Computer Science 131
Programming Languages

September 12, 2000

Static and Dynamic Semantics

Syntax vs. Semantics

- Syntax
  - What phrases may occur where.
- Semantics:
  - What the phrases mean when put together.
  - What should the answer be?
  - How should execution proceed?

Purposes of a Language Definition

- For the programmer
  - Understanding the language
  - Reasoning about programs
- For the language implementor
  - Understanding what correct implementations must/may do
  - Deciding whether program transformations are correct
  - Facilitate multiple (compatible) implementations
- For the language designer
  - Recording design decisions
  - Understanding interaction between language features
  - Reasoning about the language

Formal Definitions?

- Why a formal semantics?
  - Informal definitions invariably contain ambiguities or errors.
  - Facilitates reasoning about the language
  - Facilitates reasoning about programs in the language
  - Facilitates reasoning about program transformations
  - May permit automatic generation of implementations

- Truth in advertising: still very hard to give a formal description of a full, real language
  - But can handle quite large subsets
  - Active research topic

Simple Arithmetic Expressions

- Abstract Syntax

$\boldsymbol{v}$	$::= \underline{\boldsymbol{n}}$	(values)
$\boldsymbol{e}$	$::= \boldsymbol{v} \mid \boldsymbol{e} + \boldsymbol{e}$	(expressions)
$\boldsymbol{p}$	$::= \boldsymbol{e}$	(programs)

- What does a program in this language mean?

Two Approaches to Formal Semantics

- Denotational semantics
  - The meaning of every expression is a mathematical object (a number, a function, etc.)
  - Compositionality: meaning of an expression is a function of the meanings of its sub-expressions.
    - E.g., the meaning of a while loop

      `while` ***`b`*** **`do`** ***`e`***

      is calculated from the meanings of the guard expression ***b*** and of the loop body ***e***
  - Two expressions with the same meaning are interchangeable.

Two Approaches to Formal Semantics

- Operational semantics
  - Defines evaluation of (complete) programs
  - High-level specification of an interpreter
  - We can choose the level of abstraction
    - Which (if any) low-level machine details we want to describe
      - Data representations
      - Memory management
    - Which concepts considered primitive

Small-step Operational Semantics

- Defines a relation $\rightarrow$ between programs corresponding to one step of execution

$$\texttt{(3 + 4) + (5 + 3)} \quad \rightarrow \quad \texttt{7 + (5 + 3)}$$

$$\texttt{7 + (5 + 3)} \quad \rightarrow \quad \texttt{7 + 8}$$

$$\texttt{7 + 8} \quad \rightarrow \quad \texttt{15}$$

Small-step Operational Semantics

- We define the relation $\rightarrow^*$ to be the reflexive, transitive closure of $\rightarrow$.

  $\boldsymbol{e} \rightarrow^* \boldsymbol{e'}$ if $\boldsymbol{e} \rightarrow \boldsymbol{e'}$.

  $\boldsymbol{e} \rightarrow^* \boldsymbol{e}$ always.

  $\boldsymbol{e} \rightarrow^* \boldsymbol{e''}$ if $\boldsymbol{e} \rightarrow^* \boldsymbol{e'}$ and $\boldsymbol{e'} \rightarrow^* \boldsymbol{e''}$ for some $\boldsymbol{e'}$.

Small-step Operational Semantics

- We define the relation $\rightarrow^*$ to be the reflexive, transitive closure of $\rightarrow$.

  $\boldsymbol{e} \rightarrow^* \boldsymbol{e'}$ if $\boldsymbol{e} \rightarrow \boldsymbol{e'}$.

  $\boldsymbol{e} \rightarrow^* \boldsymbol{e}$ always.

  $\boldsymbol{e} \rightarrow^* \boldsymbol{e''}$ if $\boldsymbol{e} \rightarrow^* \boldsymbol{e'}$ and $\boldsymbol{e'} \rightarrow^* \boldsymbol{e''}$ for some $\boldsymbol{e'}$.

