Johannes Åman Pohjola
UNSW
Term 3 2022
Functional Programming

Many languages have been called functional over the years:

**Lisp**

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

**Haskell**

```haskell
maxOf :: [Int] -> Int
maxOf = foldr1 max
```

**JavaScript**

```javascript
function maxOf(arr)
{
  var max = arr.reduce(function(a, b){return Math.max(a,b)});
}
```
Many languages have been called functional over the years:

**Lisp**

\[
\text{(define } (\text{max-of } \text{lst}) \\text{)} \\text{)} \\
\text{(cond} \\
\quad [(= (\text{length } \text{lst}) 1) (\text{first } \text{lst})] \\
\quad [\text{else (max (first } \text{lst}) (\text{max-of (rest } \text{lst}))])])
\]

**Haskell**

\[
\text{maxOf} :: [\text{Int}] \rightarrow \text{Int} \\
\text{maxOf } = \text{foldr1 max}
\]

**JavaScript**

\[
\text{function maxOf(arr)} \{
\text{var max = arr.} \text{reduce} (\text{function}(a, b) \{
\text{return Math.max(a, b);}
\})
\}
\]
Functional Programming
Many languages have been called functional over the years:

Lisp

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

Haskell

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

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
```

**Lisp**

```
(define (max-of lst)
  (cond
    [(= (length lst) 1) (first lst)]
    [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.
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 $\lambda$-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.

Garbage Collection?
Why Study FP Languages?

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

Garbage Collection?
Lisp, 1958
Why Study FP Languages?

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

Garbage Collection?
Lisp, 1958

Functions as Values?

Software Transactional Memory?
GHC Haskell, 2005

Polymorphism?
ML, 1973

Type Inference?
ML, 1973

Metaprogramming?
Lisp, 1958

Lazy Evaluation?
Miranda, 1985

Monads?
Haskell, 1991
Why Study FP Languages?

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

Garbage Collection?
Lisp, 1958

Functions as Values?
Lisp, 1958
Why Study FP Languages?

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

Garbage Collection?
Lisp, 1958

Polymorphism?

Functions as Values?
Lisp, 1958
Why Study FP Languages?

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

Garbage Collection?
Lisp, 1958

Polymorphism?
ML, 1973

Functions as Values?
Lisp, 1958
Why Study FP Languages?

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

- Garbage Collection?
  - Lisp, 1958
- Functions as Values?
  - Lisp, 1958
- Polymorphism?
  - ML, 1973
- Type Inference?
- Garbage Collection?
  - Lisp, 1958
- Lazy Evaluation?
  - Miranda, 1985
- Monads?
  - Haskell, 1991
- Software Transactional Memory?
  - GHC Haskell, 2005
Why Study FP Languages?

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

- **Type Inference?**
  - ML, 1973

- **Garbage Collection?**
  - Lisp, 1958

- **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

- Garbage Collection?  
  Lisp, 1958

- Polymorphism?  
  ML, 1973

- Functions as Values?  
  Lisp, 1958

- Metaprogramming?

- Lazy Evaluation?  
  Miranda, 1985

- Monads?  
  Haskell, 1991

- Software Transactional Memory?  
  GHC Haskell, 2005
Why Study FP Languages?

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

- Type Inference?
  - ML, 1973

- Garbage Collection?
  - Lisp, 1958

- Polymorphism?
  - ML, 1973

- Metaprogramming?
  - Lisp, 1958

- Functions as Values?
  - Lisp, 1958
Why Study FP Languages?

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

- **Type Inference?**
  - ML, 1973

- **Garbage Collection?**
  - Lisp, 1958

- **Polymorphism?**
  - ML, 1973

- **Functions as Values?**
  - Lisp, 1958

- **Metaprogramming?**
  - Lisp, 1958

- **Lazy Evaluation?**
  - Miranda, 1985

- **Monads?**
  - Haskell, 1991

- **Software Transactional Memory?**
  - GHC Haskell, 2005
Why Study FP Languages?

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

- **Type Inference?**
  - ML, 1973

- **Garbage Collection?**
  - Lisp, 1958

- **Polymorphism?**
  - ML, 1973

- **Functions as Values?**
  - Lisp, 1958

- **Metaprogramming?**
  - Lisp, 1958

- **Lazy Evaluation?**
  - Miranda, 1985

- **Monads?**
  - Haskell, 1991

- **Software Transactional Memory?**
  - GHC Haskell, 2005
Why Study FP Languages?

