Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2015

Big Step OS for λ calculus

- Configuration is simply a lambda expression
  - there is no state
- Result is a different lambda expression

- Inductive definition: Base case

$$\frac{}{x \to x}$$

- Inductive definition: recursive cases

$$\frac{e \to e'}{\lambda x. e \to \lambda x. e'} \qquad \frac{??}{e_1 e_2 \to e_3}$$

Big Step OS for Imperative Programs

- The same techniques apply to programs with state
  - The big difference is that the configuration now includes state

- Example: IMP

  e:= n | x | e_1 + e_2 | e_1 == e_2 | True | False

  c:= x := e | c_1 ; c_2 | if e then c_1 else c_2 | while e do c | skip

- Now we need two types of judgments

  expressions result in values

  $$\langle e, \sigma \rangle \to n$$

  commands change the state

  $$\langle c, \sigma \rangle \to \sigma'$$

Big Step OS for Imperative Programs

- Rules for expressions are very similar to what we had before

$$\frac{}{\langle N, \sigma \rangle \to n} \qquad \frac{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n = n_1 + n_2}{\langle e_1 + e_2, \sigma \rangle \to n}$$

- We need a rule to read values from variables

$$\frac{}{\langle x, \sigma \rangle \to \sigma(x)}$$

Big Step OS for Imperative Programs

- Commands mutate the state

$$\frac{\langle e, \sigma\rangle \rightarrow e'}{\langle X := e, \sigma\rangle \rightarrow \sigma[X \rightarrow e']}$$

$$\frac{\langle c_1, \sigma\rangle \rightarrow \sigma'' \quad \langle c_2, \sigma''\rangle \rightarrow \sigma'}{\langle c_1; c_2, \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow false \quad \langle c_f, \sigma\rangle \rightarrow \sigma'}{\langle if\ e_1 then\ c_t\ else\ c_f, \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\langle e_1, \sigma\rangle \rightarrow true \quad \langle c_t, \sigma\rangle \rightarrow \sigma'}{\langle if\ e_1 then\ c_t\ else\ c_f, \sigma\rangle \rightarrow \sigma'}$$

- What about loops?

Big Step OS for Imperative Programs

- The definition for loops must be recursive

$$\frac{\langle e_1, \sigma\rangle \to false}{\langle while\ e_1\ then\ c\ \ ,\sigma\rangle\ \to \sigma}$$

$$\frac{\langle e_1, \sigma\rangle \to true \quad \langle c; while\ e_1\ then\ c, \sigma\rangle \to \sigma'}{\langle while\ e_1\ then\ c\ \ ,\sigma\rangle\ \to \sigma'}$$

$$\frac{\langle e_1, \sigma\rangle \to true \quad \langle c, \sigma\rangle \to \sigma'' \quad \langle while\ e_1\ then\ c, \sigma''\rangle \to \sigma'}{\langle while\ e_1\ then\ c\ \ ,\sigma\rangle\ \to \sigma'}$$

Small Step Semantics

- Many design decisions
  - How small is a step?
  - How do we select the next step?

- These decisions need to be defined formally

Redex

- A redex is an expression that can be reduced in one atomic step.
- The first step in defining a small step semantics is to define the redexes.

- Ex.
  - In IMP: $n_1 + n_2$ | x := n | skip; c | if true then c1 else c2 | if false then c1 else c2 | while b do c
  - In λ-calculus : (λ x. v) e2 , (λ x. e1) e2

Local reduction rules

- One for each redex
  - show how to advance one step of the execution

  - $\langle x, \sigma[x = n] \rangle \to \langle n, \sigma \rangle$
  - $\langle n_1 + n_2, \sigma \rangle \to \langle n, \sigma \rangle$ where $n = n_1 + n_2$
  - $\langle x := n, \ \sigma \rangle \to \langle skip, \sigma[x \to n] \rangle$
  - $\langle skip; \ c, \sigma \rangle \to \langle c, \sigma \rangle$
  - $\langle if\ true\ then\ c_1\ else\ c_2, \sigma \rangle \to \langle c_1, \sigma \rangle$
  - $\langle if\ false\ then\ c1\ else\ c2, \sigma \rangle \to \langle c2, \sigma \rangle$
  - $\langle while\ b\ do\ c, \sigma \rangle \to \langle if\ b\ then\ (c;\ while\ b\ do\ c)\ else\ skip, \sigma \rangle$