- For example, `(3+4)-(5-3)` $\rightarrow^*$ `5`
- A program ***p*** terminates with value ***v*** if $\boldsymbol{p} \rightarrow^* \boldsymbol{v}$
- A program ***p*** fails to terminate if

  $p \rightarrow p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow \ldots$

Small-step Operational Semantics

- Equivalent definition: inference rules (hopefully familiar from CS 80)

$$\frac{e \rightarrow e'}{e \rightarrow^* e'} \qquad \frac{}{e \rightarrow^* e}$$

$$\frac{e \rightarrow^* e' \qquad e' \rightarrow^* e''}{e \rightarrow^* e''}$$

Structured Operational Semantics

$$\frac{}{\underline{\mathbf{n}}_1 + \underline{\mathbf{n}}_2 \to \underline{\mathbf{n}_1 + \mathbf{n}_2}}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \qquad \frac{e_2 \to e_2'}{e_1 + e_2 \to e_1 + e_2'}$$

Concurrency and Non-determinism?

$$\frac{}{\underline{\mathtt{n}}_1 \ + \ \underline{\mathtt{n}}_2 \ \rightarrow \ \underline{\mathtt{n}_1\mathtt{+}\mathtt{n}_2}}$$

$$\frac{e_1 \ \rightarrow \ e_1'}{e_1 \ + \ e_2 \ \rightarrow \ e_1' \ + \ e_2} \qquad \frac{e_2 \ \rightarrow \ e_2'}{e_1 \ + \ e_2 \ \rightarrow \ e_1 \ + \ e_2'}$$

$$(\underline{3} \ + \ \underline{4}) \ + \ (\underline{5} \ + \ \underline{3}) \ \rightarrow \ \underline{7} \ + \ (\underline{5} \ + \ \underline{3})$$
$$(\underline{3} \ + \ \underline{4}) \ + \ (\underline{5} \ + \ \underline{3}) \ \rightarrow \ (\underline{3} \ + \ \underline{4}) \ + \ \underline{8}$$

Left-to-Right Evaluation

$$\frac{}{\underline{n}_1 + \underline{n}_2 \to \underline{n_1 + n_2}}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2}$$

$$\cancel{\frac{e_2 \to e_2'}{e_1 + e_2 \to e_1 + e_2'}}$$

$$\frac{e_2 \to e_2'}{v_1 + e_2 \to v_1 + e_2'}$$

Adding Booleans

- Abstract Syntax

$$
\begin{array}{lll}
\boldsymbol{v} ::= \underline{\mathtt{n}} \mid \underline{\mathtt{tt}} \mid \underline{\mathtt{ff}} & & \text{(values)} \\
\boldsymbol{e} ::= \boldsymbol{v} \mid \boldsymbol{e} + \boldsymbol{e} \mid \boldsymbol{e} < \boldsymbol{e} & & \text{(expressions)} \\
\quad\quad \mid \mathtt{if}\ \boldsymbol{e}\ \mathtt{then}\ \boldsymbol{e}\ \mathtt{else}\ \boldsymbol{e} & & \\
\boldsymbol{p} ::= \boldsymbol{e} & & \text{(programs)}
\end{array}
$$

- For example,

$\mathtt{if\ (if}\ \underline{\mathtt{3}} < \underline{\mathtt{5}}\ \mathtt{then}\ \underline{\mathtt{ff}}\ \mathtt{else}\ \underline{\mathtt{tt}}\mathtt{)\ then}\ \underline{\mathtt{1}}\mathtt{+}\underline{\mathtt{2}}\ \mathtt{else}\ \underline{\mathtt{7}}\mathtt{+}\underline{\mathtt{8}}$

Dynamic Semantics

$$\frac{}{\underline{n}_1 + \underline{n}_2 \to \underline{n_1+n_2}}$$

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \qquad \frac{e_2 \to e_2'}{v_1 + e_2 \to v_1 + e_2'}$$

$$\frac{}{\underline{n}_1 < \underline{n}_2 \to \underline{n_1<n_2}}$$

$$\frac{e_1 \to e_1'}{e_1 < e_2 \to e_1' < e_2} \qquad \frac{e_2 \to e_2'}{v_1 < e_2 \to v_1 < e_2'}$$

Dynamic Semantics

$$\frac{\boldsymbol{e_1} \rightarrow \boldsymbol{e_1}'}{\texttt{if } \boldsymbol{e_1} \texttt{ then } \boldsymbol{e_2} \texttt{ else } \boldsymbol{e_3} \rightarrow \texttt{if } \boldsymbol{e_1}' \texttt{ then } \boldsymbol{e_2} \texttt{ else } \boldsymbol{e_3}}$$

What other rules?