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

- Monads? ML, 1973
- Garbage Collection? Lisp, 1958
- Polymorphism? ML, 1973
- Functions as Values? Lisp, 1958
- Type Inference? ML, 1973
- Metaprogramming? Lisp, 1958
Why Study FP Languages?

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

- Monads?
  Haskell, 1991
- Garbage Collection?
  Lisp, 1958
- Polymorphism?
  ML, 1973
- Functions as Values?
  Lisp, 1958
- Lazy Evaluation?
  Miranda, 1985
- Type Inference?
  ML, 1973
- Metaprogramming?
  Lisp, 1958
- Software Transactional Memory?
  GHC Haskell, 2005
- Monads?
  Haskell, 1991
Think of a major innovation in the area of programming languages.

- **Monads?**
  - Haskell, 1991

- **Garbage Collection?**
  - Lisp, 1958

- **Polymorphism?**
  - ML, 1973

- **Functions as Values?**
  - Lisp, 1958

- **Lazy Evaluation?**
  - Miranda, 1985

- **Type Inference?**
  - ML, 1973

- **Software Transactional Memory?**
  - Lisp, 1958

- **Metaprogramming?**
  - Lisp, 1958
Why Study FP Languages?

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

- **Monads?**
  - Haskell, 1991

- **Garbage Collection?**
  - Lisp, 1958

- **Type Inference?**
  - ML, 1973

- **Software Transactional Memory?**
  - GHC Haskell, 2005

- **Polymorphism?**
  - ML, 1973

- **Functions as Values?**
  - Lisp, 1958

- **Metaprogramming?**
  - 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 $\beta$-reduction (or evaluation in general) is actually a confluence. In other words, functions have to be mathematical functions, and free of *side effects*.
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:

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

If we evaluate \( f \ 3 \) first, we will get \( m = 6 \), but if we \( \beta \)-reduce \( m \) first, we will get \( m = 9 \). \( \Rightarrow \) not confluent.
Making a Functional Language

We’re going to make a language called MinHS.

1. Three types of values: integers, booleans, and functions.
We’re going to make a language called MinHS.

1. Three types of values: integers, booleans, and functions.
2. Static type checking (not inference)
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)
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)
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.
Syntax

Integers \( n \ ::= \ \cdots \)

Identifiers \( f, x \ ::= \ \cdots \)

Literals \( b \ ::= \ \text{True} \mid \text{False} \)

Types \( \tau \ ::= \ \text{Bool} \mid \text{Int} \mid \tau_1 \to \tau_2 \)

Infix Operators \( \ast \ ::= \ \ast \mid + \mid == \mid \ \cdots \)

Expressions \( e \ ::= \ x \mid n \mid b \mid (e) \mid e_1 \ast e_2 \)
\[ \quad \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]
Syntax

**Integers**

\[ n ::= \cdots \]

**Identifiers**

\[ f, x ::= \cdots \]

**Literals**

\[ b ::= \text{True} \mid \text{False} \]

**Types**

\[ \tau ::= \text{Bool} \mid \text{Int} \mid \tau_1 \rightarrow \tau_2 \]

**Infix Operators**

\[ \otimes ::= * \mid + \mid == \mid \cdots \]

**Expressions**

\[ e ::= x \mid n \mid b \mid (e) \mid e_1 \otimes e_2 \]

\[ \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

\[ \mid e_1 \ e_2 \]
Syntax

**Integers**  \( n ::= \cdots \)

**Identifiers**  \( f, x ::= \cdots \)

**Literals**  \( b ::= \text{True} | \text{False} \)

**Types**  \( \tau ::= \text{Bool} | \text{Int} | \tau_1 \to \tau_2 \)

**Infix Operators**  \( \star ::= * | + | == | \cdots \)

**Expressions**  
\[
e ::= x | n | b | (e) | e_1 \star e_2 \\
| \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
| e_1 e_2 \\
| \text{recfun } f :: (\tau_1 \to \tau_2) \ x = e
\]

↑ Like \(\lambda\), 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)

