Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

- Inductive definition: Base case
  \[ x \rightarrow x \]

- Inductive definition: recursive cases

\[
\frac{e \rightarrow e'}{
\lambda x. e \rightarrow \lambda x. e'}
\]

\[
\frac{e_1 e_2 \rightarrow e_3}{??}
\]
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
  
  \[
  e := n \mid x \mid e_1 + e_2 \\
  c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
  \]

• Now we need two types of judgments
  expressions result in valuescommands change the state

\[
\langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma'
\]
Big Step OS for Imperative Programs

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

\[
\frac{\langle N, \sigma \rangle \to n}{\langle e_1, \sigma \rangle \to n_1 \quad \langle e_2, \sigma \rangle \to n_2 \quad n = n_1 + n_2}{\langle e_1 + e_2, \sigma \rangle \to n} \\
\frac{\langle e, \sigma \rangle \to n}{\langle x, \sigma \rangle \to \sigma(x)}
\]

- We need a rule to assign values to variables
Big Step OS for Imperative Programs

• Commands mutate the state

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

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

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

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

• What about loops?
• The definition for loops must be recursive

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{true}}{\langle c; \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\]

\[
\frac{\langle c, \sigma \rangle \rightarrow \sigma''}{\langle \text{while } e_1 \text{ then } c, \sigma'' \rangle \rightarrow \sigma'}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{true}}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\]
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: \( n_1 + n_2 \mid x \cdot = n \mid \text{skip}; c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \)
  - In \( \lambda \)-calculus: \((\lambda x. v) \ e_2\ , \ (\lambda x. e_1) \ e_2\)
Local reduction rules

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

- \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\)
- \(<n1+n2, \sigma> \rightarrow <n, \sigma>\) where \(n = n1 + n2\)
- \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x\rightarrow n]>\)
- \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\)
- \(<\text{if true then c1 else c2, } \sigma> \rightarrow <c1, \sigma>\)
- \(<\text{if false then c1 else c2, } \sigma> \rightarrow <c2, \sigma>\)
- \(<\text{while b do c, } \sigma> \rightarrow <\text{if b then (c; while b do c) else skip, } \sigma>\)
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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x := (x + 1) + 2, [x=2])&gt;</td>
<td>(x = (o + 1) + 2)</td>
<td>(x)</td>
</tr>
<tr>
<td>(&lt;x := (2 + 1) + 2, [x=2])&gt;</td>
<td>(x = o + 2)</td>
<td>(2 + 1)</td>
</tr>
<tr>
<td>(&lt;x := 3 + 2, [x=2])&gt;</td>
<td>(x = o;)</td>
<td>(3 + 2)</td>
</tr>
<tr>
<td>(&lt;x := 5, [x=2])&gt;</td>
<td>(o)</td>
<td>(x:=5)</td>
</tr>
<tr>
<td>(&lt;\text{skip}, [x=5])&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole
Contexts

• Contexts are defined by a grammar

• $H ::= o \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c_1 \text{ else } c_2 \mid H; c$

• 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
Call By Value evaluation (CBV)

- **Redex**

\[\langle(\lambda x. e_1) V_2\rangle \rightarrow \langle e_1[\alpha(V_2)/x]\rangle\]

- **Contexts**

\[
H ::= \text{o} \mid e_1 \mid H \mid H V
\]

\[
V ::= \lambda x. e_1
\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda s \ ((\lambda t) s)\ s\ (\lambda x. \lambda y. x) \ (\lambda w. w) \ (\lambda z. z))</td>
<td>(\lambda s \ ((\lambda t) s)\ s\ (\lambda z. z))</td>
<td>(\lambda x. \lambda y. x) ((\lambda w. w))</td>
</tr>
</tbody>
</table>

\[
\]
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
  \[ \tau ::= \ldots | \tau \text{ ref} \]
  \[ e ::= \ldots | \text{ref} \ e : \tau | e_1 ::= e_2 | e_1 ; e_2 |! e \]

- Example:
  \[ (\lambda f : \text{int } \rightarrow (\text{int ref}). \! (f \ 5)) (\lambda x : \text{int. ref } x) \]
  \[ (\lambda x : \text{int ref} . x ::= 7 ; ! x) \text{ ref } x \]

- Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset \mid h, a \rightarrow \text{val: } \tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap } h \text{ in } e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- New contexts
  - \( H := \text{ref } H \mid H := e \mid a := H \mid !H \)

- No new local reduction rules

- New global reduction rules
  - \( \text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{ in } H[a] \)

  - \( \text{heap } h \text{ in } H[!a] \rightarrow \text{heap } h \text{ in } H[v] \)
    - As long as \( a \rightarrow v \in h \)

  - \( \text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[*] \)
Typing Rules