Dynamic Semantics

$$\frac{e_1 \rightarrow e_1'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightarrow \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3}$$

$$\frac{}{\texttt{if } \underline{\texttt{tt}} \texttt{ then } e_2 \texttt{ else } e_3 \rightarrow e_2}$$

$$\frac{}{\texttt{if } \underline{\texttt{ff}} \texttt{ then } e_2 \texttt{ else } e_3 \rightarrow \underline{e}_3}$$

Stuck programs

- We now have the possibility of syntactically well-formed programs that have not yielded a value, but which cannot make progress!

```
3 + tt
if tt < ff then 3 else 5
```

Stuck programs

- We now have the possibility of syntactically well-formed programs that have not yielded a value, but which cannot make progress!

```
3 + tt
if tt < ff then 3 else 5
```

- Answer 1: who cares?
  - Implementation-dependent behavior
  - Or, program should stop with error.

Stuck programs

- We now have the possibility of syntactically well-formed programs that have not yielded a value, but which cannot make progress!

```
3 + tt
if tt < ff then 3 else 5
```

- Answer 1: who cares?
  - Implementation-dependent behavior
  - Or, program should stop with error.
- Answer 2: a type system to prevent this

Typing

- We define a collection of types. For now, just
  t **::= Int | Bool**

- We define the relation ***e*** **:** ***t***
  - A collection of inference rules is used to define when this relation holds.

- The type system is frequently called the "static semantics" of the language.

Static Semantic Rules

$$\frac{}{\underline{\mathtt{n}} : \mathtt{Int}} \qquad \frac{}{\underline{\mathtt{tt}} : \mathtt{Bool}} \qquad \frac{}{\underline{\mathtt{ff}} : \mathtt{Bool}}$$

Static Semantic Rules

$$\frac{}{\underline{\mathtt{n}} : \mathtt{Int}} \qquad \frac{}{\underline{\mathtt{tt}} : \mathtt{Bool}} \qquad \frac{}{\underline{\mathtt{ff}} : \mathtt{Bool}}$$

$$\frac{???}{e_1 + e_2 : \mathtt{Int}}$$

Static Semantic Rules

$$\frac{}{\underline{\mathtt{n}} : \mathtt{Int}} \qquad \frac{}{\underline{\mathtt{tt}} : \mathtt{Bool}} \qquad \frac{}{\underline{\mathtt{ff}} : \mathtt{Bool}}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 + e_2 : \mathtt{Int}}$$

Static Semantic Rules

$$\frac{}{\underline{\mathtt{n}} : \mathtt{Int}} \qquad \frac{}{\underline{\mathtt{tt}} : \mathtt{Bool}} \qquad \frac{}{\underline{\mathtt{ff}} : \mathtt{Bool}}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 + e_2 : \mathtt{Int}}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 < e_2 : \mathtt{Bool}}$$

Static Semantic Rules

$$\frac{}{\underline{\mathtt{n}} : \mathtt{Int}} \qquad \frac{}{\underline{\mathtt{tt}} : \mathtt{Bool}} \qquad \frac{}{\underline{\mathtt{ff}} : \mathtt{Bool}}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 + e_2 : \mathtt{Int}}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 < e_2 : \mathtt{Bool}}$$

$$\frac{e_1 : ? \qquad e_2 : ? \qquad e_3 : ?}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : ?}$$

Static Semantic Rules

$$\frac{}{\underline{\mathtt{n}} : \mathtt{Int}} \qquad \frac{}{\underline{\mathtt{tt}} : \mathtt{Bool}} \qquad \frac{}{\underline{\mathtt{ff}} : \mathtt{Bool}}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 + e_2 : \mathtt{Int}}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 < e_2 : \mathtt{Bool}}$$

$$\frac{e_1 : \mathtt{Bool} \qquad e_2 : \mathtt{?} \qquad e_3 : \mathtt{?}}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \mathtt{?}}$$

