The Substitution Model

Prof. Clarkson
Fall 2017

Today’s music: Substitute by The Who
Review

Previously in 3110: simple interpreter for expression language
- abstract syntax tree (AST)
- evaluation based on single steps
- parser and lexer (in lab)

Today:
- Formal syntax: BNF
- Formal dynamic semantics: small-step, substitution model
- Formal static semantics
FORMAL SYNTAX
Abstract syntax of expression lang.

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

\( e, x, i \): *meta-variables* that stand for pieces of syntax
- \( e \): expressions
- \( x \): program variables, aka identifiers
- \( i \): integer constants, aka literals

\( ::= \) 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
Backus and Naur

John Backus (1924-2007)
ACM Turing Award Winner 1977
“For profound, influential, and lasting contributions to the design of practical high-level programming systems”

Peter Naur (1928-2016)
ACM Turing Award Winner 2005
“For fundamental contributions to programming language design”
Abstract syntax of expr. lang.

e ::= x | i | e1+e2
    | let x = e1 in e2

Note resemblance of BNF to AST type:

type expr =
    | Var of string
    | Int of int
    | Add of expr * expr
    | Let of string * expr * expr
Extended with Booleans

\[ e ::= x \mid i \mid b \]
\[ \mid e_1 + e_2 \mid e_1 \&\& e_2 \]
\[ \mid \text{let } x = e_1 \text{ in } e_2 \]
\[ \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \]

\[ v ::= i \mid b \]
FORMAL DYNAMIC SEMANTICS
Dynamic semantics

Defined as a binary relation:
\[ e \rightarrow e' \]
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:
\[ v ::= i | b \]
Dynamic semantics

Reflexive transitive closure of $\rightarrow$ is written $\rightarrow^*$

e $\rightarrow^*$ e' read as e multisteps to e'

e.g.,

$(5+2)+0 \rightarrow^* (5+2)+0$
$(5+2)+0 \rightarrow^* 7+0$
$(5+2)+0 \rightarrow^* 7$

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

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

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

\[ v_1 + v_2 \rightarrow i \]
\[ \text{if } i \text{ is the result of primitive operation } v_1 + v_2 \]
Dynamic semantics of expr. lang.

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

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

recall: read \(e_2\{v_1/x\}\) as \(e_2\) with \(v_1\) substituted for \(x\)
(as we defined last lecture and implemented in \texttt{subst})

so we call this the \texttt{substitution model of evaluation}
Evaluation models

• 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`
  – Inefficient implementation: have to do too much substitution at run time
  – Not really what OCaml does
  – Good mental model for evaluation

• Big-step environment model:
  – Keep a data structure that binds variables to values
  – At the heart of what OCaml really does
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:

\[ \text{v} \rightarrow \text{v} \]  \hspace{1cm} \text{(values are already done evaluating)}

\[ \text{x} \rightarrow \text{x} \]  \hspace{1cm} \text{(should have been substituted away)}

But they do multistep (because they can take 0 steps):

\[ \text{v} \rightarrow^* \text{v} \]

\[ \text{x} \rightarrow^* \text{x} \]
Scaling up to OCaml

Read notes on website: full dynamic semantics for core OCaml:

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

**Missing:** other built-in types, records, lists, options, declarations, patterns in function arguments and let bindings, if expressions, recursion
FORMAL STATIC SEMANTICS
Static semantics

We can have nonsensical expressions:

5 + false

if 5 then true else 0

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

Syntax:

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

Type checking:

if \( e_1 \) has type \textbf{bool} and \( e_2 \) has type \( t \) and \( e_3 \) has type \( t \)
then \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) has type \( t \)
Static semantics

Defined as a ternary relation:

\[ T \vdash e : t \]

- 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
Static semantics

e.g.,

\[
\begin{align*}
x : \text{int} & \vdash x + 2 : \text{int} \\
x : \text{int}, y : \text{int} & \vdash x < y : \text{bool} \\
& \vdash 5 + 2 : \text{int}
\end{align*}
\]
Static semantics of expr. lang.

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

\[ T \vdash b : \text{bool} \]
Static semantics of expr. lang.

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

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

\[ T \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t \]

\[
\begin{align*}
\text{if} & \quad T \vdash e_1 : \text{bool} \\
\text{and} & \quad T \vdash e_2 : t \\
\text{and} & \quad T \vdash e_3 : t 
\end{align*}
\]

\[ T \vdash \text{let } x : t_1 = e_1 \text{ in } e_2 : t_2 \]

\[
\begin{align*}
\text{if} & \quad T \vdash e_1 : t_1 \\
\text{and} & \quad T, x : t_1 \vdash e_2 : t_2 
\end{align*}
\]

\[ T, x : t \vdash x : t \]

To avoid need for type inference, require type annotation here.
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 \( e : t \), then either \( e \) is a value or \( e \) can take a step.
**Preservation**: if \( e : t \), and if \( e \) takes a step to \( e' \), then \( e' : t \).

Type safety = progress + preservation

Proving type safety is a fun part of CS 4110
Q: Why bother doing proofs about programming languages? They are almost always boring if the definitions are right.

A: The definitions are almost always wrong.

—Anonymous
Interpreter for expr. lang.

See `interp3.ml` in code for this lecture

1. Type-checks expression, then
2. Evaluates expression
Upcoming events

• [Wed] A3 due

This is not a substitute.

THIS IS 3110