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

Example (Average Function)

```haskell
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?
We don’t need no let

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

\[
\text{let } x :: \tau_1 = e_1 \text{ in } e_2 :: \tau_2 \equiv (\text{recfun } f :: (\tau_1 \to \tau_2) \ x = e_2) \ e_1
\]
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>(Num ( n ))</td>
</tr>
<tr>
<td>( b )</td>
<td>(Lit ( n ))</td>
</tr>
</tbody>
</table>
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>
</tbody>
</table>
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>(Num $n$)</td>
</tr>
<tr>
<td>$b$</td>
<td>(Lit $n$)</td>
</tr>
<tr>
<td>$\text{if } c \text{ then } t \text{ else } e$</td>
<td>(If $c$ $t$ $e$)</td>
</tr>
<tr>
<td>$e_1$ $e_2$</td>
<td>(Apply $e_1$ $e_2$)</td>
</tr>
</tbody>
</table>
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>(Num ( n ))</td>
</tr>
<tr>
<td>( b )</td>
<td>(Lit ( n ))</td>
</tr>
<tr>
<td>if ( c ) then ( t ) else ( e )</td>
<td>(If ( c ) ( t ) ( e ))</td>
</tr>
<tr>
<td>( e_1 ) ( e_2 )</td>
<td>(Apply ( e_1 ) ( e_2 ))</td>
</tr>
<tr>
<td>recfun ( f :: (\tau_1 \rightarrow \tau_2) ) ( x = e )</td>
<td>(Recfun ( \tau_1 ) ( \tau_2 ) ( f ) ( x ) ( e ))</td>
</tr>
</tbody>
</table>

What changes when we move to higher order abstract syntax?

1. \( \text{Var} \) terms go away – we use the meta-language’s variables.
2. \( \text{Recfun} \) now uses meta-language abstraction: \( \text{Recfun} \( \tau_1 \) \( \tau_2 \) (\( f \). \( x \). \( e \)) \).
### 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>\textbf{if} ( c ) \textbf{then} ( t ) \textbf{else} ( 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>\textbf{recfun} ( f :: (\tau_1 \to \tau_2) \ x = e )</td>
<td>((\text{Recfun } \tau_1 \ \tau_2 \ f \ \ x \ 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. \((\text{Recfun } \tau_1 \ \tau_2 \ f \ \ x \ e)\) now uses meta-language abstraction: \((\text{Recfun } \tau_1 \ \tau_2 \ (f . x . e))\).
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>(Num ( n ))</td>
</tr>
<tr>
<td>( b )</td>
<td>(Lit ( n ))</td>
</tr>
<tr>
<td><strong>if</strong> ( c ) <strong>then</strong> ( t ) <strong>else</strong> ( e )</td>
<td>(If ( c ) ( t ) ( e ))</td>
</tr>
<tr>
<td>( e_1 ) ( e_2 )</td>
<td>(Apply ( e_1 ) ( e_2 ))</td>
</tr>
<tr>
<td><strong>recfun</strong> ( f :: (( \tau_1 ) \rightarrow ( \tau_2 )) ) ( x ) = ( e )</td>
<td>(Recfun ( \tau_1 ) ( \tau_2 ) ( f ) ( x ) ( e ))</td>
</tr>
<tr>
<td>( x )</td>
<td>(Var ( x ))</td>
</tr>
</tbody>
</table>

What changes when we move to **higher order** abstract syntax?
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>(Num (n))</td>
</tr>
<tr>
<td>(b)</td>
<td>(Lit (n))</td>
</tr>
<tr>
<td>if (c) then (t) else (e)</td>
<td>(If (c) (t) (e))</td>
</tr>
<tr>
<td>(e_1) (e_2)</td>
<td>(Apply (e_1) (e_2))</td>
</tr>
<tr>
<td>recfun (f :: (\tau_1 \rightarrow \tau_2) \ x = e)</td>
<td>(Recfun (\tau_1) (\tau_2) (f) (x) (e))</td>
</tr>
<tr>
<td>(x)</td>
<td>(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. \((\text{Recfun } \tau_1 \tau_2 f x e)\) now uses meta-language abstraction: \((\text{Recfun } \tau_1 \tau_2 (f \ x \ e))\).
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.
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:

\[ \Gamma \vdash e : \tau \]

The context \( \Gamma \) includes typing assumptions for the variables:

\[ x : \text{Int}, y : \text{Int} \vdash (\text{Plus} \ x \ y) : \text{Int} \]
Static Semantics

\[ \Gamma \vdash (\text{Num } n) : \text{Int} \quad \Gamma \vdash (\text{Lit } b) : \text{Bool} \]