Static Semantic Rules

$$\frac{}{\underline{\mathtt{n}} : \mathtt{Int}} \qquad \frac{}{\underline{\mathtt{tt}} : \mathtt{Bool}} \qquad \frac{}{\underline{\mathtt{ff}} : \mathtt{Bool}}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 + e_2 : \mathtt{Int}}$$

$$\frac{e_1 : \mathtt{Int} \qquad e_2 : \mathtt{Int}}{e_1 < e_2 : \mathtt{Bool}}$$

$$\frac{e_1 : \mathtt{Bool} \qquad e_2 : t \qquad e_3 : t}{\mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : t}$$

Claim: Type Soundness

- Well-typed programs (i.e., programs that have some type) can't get stuck
  - That is, must either eventually reach a value or fail to terminate
- Why is this important? How to prove this?
  - See next lecture

But...

- Some programs wouldn't get stuck but still don't typecheck

  `(if ff then tt else 4) + 1`

- For any interesting language, a type system preventing all bad programs also rejects programs that would run without problems.
- Research topic: type systems that catch as many errors as possible, but don't reject useful programs

Adding Local Definitions

- Abstract Syntax

$$
\begin{array}{lll}
\boldsymbol{v} ::= \underline{\mathtt{n}} \mid \underline{\mathtt{tt}} \mid \underline{\mathtt{ff}} & & \text{(values)} \\
\boldsymbol{e} ::= \boldsymbol{v} \mid \boldsymbol{e} + \boldsymbol{e} \mid \boldsymbol{e} < \boldsymbol{e} & & \text{(expressions)} \\
\quad\quad \mid \mathtt{if}\ \boldsymbol{e}\ \mathtt{then}\ \boldsymbol{e}\ \mathtt{else}\ \boldsymbol{e} & & \\
\quad\quad \mid \boldsymbol{x} & & \\
\quad\quad \mid \mathtt{let}\ \boldsymbol{x}\ \mathtt{be}\ \boldsymbol{e}\ \mathtt{in}\ \boldsymbol{e} & & \\
\boldsymbol{p} ::= \boldsymbol{e} & & \text{(programs)}
\end{array}
$$

Scoping For This Language

- In the expression **`let x be`** $\boldsymbol{e_1}$ **`in`** $\boldsymbol{e_2}$
  - The variable x is bound
  - The scope of x (where it can be referenced) is $\boldsymbol{e_2}$

Scoping For This Language

- All α-equivalent expressions are identified
  - E.g., the following are the same expressions

    `let x be 3 in x + y`

    `let z be 3 in z + y`
  - But not

    `let y be 3 in y + y`

Changes to Dynamic Semantics

- Add the two rules:

$$\frac{e_1 \to e_1'}{\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2 \to \texttt{let}\ x\ \texttt{be}\ e_1'\ \texttt{in}\ e_2}$$

$$\frac{}{\texttt{let}\ x\ \texttt{be}\ v_1\ \texttt{in}\ e_2 \to e_2[x \to v_1]}$$

Changes to Dynamic Semantics

- What if we had just this single rule instead?

$$\frac{}{\texttt{let}\ x\ \texttt{be}\ e_1\ \texttt{in}\ e_2 \;\rightarrow\; e_2[x \rightarrow e_1]}$$

- Consider **`let x be 1+2 in x+x`**

Changes to the Static Semantics

- Typing is now *context-sensitive*
  - What is the type of **x** ?
  - Is **x+<u>3</u>** well-typed?

