Functional Programming

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

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)))]))
```

Functional Programming

Many languages have been called functional over the years:

Haskell

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

Lisp

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

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

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);
  });
}
```

Functional Programming

Many languages have been called functional over the years:

Haskell

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

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.

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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?
Haskell, 1991

Type Inference?
ML, 1973

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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.

Type Inference?
ML, 1973

Monads?
Haskell, 1991

Garbage Collection?
Lisp, 1958

Software Transactional Memory?
GHC Haskell, 2005

Metaprogramming?
Lisp, 1958

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 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:

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

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.

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:

$$\begin{array}{l} count = 0; \\ f\ x \ = \{count := count + x; \textbf{return}\ count\}; \\ m = (\lambda y.\ y + y)\ (f\ 3) \end{array}$$

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)

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

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.

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.

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.

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	$::=$	$\texttt{True} \mid \texttt{False}$
Types	τ	$::=$	$\mathrm{Bool} \mid \mathrm{Int} \mid \tau_1 \to \tau_2$
Infix Operators	$\circledast$	$::=$	$\texttt{*} \mid \texttt{+} \mid \texttt{==} \mid \cdots$
Expressions	e	$::=$	$x \mid n \mid b \mid (e) \mid e_1 \circledast e_2$
		$\mid$	$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$
		$\mid$	$e_1\ e_2$
		$\mid$	$\texttt{recfun}\ f :: (\tau_1 \to \tau_2)\ x = e$
			$\uparrow$ 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.

Syntax

Integers	n	$::=$	$\cdots$
Identifiers	f, x	$::=$	$\cdots$
Literals	b	$::=$	`True` $\mid$ `False`
Types	τ	$::=$	$\mathrm{Bool} \mid \mathrm{Int} \mid \tau_1 \to \tau_2$
Infix Operators	$\circledast$	$::=$	`*` $\mid$ `+` $\mid$ `==` $\mid$ $\cdots$
Expressions	e	$::=$	$x \mid n \mid b \mid (e) \mid e_1 \circledast e_2$
		$\mid$	`if` e_1 `then` e_2 `else` e_3
		$\mid$	$e_1\ e_2$
		$\mid$	`recfun` $f :: (\tau_1 \to \tau_2)\ x = e$
			$\uparrow$ 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.

Syntax

$$
\begin{array}{llll}
\textit{Integers} & n & ::= & \cdots \\
\textit{Identifiers} & f, x & ::= & \cdots \\
\textit{Literals} & b & ::= & \texttt{True} \mid \texttt{False} \\
\textit{Types} & \tau & ::= & \mathrm{Bool} \mid \mathrm{Int} \mid \tau_1 \to \tau_2 \\
\textit{Infix Operators} & \circledast & ::= & \texttt{*} \mid \texttt{+} \mid \texttt{==} \mid \cdots \\
\textit{Expressions} & e & ::= & x \mid n \mid b \mid (e) \mid e_1 \circledast e_2 \\
 & & \mid & \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\
 & & \mid & e_1\ e_2 \\
 & & \mid & \texttt{recfun}\ f :: (\tau_1 \to \tau_2)\ x = e
\end{array}
$$

$\uparrow$ 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)

$$
\begin{array}{l}
\textbf{recfun}\ \mathit{divBy5} :: (\mathrm{Int} \rightarrow \mathrm{Int})\ x = \\
\quad \textbf{if}\ x < 5 \\
\qquad \textbf{then}\ 0 \\
\qquad \textbf{else}\ 1 + \mathit{divBy5}\ (x - 5)
\end{array}
$$

Example (Average Function)

$$
\begin{array}{l}
\textbf{recfun}\ \mathit{average} :: (\mathrm{Int} \rightarrow (\mathrm{Int} \rightarrow \mathrm{Int}))\ x = \\
\quad \textbf{recfun}\ \mathit{avX} :: (\mathrm{Int} \rightarrow \mathrm{Int})\ y = \\
\qquad (x + y)\ /\ 2
\end{array}
$$

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

We don't need no let

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

$$\textbf{let}\ x :: \tau_1 = e_1\ \textbf{in}\ e_2 :: \tau_2 \quad \equiv \quad (\textbf{recfun}\ f :: (\tau_1 \rightarrow \tau_2)\ x = e_2)\ e_1$$

We don't need no let

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

$$\textbf{let } x :: \tau_1 = e_1 \textbf{ in } e_2 :: \tau_2 \quad \equiv \quad (\textbf{recfun } f :: (\tau_1 \to \tau_2)\ x = e_2)\ e_1$$

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num`, `Lit`)
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then** t **else** e becomes (`If` c t e).
4. Function applications e_1 e_2 become explicit (`Apply` e_1 e_2).
5. **recfun** $f :: (\tau_1 \rightarrow \tau_2)\ x = e$ becomes (`Recfun` τ_1 τ_2 f x e).
6. Variable usages are wrapped in a term (`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 $(f.\ x.\ e)$).

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num`, `Lit`)
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then** t **else** e becomes (`If` c t e).
4. Function applications e_1 e_2 become explicit (`Apply` e_1 e_2).
5. **recfun** $f :: (\tau_1 \rightarrow \tau_2)\ x = e$ becomes (`Recfun` τ_1 τ_2 f x e).
6. Variable usages are wrapped in a term (`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 $(f.\ x.\ e)$).

