CS 3110

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 | i | e1+e2
    | let x = e1 in e2
```

`e`, **`x`**, **`i`**: *meta-variables* that stand for pieces of syntax

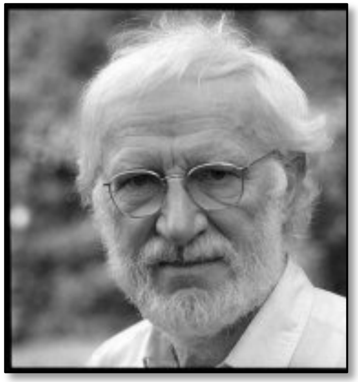
- **`e`**: expressions
- **`x`**: program variables, aka identifiers
- **`i`**: integer constants, aka literals

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

v ::= i | b
```

FORMAL DYNAMIC SEMANTICS

Dynamic semantics

Defined as a binary relation:

`e --> e'`

Read as `e` takes a single step to `e'`

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

Expressions continue to step until they reach a *value*

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

Values are a syntactic subset of expressions:

`v ::= i | b`

Dynamic semantics

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

`e -->* e'` read as `e` multisteps to `e'`

e.g.,

```
(5+2)+0 -->* (5+2)+0
(5+2)+0 -->* 7+0
(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.

```
e1 + e2 --> e1' + e2
```

if `e1 --> e1'`

```
v1 + e2 --> v1 + e2'
```

if `e2 --> e2'`

```
v1 + v2 --> i
```

if **`i`** *is the result of primitive operation* **`v1+v2`**

Dynamic semantics of expr. lang.

```
let x = e1 in e2 --> let x = e1' in e2
```
if `e1 --> e1'`

```
let x = v1 in e2 --> e2{v1/x}
```

recall: read **`e2{v1/x}`** as **`e2`** with **`v1`** substituted for **`x`**
(as we defined last lecture and implemented in **`subst`**)

so we call this the 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:

```
v -/->
```
(values are already done evaluating)

```
x -/->
```
(should have been substituted away)

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

```
v -->* v
x -->* x
```

Scaling up to OCaml

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

```
e ::= x | e1 e2 | fun x -> e
    | i | e1 + e2
    | (e1, e2) | fst e1 | snd e2
    | Left e | Right e
    | match e with Left x -> e1 | Right y -> e2
    | let x = e1 in e2
```

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:

```
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 as a ternary relation:

```
T |- e : t
```

- Read as in typing context `T`, expression `e` has type `t`
- Turnstile `|-` 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.,

```
x:int |- x+2 : int
x:int,y:int |- x<y : bool
|- 5+2 : int
```

Static semantics of expr. lang.

```
T |- i : int

T |- b : bool
```

Static semantics of expr. lang.

```
T |- e1 + e2 : int
```
if `T |- e1 : int`
and `T |- e2 : int`

```
T |- e1 && e2 : bool
```
if `T |- e1 : bool`
and `T |- e2 : bool`

Static semantics of expr. lang.

```
T |- if e1 then e2 else e3 : t
   if  T |- e1 : bool
   and T |- e2 : t
   and T |- e3 : t
```

To avoid need for type inference, require type annotation here

```
T |- let x:t1 = e1 in e2 : t2
   if  T |- e1 : t1
   and T, x:t1 |- e2 : t2

T, x:t |- x : t
```

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