Global reduction rules

- A simple algorithm
  - start with a program
  - identify a redex
  - reduce according to local reduction rules
  - repeat until you can't reduce anymore
- We need rules to define the next redex

Contexts

- We use H to refer to a context.
- H[r] is a program fragment consisting of redex r in context H

- Global reduction rules can be defined from local reduction rules as flows

- if $<r, \sigma> \rightarrow <e, \sigma'>$ then $<H[r], \sigma> \rightarrow <H[e], \sigma'>$

  - How we define the set of contexts will determine the order in which local reductions are applied.

Example

Configuration	Context	Redex
<x := (x + 1) + 2, [x=2]>	x = (o + 1) + 2	x
<x := (2 + 1) + 2, [x=2]>	x = o + 2	2 + 1
<x := 3 + 2, [x=2]>	x = o;	3 + 2
<x := 5, [x=2]>	o	x:=5
<skip, [x=5]>		

`The context is a program with a hole`

Contexts

- Contexts are defined by a grammar
- H ::= o | n + H | H + e | x:= H
  | if H then c1 else c2 | H; c
- The grammar defines the evaluation order
  - Note in a + b, a is evaluated before b.
- We can define redexes and contexts to
  - define the order of evaluation
  - define short circuit behavior

Contexts

- How do we know if our contexts and redexes are well defined?

- Decomposition theorem:

  If c is not "skip", then there exist unique H and r such that c is H[r]

  - Exist guarantees progress
  - Unique guarantees determinism

ML Style References

- Adding references

$$\tau ::= \ldots \mid \tau\ ref$$

$$\mathrm{e} ::= \ldots \mid ref\ e \mid e_1 \coloneqq e_2 \mid e_1; e_2 \mid !\, e$$

- Example:

$$\big(\lambda f{:}\, int\ \rightarrow (int\ ref).\ !\,(f\ 5)\big)\ (\lambda x{:}\, int.\ ref\ x)$$

$$(\lambda x{:}\, int\ ref\ .\ x \coloneqq 7; !\, x)\ ref\ x$$

- Equational reasoning is gone!

Modeling the Heap

- Heap is a map from addresses to values
  - $h ::= \emptyset \mid h, a \rightarrow val{:}\,\tau$
- A Program is an expression + a heap
  - $p \coloneqq heap\ h\ in\ e$
  - Heap addresses act as bound variables in expression

Small Step Semantics with Heap

- New contexts (in addition to the ones before)
  - H := ref H | H:=e | addrs:= H | !H

- No new local reduction rules

- New global reduction rules
  - $heap\ h\ in\ H[ref\ v{:}\tau] \rightarrow heap\ h, (a \rightarrow v){:}\tau in\ H[a]$
  - $heap\ h\ in\ H[!\,a] \rightarrow heap\ h\ in\ H[v]$
    - As long as $a \rightarrow v \in h$
  - $heap\ h\ in\ H[a \coloneqq v] \rightarrow heap\ h[a \rightarrow v]{:}\tau in\ H[*]$

Additional typing rules for references

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (ref\ e{:}\,\tau){:}\,\tau\ ref}$$

$$\frac{\Gamma \vdash e : \tau\ ref}{\Gamma \vdash\ !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau\ ref \Gamma \vdash e_2{:}\,\tau}{\Gamma \vdash e_1 \coloneqq e_2{:}\,unit}$$

References and polymorphism

$$let\ x : \forall t.(t \to t) ref = \Lambda t. ref\ (\lambda x : t. x)$$
$$in\ \ x[bool] := \lambda x : bool. not\ x\ ;$$
$$(!\, x[int])\ 5$$

- This is a big problem
- Solution: Disallow side effects in let.

6.820 Fundamentals of Program Analysis
Fall 2015