Changes to the Static Semantics

- Typing is now *context-sensitive*
  - What is the type of **`x`** ?
  - Is **`x+3`** well-typed?
    - Yes, in the program **`let x be 4 in x+3`**
    - No, in the program **`let x be tt in x+3`**

- Need to know the types of *free* variables

Changes to the Static Semantics

- 3-place typing relation $\Gamma \vdash$ ***e* : *t***
  - Here Γ is a *type environment*
    - Mapping from variables to their types
  - Type environments will be written as a list
    - e.g., **x:Int, y:Bool, z:Int**
  - The notation Γ(**x**) gives the type of **x** (lookup)
  - The notation

    Γ, **x:Int**

    is the extension of Γ that maps **x** to **Int** (insert)
  - We write $\vdash$ ***e* : *t*** when type env. is empty

Static Semantics

$$\frac{}{\Gamma \vdash \underline{\mathbf{n}} : \mathbf{Int}} \qquad \frac{}{\Gamma \vdash \underline{\mathbf{tt}} : \mathbf{Bool}} \qquad \frac{}{\Gamma \vdash \underline{\mathbf{ff}} : \mathbf{Bool}}$$

Static Semantics

$$\frac{}{\Gamma \vdash \underline{\mathtt{n}} : \mathtt{Int}} \qquad \frac{}{\Gamma \vdash \underline{\mathtt{tt}} : \mathtt{Bool}} \qquad \frac{}{\Gamma \vdash \underline{\mathtt{ff}} : \mathtt{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Int} \qquad \Gamma \vdash e_2 : \mathtt{Int}}{\Gamma \vdash e_1 + e_2 : \mathtt{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Int} \qquad \Gamma \vdash e_2 : \mathtt{Int}}{\Gamma \vdash e_1 < e_2 : \mathtt{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathtt{Bool} \qquad \Gamma \vdash e_2 : \mathtt{t} \qquad \Gamma \vdash e_3 : \mathtt{t}}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \mathtt{t}}$$

Static Semantics:
The Interesting Rules

$$\frac{}{\Gamma \vdash \boldsymbol{x} \ \ \textbf{:} \ \ \textbf{?}}$$

Static Semantics: The Interesting Rules

$$\frac{}{\Gamma \vdash \boldsymbol{x} \;:\; \Gamma(\boldsymbol{x})}$$

Static Semantics:
The Interesting Rules

$$\frac{}{\Gamma \vdash \boldsymbol{x} \texttt{ : } \Gamma(\boldsymbol{x})}$$

$$\frac{\texttt{???}}{\Gamma \vdash \texttt{let } \boldsymbol{x} \texttt{ be } \boldsymbol{e}_1 \texttt{ in } \boldsymbol{e}_2 \texttt{ : } \boldsymbol{t}_2}$$

Static Semantics: The Interesting Rules

$$\frac{}{\Gamma \vdash \boldsymbol{x} \;\mathtt{:}\; \Gamma(\boldsymbol{x})}$$

$$\frac{\Gamma \vdash \boldsymbol{e_1} \;\mathtt{:}\; \boldsymbol{t_1} \qquad \Gamma,\ \mathtt{x}\mathtt{:}\boldsymbol{t_1} \vdash \boldsymbol{e_2} \;\mathtt{:}\; \boldsymbol{t_2}}{\Gamma \vdash \mathtt{let}\ \boldsymbol{x}\ \mathtt{be}\ \boldsymbol{e_1}\ \mathtt{in}\ \boldsymbol{e_2} \;\mathtt{:}\; \boldsymbol{t_2}}$$

Claim: Type Soundness

- If a program is well-typed *in the empty type environment* (i.e., $\vdash$ ***p* : *t*** for some ***t***) then it cannot get stuck

  – Note that such programs must have no free variables.

Adding Recursive Functions

- Abstract Syntax

$$
\begin{array}{lll l}
v & ::= & \underline{\mathtt{n}} \mid \underline{\mathtt{tt}} \mid \underline{\mathtt{ff}} & \text{(values)} \\
 & \mid & \mathtt{fun}\ x(x:t):t\ \mathtt{is}\ e & \\
e & ::= & v \mid e + e \mid e < e & \text{(expressions)} \\
 & \mid & \mathtt{if}\ e\ \mathtt{then}\ e\ \mathtt{else}\ e & \\
 & \mid & x \mid \mathtt{let}\ x\ \mathtt{be}\ e\ \mathtt{in}\ e & \\
 & \mid & e\ e & \\
p & ::= & e & \text{(programs)}
\end{array}
$$

