CS 3110

Formal Semantics

Prof. Clarkson
Fall 2016

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

Today:

- Formal syntax: BNF
- Formal dynamic semantics:
  - small-step, substitution model
  - large-step, environment model
- Formal static semantics

FORMAL SYNTAX

Notation

- The 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...

Abstract syntax of expression lang.

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

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

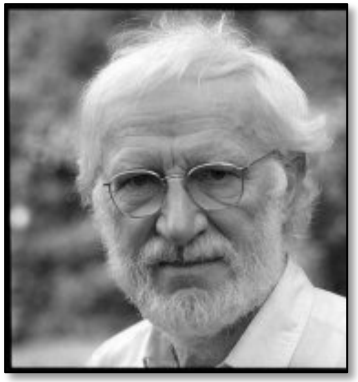
- **`e`**: expressions
- **`x`**: program variables
- **`i`**: integers

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

Note how closely the BNF resembles the OCaml variant we used to represent it!

FORMAL DYNAMIC SEMANTICS

Language we'll use for now

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

v ::= i | b
```

Dynamic semantics

Defined by a *judgement*:

```
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 --> n
```

if **`n`** *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

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 -/->
x -/->
```

- Values don't step because they're done computing
- Variables don't step we should never reach a variable; it should have already been substituted away

But they do multistep (because in 0 steps they are themselves):

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

Scaling up to OCaml

Read notes on website: full dynamic semantics
for core sublanguage of 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, unimportant: other built-in types, records, lists, options, declarations, patterns in function arguments and let bindings, `if`

Missing, important: `let rec`

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 by a *judgement*:

```
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 ext. expr. lang.

```
T |- i : int

T |- b : bool

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

Static semantics of ext. 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 ext. expr. lang.

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

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

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

Interpreter for ext. expr. lang.

See **`interp3.ml`** in code for this lecture

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

Preview of next lecture

Today we saw:

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

Next time we'll see:

- **Big-step environment model:** keep a data structure around that binds variables to values
  - Also a good mental model
  - At the heart of what OCaml really does

Upcoming events

- [ASAP] Prelim1 grades out
- [Friday] MS0 due, no late submissions

This is not just semantics.

THIS IS 3110