**Review: Dynamic Semantics**

```
n_1 + n_2 \rightarrow n_1 \otimes n_2

n_1 \otimes n_2 \rightarrow n_1 \otimes n_2

n_1 \leq n_2 \rightarrow n_1 \leq n_2

v_1 \rightarrow v_1'

e_1 \rightarrow e_1'
e_2 \rightarrow e_2'
e_1 + e_2 \rightarrow e_1' + e_2'
e_1 < e_2 \rightarrow e_1 < e_2'

v + e_2 \rightarrow v + e_2'

v \leq e_2 \rightarrow v \leq e_2'

if e_1 then e_2 else e_3 \rightarrow if e_1' then e_2 else e_3

if tt then e_2 else e_3 \rightarrow e_2

if ff then e_2 else e_3 \rightarrow e_3
```
Review: Dynamic Semantics

\[ \text{let } x \text{ be } e_1 \text{ in } e_2 \rightarrow \text{let } x \text{ be } e_1' \text{ in } e_2 \]
\[ \text{let } x \text{ be } v_1 \text{ in } e_2 \rightarrow e_2[x \rightarrow v_1] \]

Examples

- \text{let } x \text{ be } 3+4 \text{ in } (1+2)+x \rightarrow \text{let } x \text{ be } 7 \text{ in } (1+2)+x \rightarrow (1+2)+7 \rightarrow 3+7 \rightarrow 10
- \text{let } x \text{ be } 1+1 \text{ in } x+x \rightarrow \rightarrow \rightarrow

Review: Alternative Dynamic Semantics

• What if instead we had this single rule for let?

\[ \text{let } x \text{ be } e_1 \text{ in } e_2 \rightarrow e_2[x \rightarrow e_1] \]

• let \( x \) be 1+1 in x+x

→
→
→
→

Adding Function Values

• Abstract Syntax

\[ v := n \mid tt \mid ff \]
\[ \mid \text{fix } f(x) \text{ is } e \]
\[ e := v \mid e + e \mid e \leq e \]
\[ \mid \text{if } e \text{ then } e \text{ else } e \]
\[ \mid x \mid \text{let } x \text{ be } e \text{ in } e \]
\[ \mid e \ e \]

(values)

(expressions)
Recursive Function Values

• The function value
  \[ \text{fix } f(x) \text{ is } e \]
corresponds roughly to the SML code
  \begin{verbatim}
  let fun f(x)=e
  in f
  end
  \end{verbatim}

• In particular, note that the scope of \( f \) is \( e \) and the
  scope of \( x \) is \( e \), and that's it.
• This does not permit other code to refer to this
  function as \( f \)!

Example Expressions

\( \text{(fix } f(x) \text{ is } x+1) \ 3 \)
\( \text{(fix } f(x) \text{ is } f(x+1)) \ 3 \)

let succ be \((\text{fix } f(x) \text{ is } x+1)\)
in let n be 4 in
\( \text{(succ n) + (succ (n+1))} \)

let fib be
\( \text{fix } g(x) \text{ if } x < 0 \text{ then } 1 \text{ else } \)
  \( \text{if } x < 1 \text{ then } 1 \text{ else } \)
  \( g(x+(-1)) + g(x+(-2)) \)
in
\( \text{fib (1+1)} \)
\( (* \text{ not } g(1+1) \ ! \ *)) \)

Conventions

• The body of a function is assumed to extend as far
  as possible:

\[ \text{fix } f(x) \text{ is } x+x \]
\[ = \text{fix } f(x) \text{ is } (x+x) \]
\[ \neq (\text{fix } f(x) \text{ is } x)+x \]

Dynamic Semantics

• Add the rules

\[ e_1 \rightarrow e_1' \]
\[ e_2 \rightarrow e_2' \]
\[ e_1 \ e_2 \rightarrow e_1' \ e_2' \]
\[ v_1 \ e_2 \rightarrow v_1 \ e_2' \]

\[ (\text{fix } f(x) \text{ is } e) \ v \rightarrow \]
\[ e[v \rightarrow [f \rightarrow \text{fix } f(x) \text{ is } e]] \]
Back to Stuck Programs

• Recall: some programs have not reached a final state, but cannot make progress
  \[3 + \text{tt}\]
  \[\text{if tt} < \text{ff} \text{ then } 3 \text{ else } 5\]
• Such programs are "about to go wrong"
  - About to apply an operation using the wrong sort of operands
• We can prevent such problems with a type system
  - Let's start with just the type system for arithmetic and conditionals.

