CS 3110

Formal Semantics

Prof. Clarkson

Fall 2015

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 | i | e1 + e2
    | 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

FORMAL DYNAMIC SEMANTICS

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
```

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.

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

v1 + e2 --> v1 + e2'
  if e2 --> e2'

v1 + v2 --> n
  if n is the sum of v1 and 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}
```

read `e2{v1/x}` as `e2` with `v1` substituted for `x`
(as we 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 -/->
```

But they do multistep:

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

because multistep includes 0 steps
(i.e., it is the *reflexive* transitive closure of **-->**)

- 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 | 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

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

```
e ::= ... | b | e1 && e2
    | if e1 then e2 else e3
v ::= ... | b
```

Now we could get nonsensical expressions, e.g.,

```
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
- The typing context is a new idea, but obviously needed to give types of variables in scope

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
```

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 --> e' --> ... --> v`
- We'll abstract that fact to `e ==> 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` ⇓ `v`
- **Goal:** `e ==> v` if and only if `e -->* v`
  - Another 4110 theorem

Values

- Values are already done:
  - Evaluation rule: `v ==> v`

- Constants are values
  - **`42`** is a value, so **`42 ==> 42`**
  - **`true`** is a value, so **`true ==> 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 ==> ???
```

- 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