\[ \Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int} \]

\[ \Gamma \vdash (\text{Plus } e_1 \ e_2) : \text{Int} \]

\[ \Gamma \vdash (\text{If } e_1 \ e_2 \ e_3) : \]
Let's implement a type checker.
Static Semantics

\[ \Gamma \vdash (\text{Num } n) : \text{Int} \quad \Gamma \vdash (\text{Lit } b) : \text{Bool} \]

\[ \begin{align*}
\Gamma & \vdash e_1 : \text{Int} \\
\Gamma & \vdash e_2 : \text{Int}
\end{align*} \]

\[ \Gamma \vdash (\text{Plus } e_1 e_2) : \text{Int} \]

\[ \Gamma \vdash e_1 : \text{Bool} \]

\[ \Gamma \vdash (\text{If } e_1 e_2 e_3) : \]

Let's implement a type checker.
Static Semantics

\[ \Gamma \vdash (\text{Num } n) : \text{Int} \quad \Gamma \vdash (\text{Lit } b) : \text{Bool} \]

\[ \Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int} \]

\[ \Gamma \vdash (\text{Plus } e_1 e_2) : \text{Int} \]

\[ \Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \]

\[ \Gamma \vdash (\text{If } e_1 e_2 e_3) : \tau \]
Let's implement a type checker.

\[
\begin{align*}
\Gamma \vdash \text{(Num } n\text{)} : \text{Int} & \quad \Gamma \vdash \text{(Lit } b\text{)} : \text{Bool} \\
\Gamma \vdash e_1 : \text{Int} & \quad \Gamma \vdash e_2 : \text{Int} \\
\Gamma \vdash (\text{Plus } e_1 \ e_2) : \text{Int} \\
\Gamma \vdash e_1 : \text{Bool} & \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \\
\Gamma \vdash (\text{If } e_1 \ e_2 \ e_3) : \tau \\
(x : \tau) \in \Gamma & \quad \Gamma \vdash (\text{Var } x) : \tau
\end{align*}
\]
Static Semantics

\[
\Gamma \vdash (\text{Num } n) : \text{Int} \quad \Gamma \vdash (\text{Lit } b) : \text{Bool}
\]

\[
\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}
\]

\[
\Gamma \vdash (\text{Plus } e_1 e_2) : \text{Int}
\]

\[
\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau
\]

\[
\Gamma \vdash (\text{If } e_1 e_2 e_3) : \tau
\]

\[
(\mathit{x} : \tau) \in \Gamma \\
\Gamma, \quad \Gamma \vdash e : \tau_2
\]

\[
\Gamma \vdash (\text{Var } \mathit{x}) : \tau \\
\Gamma \vdash (\text{Recfun } \tau_1 \tau_2 (f. \mathit{x}. e)) : \\
\]
Let's implement a type checker.

\[
\begin{align*}
\Gamma \vdash (\text{Num } n) : \text{Int} & \quad \Gamma \vdash (\text{Lit } b) : \text{Bool} \\
\Gamma \vdash e_1 : \text{Int} & \quad \Gamma \vdash e_2 : \text{Int} \\
\Gamma \vdash (\text{Plus } e_1 e_2) : \text{Int} \\
\Gamma \vdash e_1 : \text{Bool} & \quad \Gamma \vdash e_2 : \tau & \quad \Gamma \vdash e_3 : \tau \\
\Gamma \vdash (\text{If } e_1 e_2 e_3) : \tau \\
(x : \tau) \in \Gamma & \quad \Gamma, x : \tau_1, \quad \Gamma \vdash e : \tau_2 \\
\Gamma \vdash (\text{Var } x) : \tau & \quad \Gamma \vdash (\text{Recfun } \tau_1 \tau_2 (f. \ x. \ e)) : \tau
\end{align*}
\]
Static Semantics

\[
\begin{align*}
\Gamma \vdash (\text{Num } n) : \text{Int} & \quad \Gamma \vdash (\text{Lit } b) : \text{Bool} \\
\Gamma \vdash e_1 : \text{Int} & \quad \Gamma \vdash e_2 : \text{Int} \quad \Gamma \vdash (\text{Plus } e_1 \ e_2) : \text{Int} \\
\Gamma \vdash e_1 : \text{Bool} & \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \quad \Gamma \vdash (\text{If } e_1 \ e_2 \ e_3) : \tau \\
(x : \tau) \in \Gamma & \quad \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2) \vdash e : \tau_2 \\
\Gamma \vdash (\text{Var } x) : \tau & \quad \Gamma \vdash (\text{Recfun } \tau_1 \ \tau_2 \ (f \ x \ e)) : \\
\end{align*}
\]
Static Semantics

\[
\begin{align*}
\Gamma \vdash (\text{Num } n) : \text{Int} & \quad \Gamma \vdash (\text{Lit } b) : \text{Bool} \\
\Gamma \vdash e_1 : \text{Int} & \quad \Gamma \vdash e_2 : \text{Int} \\
& \quad \Gamma \vdash (\text{Plus } e_1 e_2) : \text{Int} \\