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num`, `Lit`)
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then** t **else** e becomes (`If` c t e).
4. Function applications e_1 e_2 become explicit (`Apply` e_1 e_2).
5. **recfun** $f :: (\tau_1 \rightarrow \tau_2)\ x = e$ becomes (`Recfun` τ_1 τ_2 f x e).
6. Variable usages are wrapped in a term (`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 $(f.\ x.\ e)$).

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num`, `Lit`)
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then** t **else** e becomes (`If` c t e).
4. Function applications e_1 e_2 become explicit (`Apply` e_1 e_2).
5. **recfun** $f :: (\tau_1 \to \tau_2)\ x = e$ becomes (`Recfun` τ_1 τ_2 f x e).
6. Variable usages are wrapped in a term (`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 $(f.\ x.\ e)$).

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num`, `Lit`)
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then** t **else** e becomes (`If` c t e).
4. Function applications e_1 e_2 become explicit (`Apply` e_1 e_2).
5. **recfun** $f :: (\tau_1 \rightarrow \tau_2)\ x = e$ becomes (`Recfun` τ_1 τ_2 f x e).
6. Variable usages are wrapped in a term (`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 $(f.\ x.\ e)$).

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num`, `Lit`)
2. Operators like $a + b$ become (`Plus` a b).
3. **if** c **then** t **else** e becomes (`If` c t e).
4. Function applications $e_1\ e_2$ become explicit (`Apply` e_1 e_2).
5. **recfun** $f :: (\tau_1 \rightarrow \tau_2)\ x = e$ becomes (`Recfun` τ_1 τ_2 f x e).
6. Variable usages are wrapped in a term (`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 $(f.\ x.\ e)$).

Abstract Syntax

Moving to first order abstract syntax, we get:

1. Things like numbers and boolean literals are wrapped in terms (`Num`, `Lit`)
2. Operators like $a + b$ become ($\texttt{Plus}\ a\ b$).
3. **if** c **then** t **else** e becomes ($\texttt{If}\ c\ t\ e$).
4. Function applications $e_1\ e_2$ become explicit ($\texttt{Apply}\ e_1\ e_2$).
5. **recfun** $f :: (\tau_1 \to \tau_2)\ x = e$ becomes ($\texttt{Recfun}\ \tau_1\ \tau_2\ f\ x\ e$).
6. Variable usages are wrapped in a term ($\texttt{Var}\ x$).

What changes when we move to higher order abstract syntax?

1. `Var` terms go away – we use the meta-language's variables.
2. ($\texttt{Recfun}\ \tau_1\ \tau_2\ f\ x\ e$) now uses meta-language abstraction: ($\texttt{Recfun}\ \tau_1\ \tau_2\ (f.\ x.\ e)$).

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.

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:

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

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:

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

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

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

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

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

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

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

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

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

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

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

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

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

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

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

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

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

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

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

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

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

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

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

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

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

Let's implement a *type checker*.

Static Semantics

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

$$\frac{\Gamma \vdash e_1 : \mathrm{Int} \qquad \Gamma \vdash e_2 : \mathrm{Int}}{\Gamma \vdash (\texttt{Plus}\ e_1\ e_2) : \mathrm{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{If}\ e_1\ e_2\ e_3) : \tau}$$

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

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

Let's implement a *type checker*.

Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states: All well typed expressions.

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

Evaluation of built-in operations:

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

(and so on as per arithmetic expressions)

Dynamic Semantics

Structural Operational Semantics (Small-Step)

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

Evaluation of built-in operations:

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

(and so on as per arithmetic expressions)

Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states: All well typed expressions.

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

Evaluation of built-in operations:

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

(and so on as per arithmetic expressions)

Dynamic Semantics

Structural Operational Semantics (Small-Step)

Initial states: All well typed expressions.

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

Evaluation of built-in operations:

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

(and so on as per arithmetic expressions)

Specifying If

$$\frac{e_1 \mapsto e_1'}{(\texttt{If}\ e_1\ e_2\ e_3) \mapsto (\texttt{If}\ e_1'\ e_2\ e_3)}$$

$$\frac{}{(\texttt{If}\ (\texttt{Lit True})\ e_2\ e_3) \mapsto e_2}$$

$$\frac{}{(\texttt{If}\ (\texttt{Lit False})\ e_2\ e_3) \mapsto e_3}$$

How about Functions?

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

Evaluating function application requires us to:

1. Evaluate the left expression to get the function being applied
2. Evaluate the right expression to get the argument value
3. Evaluate the function's body, after supplying substitutions for the abstracted variables.

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

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

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

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 the right expression to get the argument value
3. Evaluate the function's body, after supplying substitutions for the abstracted variables.

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

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

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

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 the right expression to get the argument value
3. Evaluate the function's body, after supplying substitutions for the abstracted variables.

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

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

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

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 the right expression to get the argument value
3. Evaluate the function's body, after supplying substitutions for the abstracted variables.

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

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

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

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 the right expression to get the argument value
3. Evaluate the function's body, after supplying substitutions for the abstracted variables.

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

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

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