Today’s music: “Down to Earth” by Peter Gabriel from the WALL-E soundtrack
Review

Previously in 3110:
• simple interpreter for expression language:
  – abstract syntax tree (AST)
  – small-step, substitution model of evaluation
  – parser and lexer
• formal syntax: BNF

Today:
• Formal dynamic semantics:
  – small-step, substitution model
  – large-step, environment model
• Formal static semantics
Review: Notation

• The interpreter code we've written is one way of defining the syntax and semantics of a language.
• Programming language designers have another more compact notation that's independent of the implementation language of interpreter...
Review: Abstract syntax

\[ e ::= x \mid i \mid e_1 + e_2 \]
\[ \mid \text{let } x = e_1 \text{ in } e_2 \]

\(e, x, i\): *meta-variables* that stand for pieces of syntax
- \(e\): expressions
- \(x\): program variables
- \(i\): integers

\(\ ::= \) and \(\mid\) are *meta-syntax*: used to describe syntax of language

notation is called *Backus-Naur Form (BNF)* from its use by Backus and Naur in their definition of Algol-60
FORMAL DYNAMIC SEMANTICS
Dynamic semantics

Defined by a judgement:
\[ e \rightarrow e' \]
Read as \( e \) takes a single step to \( e' \)
e.g., \((5+2)+0 \rightarrow 7+0\)

Expressions continue to step until they reach a value
e.g., \((5+2)+0 \rightarrow 7+0 \rightarrow 7\)

Values are a syntactic subset of expressions:
\( v ::= i \)
Dynamic semantics

Reflexive, transitive closure of --> is written -->* 

e -->* e' read as e multisteps to e' or e evaluates to e'

e.g., (5+2) + 0 -->* 7

This style of definition is called a small-step semantics: based on taking single small steps
Dynamic semantics of expr. lang.

\[ e_1 + e_2 \rightarrow e_1' + e_2 \]
  \[ \text{if } e_1 \rightarrow e_1' \]

\[ v_1 + e_2 \rightarrow v_1 + e_2' \]
  \[ \text{if } e_2 \rightarrow e_2' \]

\[ v_1 + v_2 \rightarrow n \]
  \[ \text{if } n \text{ is the sum of } v_1 \text{ and } v_2 \]
Dynamic semantics of expr. lang.

\[
\text{let } x = e_1 \text{ in } e_2 \longrightarrow \text{let } x = e_1' \text{ in } e_2 \\
\quad \text{if } e_1 \longrightarrow e_1'
\]

\[
\text{let } x = v_1 \text{ in } e_2 \longrightarrow e_2\{v_1/x\}
\]

read \( e_2\{v_1/x\} \) as \( e_2 \) with \( v_1 \) substituted for \( x \) (as we implemented in \textit{subst})

so we call this the \textit{substitution model of evaluation}
Dynamic semantics of expr. lang.

if e1 then e2 else e3
--> if e1' then e2 else e3
    if e1 --> e1'

if true then e2 else e3 --> e2

if false then e2 else e3 --> e3
Dynamic semantics of expr. lang.

Values and variables do not single step:

\[ v \rightarrow^* v \]
\[ x \rightarrow^* x \]

But they do multistep:

\[ v \rightarrow^* v \]
\[ x \rightarrow^* x \]

because multistep includes 0 steps (i.e., it is the reflexive transitive closure of \( \rightarrow^* \))

- values don't step because they're done computing
- variables don't step because they're an error: we should never reach a variable; it should have already been substituted away
Scaling up to OCaml

Read notes on website: full dynamic semantics for essential sublanguage of OCaml:

\[e ::= x \mid e_1 \ e_2 \mid \text{fun } x \rightarrow e \]
\[\mid i \mid e_1 + e_2 \]
\[\mid (e_1, \ e_2) \mid \text{fst } e_1 \mid \text{snd } e_2 \]
\[\mid \text{Left } e \mid \text{Right } e \]
\[\mid \text{match } e \text{ with } \text{Left } x \rightarrow e_1 \mid \text{Right } y \rightarrow e_2 \]
\[\mid \text{let } x = e_1 \text{ in } e_2 \]

**Missing, unimportant:** other built-in types, records, lists, options, declarations, patterns in function arguments and let bindings, \textbf{if}

**Missing, important:** \textbf{let} \textbf{rec}
FORMAL STATIC SEMANTICS
Static semantics

Suppose we add Booleans, conjunction, and `if` expressions to language:

\[
e ::= \ldots \mid b \mid e_1 \&\& e_2 \\
\quad \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \\
v ::= \ldots \mid b
\]