\Gamma \vdash e_1 : \text{Bool} & \quad \Gamma \vdash e_2 : \tau & \quad \Gamma \vdash e_3 : \tau \\
& \quad \Gamma \vdash (\text{If } e_1 e_2 e_3) : \tau \\
(x : \tau) \in \Gamma & \quad \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2) \vdash e : \tau_2 \\
\Gamma \vdash (\text{Var } x) : \tau & \quad \Gamma \vdash (\text{Recfun } \tau_1 \tau_2 (f. x. e)) : \tau_1 \rightarrow \tau_2 \\
& \quad \Gamma \vdash (\text{Apply } e_1 e_2) : \\
\end{align*}
\]
Let's implement a type checker.
Let's implement a type checker.
Static Semantics

\[
\Gamma \vdash (\text{Num } n) : \text{Int} \quad \Gamma \vdash (\text{Lit } b) : \text{Bool}
\]
\[
\Gamma \vdash e_1 : \text{Int} \quad \Gamma \vdash e_2 : \text{Int}
\]
\[
\Gamma \vdash (\text{Plus } e_1 e_2) : \text{Int}
\]
\[
\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau
\]
\[
\Gamma \vdash (\text{If } e_1 e_2 e_3) : \tau
\]
\[
(x : \tau) \in \Gamma \quad \Gamma, x : \tau_1, f : (\tau_1 \to \tau_2) \vdash e : \tau_2
\]
\[
\Gamma \vdash (\text{Var } x) : \tau \quad \Gamma \vdash (\text{Recfun } \tau_1 \tau_2 (f \cdot x \cdot e)) : \tau_1 \to \tau_2
\]
\[
\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1
\]
\[
\Gamma \vdash (\text{Apply } e_1 e_2) : \tau_2
\]
Static Semantics

\[
\begin{align*}
\Gamma \vdash (\text{Num } n) : \text{Int} & \quad \Gamma \vdash (\text{Lit } b) : \text{Bool} \\
\Gamma \vdash e_1 : \text{Int} & \quad \Gamma \vdash e_2 : \text{Int} \\
\Gamma \vdash (\text{Plus } e_1 e_2) : \text{Int} \\
\Gamma \vdash e_1 : \text{Bool} & \quad \Gamma \vdash e_2 : \tau & \quad \Gamma \vdash e_3 : \tau \\
\Gamma \vdash (\text{If } e_1 e_2 e_3) : \tau \\
(x : \tau) \in \Gamma & \quad \Gamma, x : \tau_1, f : (\tau_1 \rightarrow \tau_2) \vdash e : \tau_2 \\
\Gamma \vdash (\text{Var } x) : \tau & \quad \Gamma \vdash (\text{Recfun } \tau_1 \tau_2 (f \cdot x \cdot e)) : \tau_1 \rightarrow \tau_2 \\
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 & \quad \Gamma \vdash e_2 : \tau_1 \\
\Gamma \vdash (\text{Apply } e_1 e_2) : \tau_2
\end{align*}
\]

Let’s implement a type checker.
Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states:

All well typed expressions.

Final states:

\((\text{Num } n), (\text{Lit } b))\)

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)
## Structural Operational Semantics (Small-Step)

**Initial states:** All well typed expressions.

**Final states:**

- $(\text{Num}, n)$
- $(\text{Lit}, b)$
- ${\text{Recfun}}$ too!
Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states: All well typed expressions.
Final states: $(\text{Num } n), (\text{Lit } b)$,
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:

\[
\begin{align*}
e_1 & \mapsto e'_1 \\
(\text{Plus } e_1 e_2) & \mapsto (\text{Plus } e'_1 e_2)
\end{align*}
\]

(and so on as per arithmetic expressions)
Specifying `if`

\[
\begin{align*}
e_1 & \mapsto e'_1 \\
(\text{If } e_1 e_2 e_3) & \mapsto (\text{If } e'_1 e_2 e_3) \\
(\text{If (Lit True) } e_2 e_3) & \mapsto e_2 \\
(\text{If (Lit False) } e_2 e_3) & \mapsto e_3
\end{align*}
\]
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.

\[
\begin{align*}
&\quad e_1 \mapsto e'_1 \\
&\quad \text{(Apply } e_1\ e_2) \mapsto \text{(Apply } e'_1\ e_2) \\
&\quad e_2 \mapsto e'_2 \\
&\quad \text{(Apply } (\text{Recfun} \ldots)\ e_2) \mapsto \text{(Apply } (\text{Recfun} \ldots)\ e'_2) \\
\end{align*}
\]
How about Functions?

Recall that \texttt{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 \texttt{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.

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