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad & \Gamma \vdash e : \tau \text{ref} \\
\Gamma \vdash (\text{ref } e : \tau) : \tau \text{ref} & \quad & \Gamma \vdash !e : \tau \\
\Gamma \vdash e_1 : \tau \text{ref} \Gamma \vdash e_2 : \tau & \quad & \Gamma \vdash e_1 := e_2 : \text{unit}
\end{align*}
\]
References and polymorphism

- \( \text{let } x : \forall t. (t \to t) \text{ref} = \Lambda t. \text{ref} \ (\lambda x : t. x) \)
- \( \text{in} \)
  - \( x[\text{bool}] : = \lambda x : \text{bool}. \text{not } x \)
  - \( (! x[\text{int}]) \ 5 \)

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

Inductive definition: Base case

$$x \rightarrow x$$

Inductive definition: recursive cases

$$e \rightarrow e'$$

$$\frac{\lambda x. e \rightarrow \lambda x. e'}{e_1 e_2 \rightarrow e_3}$$
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
  
  \[ e := n \mid x \mid e_1 + e_2 \]
  \[ c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

• Now we need two types of judgments
  expressions result in values commands change the state

\[ \langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma' \]
Big Step OS for Imperative Programs

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

\[
\frac{\langle N, \sigma \rangle \rightarrow n}{\langle e_1, \sigma \rangle \rightarrow n_1 \quad \langle e_2, \sigma \rangle \rightarrow n_2 \quad n = n_1 + n_2}
\]

\[
\langle e_1 + e_2, \sigma \rangle \rightarrow n
\]

- We need a rule to assign values to variables

\[
\frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)}
\]
Big Step OS for Imperative Programs

• Commands mutate the state

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

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

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

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

• What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\frac{\langle e_1, \sigma \rangle \rightarrow false}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow true \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow true \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle \text{while } e_1 \text{ then } c, \sigma'' \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\]
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: \( n_1 + n_2 \ | \ x := n \ | \ \text{skip} \; c \ | \ \text{if} \; \text{true} \; \text{then} \; c_1 \; \text{else} \; c_2 \ | \ \text{if} \; \text{false} \; \text{then} \; c_1 \; \text{else} \; c_2 \ | \ \text{while} \; b \; \text{do} \; c \)
  - In \( \lambda \)-calculus: \((\lambda x. v) \; e_2\) , \((\lambda x. e_1) \; e_2\)
Local reduction rules

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

- \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\)
- \(<n1+n2, \sigma> \rightarrow <n, \sigma>\) where n = n1 + n2
- \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x\rightarrow n]>\)
- \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\)
- \(<\text{if true then } c1 \text{ else } c2, \sigma> \rightarrow <c1, \sigma>\)
- \(<\text{if false then } c1 \text{ else } c2, \sigma> \rightarrow <c2, \sigma>\)
- \(<\text{while } b \text{ do } c, \sigma> \rightarrow <\text{if } b \text{ then } (c; \text{ while } b \text{ do } c) \text{ else skip}, \sigma>\)
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

The context is a program with a hole

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x := (x + 1) + 2, [x=2])&gt;</td>
<td>(x = (o + 1) + 2)</td>
<td>(x)</td>
</tr>
<tr>
<td>(&lt;x := (2 + 1) + 2, [x=2])&gt;</td>
<td>(x = o + 2)</td>
<td>(2 + 1)</td>
</tr>
<tr>
<td>(&lt;x := 3 + 2, [x=2])&gt;</td>
<td>(x = o;)</td>
<td>(3 + 2)</td>
</tr>
<tr>
<td>(&lt;x := 5, [x=2])&gt;</td>
<td>(o)</td>
<td>(x:=5)</td>
</tr>
<tr>
<td>(&lt;\text{skip}, [x=5])&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contexts

- Contexts are defined by a grammar

- $H ::= o \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c1 \text{ else } c2 \mid H; c$

- 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
Call By Value evaluation (CBV)

- Redex

\[\langle (\lambda x. e_1)V_2 \rangle \rightarrow \langle e_1[\alpha(V_2)/x] \rangle\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(\lambda s ((\lambda t) s) s) (\lambda x. \lambda y. x) (\lambda w w) (\lambda z z)</code></td>
<td><code>(\lambda s ((\lambda t) s) s) (\lambda w w) (\lambda z z)</code></td>
<td><code>(\lambda x. \lambda y. x) (\lambda w w)</code></td>
</tr>
</tbody>
</table>

Contexts

\[H ::= \text{o} \mid e_1 \mid H \mid H V\]

\[V ::= \lambda x. e_1\]
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

\[ \tau ::= \ldots \mid \tau \text{ref} \]

\[ e ::= \ldots \mid \text{ref } e : \tau \mid e_1 := e_2 \mid e_1 ; e_2 \mid ! e \]