A Simple Static Semantics

• We start with just two types
  \[t ::= \text{int} \mid \text{bool}\]
• We will define a typing relation
  \[e : t\]
• A type system is frequently called the "static semantics" of the language.

Static Semantic Rules

\[\frac{n : \text{int} \quad \text{tt} : \text{bool} \quad \text{ff} : \text{bool}}{\frac{\quad \frac{\quad \frac{\quad e_1 + e_2 :}{\quad e_1 < e_2 :}}{\quad \text{if } e_1 \text{ then } e_2 \text{ else } e_3 :}}{\text{Claim: Type Soundness}}\]

• 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
  
  \[(\text{if } \text{ff} \text{ then } \text{tt} \text{ else } 4) + 1\]

• 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

Changes to the Static Semantics

• Well-typedness is now context-sensitive
  - What is the type of \( x \) ?
  - Is \( x+3 \) well-typed?

  \[
  \begin{align*}
  &\text{let } x \text{ be } 4 \text{ in } x+3 \\
  &\text{let } x \text{ be } \text{tt} \text{ in } x+3
  \end{align*}
  \]

• To determine types we need to know the types of the free variables

Conditional Judgments

• The typing judgments are now conditional
  
  "If \( x : \text{int} \) then \( x+3 : \text{int} \)"
  "If \( x : \text{bool} \) then if \( x \) then 3 else 4 : \text{int}"

• Given assumptions about the types of variables, we can conclude code is well-typed

  \[
  \begin{align*}
  &x:\text{int} \vdash x+3 : \text{int} \\
  &x:\text{bool} \vdash \text{if } x \text{ then } 3 \text{ else } 4 : \text{int}
  \end{align*}
  \]

Typing Environments

• An environment is a lookup table for variables
  - A type environment associates variables with types

• Notation
  - We use \( \Gamma \) to denote an arbitrary type environment.
  - Type environments can be written as a list
    - e.g., \( x:\text{int}, y:\text{bool}, z:\text{int} \)
  - The notation \( \Gamma(x) \) gives the type of \( x \)
  - The notation \( \Gamma', x:\text{int} \)
    - is the extension of \( \Gamma \) that maps \( x \) to \text{int}
Updated Static Semantics

\[
\begin{align*}
& \Gamma \vdash n : \text{int} & \Gamma \vdash \text{tt} : \text{bool} & \Gamma \vdash \text{ff} : \text{bool} \\
& \Gamma \vdash e_1 : \text{int} & \Gamma \vdash e_2 : \text{int} \\
& \Gamma \vdash e_1 + e_2 : \text{int} \\
& \Gamma \vdash e_1 : \text{int} & \Gamma \vdash e_2 : \text{int} \\
& \Gamma \vdash e_1 < e_2 : \text{bool} \\
& \Gamma \vdash e_1 : \text{bool} & \Gamma \vdash e_2 : t & \Gamma \vdash e_3 : t \\
& \Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \\
\end{align*}
\]

The Interesting Rules

\[
\begin{align*}
& \Gamma \vdash x : \Gamma(x) \\
& \Gamma \vdash e_1 : t_1 & \Gamma, x : t_1 \vdash e_2 : t_2 \\
& \Gamma \vdash \text{let } x \text{ be } e_1 \text{ in } e_2 : t_2 \\
\end{align*}
\]

Functions

\[
\begin{align*}
& t ::= \text{int} \mid \text{bool} \mid t \rightarrow t \\
& \Gamma \vdash e_1 : t_1 \rightarrow t & \Gamma \vdash e_2 : t_2 \\
& \Gamma \vdash e_1 \_ e_2 : t \\
& \Gamma, x : t_1, f : t_1 \rightarrow t_2 \vdash e : t_2 \\
& \Gamma \vdash \text{fix } f(x) \text{ is } e : t_1 \rightarrow t_2 \\
\end{align*}
\]