WARNING

- This is not SML!
- The expression **fun f(x:t_1):t_2 is *e***
  - is a function value, like SML's **fn x => *e*** except that it can be recursive.
  - The equivalent SML code would be

```
let
   fun f(x:t1):t2 = e
in
   f
end
```

Scoping

- In the expression **`fun f(x:`**$\boldsymbol{t_1}$**`):`**$\boldsymbol{t_2}$ **`is`** ***e***
  - Both **`f`** and **`x`** as bound variables
    - The name of the argument is **`x`**
    - The *local* name of the function is **`f`**
  - Both **`f`** and **`x`** have ***e*** as their scope

α-conversion

- The following are the exact same programs:

```
fun f(x:Int):Int is f(h(x+y))
fun g(z:Int):Int is f(h(x+y))
```

- But not:

```
fun h(y:Int):Int is h(h(y+y))
```

Example Code

```
(fun f(x:Int):Int is x+1) 3
```

Example Code

```
let
    fact be (fun g(y:Int):Int is
              if (y=0) then 1
               else y*(g(y-1)))
in
    (fact 3) + (fact 4)
```

Dynamic Semantics

- Add the rules

$$\frac{e_1 \rightarrow e_1'}{e_1 \ e_2 \rightarrow e_1' \ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1 \ e_2 \rightarrow v_1 \ e_2'}$$

Static Semantics

```
t ::= Int | Bool | t -> t
```

Static Semantics

$$\boldsymbol{t} \texttt{ ::= Int | Bool | } \boldsymbol{t} \texttt{ -> } \boldsymbol{t}$$

$$\frac{\Gamma \vdash e_1 : \texttt{?} \qquad \Gamma \vdash e_2 : \texttt{?}}{\Gamma \vdash e_1\ e_2 : t}$$

Static Semantics

t **::= Int | Bool |** t **->** t

$$\frac{\Gamma \vdash e_1 : t_2 \texttt{->} t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ e_2 : t}$$

Static Semantics

t **::= Int | Bool |** t **->** t

$$\frac{\Gamma \vdash e_1 : t_2 \texttt{->} t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ e_2 : t}$$

$$\frac{\texttt{???}}{\Gamma \vdash \texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e : \texttt{???}}$$

Static Semantics

t ::= **Int** | **Bool** | t -> t

$$\frac{\Gamma \vdash e_1 : t_2 \texttt{->} t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ e_2 : t}$$

$$\frac{???}{\Gamma \vdash \texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e : t_1 \texttt{->} t_2}$$

Static Semantics

$$t ::= \texttt{Int} \mid \texttt{Bool} \mid t \texttt{ -> } t$$

$$\frac{\Gamma \vdash e_1 : t_2\texttt{->}t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1\ e_2 : t}$$

$$\frac{\Gamma, x{:}t_1, f{:}t_1\texttt{->}t_2 \vdash e : t_2}{\Gamma \vdash \texttt{fun } f(x{:}t_1){:}t_2 \texttt{ is } e : t_1\texttt{->}t_2}$$

Dynamic Semantics

- Add the rules

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}$$

Dynamic Semantics

- Add the rules

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ e_2 \to v_1\ e_2'}$$

$$\frac{}{(\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2)\ v \to\ ???}$$

Dynamic Semantics

- Add the rules

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}$$

$$\frac{}{\texttt{(fun } f(x{:}t_1){:}t_2 \texttt{ is } e_2\texttt{)}\ v \rightarrow e_2[x \rightarrow ?, f \rightarrow ?]}$$

Dynamic Semantics

- Add the rules

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}$$

$$\frac{}{(\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2)\ v \rightarrow (e_2[x \rightarrow v])[f \rightarrow ?]}$$

Dynamic Semantics

- Add the rules

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}$$

$$\frac{}{\begin{array}{l}(\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e)\ v \ \rightarrow \\ \quad (e_2[x \rightarrow v])[f \rightarrow (\texttt{fun}\ f(x{:}t_1){:}t_2\ \texttt{is}\ e_2)]\end{array}}$$