Now we could get nonsensical expressions, e.g.,
- \(5 + \text{false}\)
- `if 5 then true else 0`

Need static semantics (type checking) to rule those out...
if expressions [from lec 2]

Syntax:

    if e1 then e2 else e3

Type checking:

    if e1 has type bool and e2 has type t and e3 has type t
    then if e1 then e2 else e3 has type t
Static semantics

Defined by a judgement:

\[ T \vdash e : t \]

- Read as in typing context \( T \), expression \( e \) has type \( t \)
- Turnstile \( \vdash \) can be read as "proves" or "shows"
- You’re already used to \( e : t \), because utop uses that notation
- Typing context is a dictionary mapping variable names to types
- The typing context is a new idea, but obviously needed to give types of variables in scope
Static semantics

e.g.,

\[ x : \text{int} \vdash x + 2 : \text{int} \]
\[ x : \text{int}, y : \text{int} \vdash x < y : \text{bool} \]
\[ \vdash 5 + 2 : \text{int} \]
Static semantics of ext. expr. lang.

\[ T \vdash i : \text{int} \]

\[ T \vdash b : \text{bool} \]

\[ T, x : t \vdash x : t \]
Static semantics of ext. expr. lang.

\[ T \ |- \ e_1 + e_2 : \text{int} \]
\[
\text{if} \quad T \ |- \ e_1 : \text{int}
\] \[ \quad \text{and} \quad T \ |- \ e_2 : \text{int} \]

\[ T \ |- \ e_1 \&\& e_2 : \text{bool} \]
\[
\text{if} \quad T \ |- \ e_1 : \text{bool}
\] \[ \quad \text{and} \quad T \ |- \ e_2 : \text{bool} \]
Static semantics of ext. expr. lang.

\[ T |- \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]
\[ \text{if } T |- e_1 : \text{bool} \]
\[ \text{and } T |- e_2 : t \]
\[ \text{and } T |- e_3 : t \]

\[ T |- \text{let } x : t_1 = e_1 \text{ in } e_2 : t_2 \]
\[ \text{if } T |- e_1 : t_1 \]
\[ \text{and } T, x : t_1 |- e_2 : t_2 \]
Interpreter for ext. expr. lang.

See interp3.ml in code for this lecture

1. Type checks expression
2. Evaluates expression
Purpose of type system

Ensure **type safety**: well-typed programs don't get **stuck**:
• haven't reached a value, and
• unable to evaluate further

Lemmas:
**Progress**: if $e$ has type $t$, then either $e$ is a value or $e$ can take a step.

**Preservation**: if $e$ has type $t$, and if $e$ takes a step to $e'$, then $e'$ has type $t$.

Type safety = progress + preservation

Proving type safety is a fun part of 4110
ANOTHER FORMAL DYNAMIC SEMANTICS
Dynamic semantics

Two different models of evaluation:

- **Small-step substitution model**: substitute value for variable in body of `let` expression
  - And in body of function, since `let x = e1 in e2` behaves the same as `(fun x -> e2) e1`
  - What we’ve done so far; good mental model for evaluation
  - Not really what OCaml does

- **Big-step environment model**: keep a data structure around that binds variables to values
  - What we’ll do now; also a good mental model
  - Much closer to what OCaml really does
Syntax

e ::= x | i | b
    | e1 + e2 | e1 && e2
    | let x = e1 in e2
    | if e1 then e2 else e3

v ::= i | b
New evaluation judgement

• *Big-step semantics*: we model just the reduction from the original expression to the final value

• Suppose $e \rightarrow e' \rightarrow \ldots \rightarrow v$

• We'll abstract that fact to $e \Rightarrow v$
  ─ forget about all the intermediate expressions
  ─ new notation means $e$ *evaluates (down) to* $v$, equiv. $e$ *takes a big step to* $v$
  ─ textbooks use down arrows: $e \downarrow v$

• **Goal**: $e \Rightarrow v$ if and only if $e \rightarrow* v$
  ─ Another 4110 theorem
Values

• Values are already done:
  – Evaluation rule: \( v \implies v \)

• Constants are values
  – 42 is a value, so \( 42 \implies 42 \)
  – true is a value, so \( true \implies true \)
Operator evaluation

e1 + e2 ==> v
  if e1 ==> i1
  and e2 ==> i2
  and v is the result of primitive operation i1 + i2

e.g.,
true && false ==> false
1 + 2 ==> 3
1 + (2+3) ==> 6
Variables

• What does a variable name evaluate to?
  \[ x \implies ??? \]

• Trick question: we don’t have enough information to answer it

• To be continued...
Upcoming events

• [Thursday] A3 due

This is not just semantics.

THIS IS 3110