• Example:

\[
(\lambda f : \text{int} \to (\text{int ref}). \! (f \ 5)) (\lambda x : \text{int}. \text{ref } x)
\]

\[
(\lambda x : \text{int ref} . \ x := 7 ; ! x) \text{ref } x
\]

• Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  \[ h ::= \emptyset \mid h, a \rightarrow \text{val}: \tau \]

- A Program is an expression + a heap
  \[ p ::= \text{heap } h \text{ in } e \]

- Heap addresses act as bound variables in expression
Small Step Semantics with Heap

• New contexts
  - \( H := \text{ref } H \mid H := e \mid a := H \mid !H \)

• No new local reduction rules

• New global reduction rules
  - \( \text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{ in } H[a] \)

  - \( \text{heap } h \text{ in } H[!a] \rightarrow \text{heap } h \text{ in } H[v] \)
    - As long as \( a \rightarrow v \in h \)
  - \( \text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[*] \)
Typing Rules

\[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash (\text{ref } e : \tau) : \tau \text{ref} \]
\[ \Gamma \vdash !e : \tau \]

\[ \Gamma \vdash e_1 : \tau \text{ref} \Gamma \vdash e_2 : \tau \]
\[ \Gamma \vdash e_1 := e_2 : \text{unit} \]
References and polymorphism

- \( \text{let } x : \forall t. (t \rightarrow t) \text{ref} = \Lambda t. \text{ref} (\lambda x : t. x) \)
- \( \text{in} \)
- \( x[\text{bool}] : = \lambda x : \text{bool}. \text{not } x \)
- \( (! x[\text{int}]) 5 \)

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

Inductive definition: Base case

\[
x \rightarrow x
\]

Inductive definition: recursive cases

\[
e \rightarrow e' \\
\lambda x. e \rightarrow \lambda x. e'
\]

\[
e_1 e_2 \rightarrow e_3
\]
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 | e₁ + e₂
  c := x := e | c₁ ; c₂ | if e then c₁ else c₂ | while e do c

• Now we need two types of judgments
  expressions result in values commands change the state

\[ \langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma' \]
Big Step OS for Imperative Programs

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

\[
\frac{\langle N, \sigma \rangle \rightarrow n}{\langle e_1, \sigma \rangle \rightarrow n_1 \quad \langle e_2, \sigma \rangle \rightarrow n_2 \quad n = n_1 + n_2} \frac{\langle e_1, \sigma \rangle + \langle e_2, \sigma \rangle \rightarrow n}{\langle e_1 + e_2, \sigma \rangle \rightarrow n}
\]

- We need a rule to assign values to variables

\[
\frac{\langle x, \sigma \rangle \rightarrow \sigma(x)}{}
\]
Big Step OS for Imperative Programs

- Commands mutate the state

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

\[
\frac{c_1, \sigma \rightarrow \sigma''}{\langle c_1 ; c_2, \sigma \rangle \rightarrow \sigma'}
\]

\[
\frac{e_1, \sigma \rightarrow false}{if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rightarrow \sigma'}
\]

\[
\frac{e_1, \sigma \rightarrow true}{if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rightarrow \sigma'}
\]

- What about loops?
Big Step OS for Imperative Programs

• The definition for loops must be recursive

\[
\begin{align*}
\langle e_1, \sigma \rangle \rightarrow \text{false} \quad &\Rightarrow \quad \langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma \\
\langle e_1, \sigma \rangle \rightarrow \text{true} \quad &\Rightarrow \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma' \\
\langle e_1, \sigma \rangle \rightarrow \text{true} \quad &\Rightarrow \quad \langle \text{while } e_1 \text{ then } c, \sigma' \rangle \rightarrow \sigma' \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma' 
\end{align*}
\]
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: $n_1 + n_2 \mid x^* = n \mid \text{skip}; c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$
  - In $\lambda$-calculus: $\langle \lambda \ x. \ v \rangle \ e_2 \ , \ (\lambda \ x. \ e_1) \ e_2$
Local reduction rules

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

- \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\>
- \(<n1+n2, \sigma> \rightarrow <n, \sigma>\> \text{ where } n = n1 + n2
- \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x\mapsto n]>\>
- \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\>
- \(<\text{if true then c1 else c2}, \sigma> \rightarrow <c1, \sigma>\>
- \(<\text{if false then c1 else c2}, \sigma> \rightarrow <c2, \sigma>\>
- \(<\text{while b do c}, \sigma> \rightarrow <\text{if b then (c; while b do c) else skip}, \sigma>\>
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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x := (x + 1) + 2, [x=2]&gt;</td>
<td>x = (o + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>&lt;x := (2 + 1) + 2, [x=2]&gt;</td>
<td>x = o + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>&lt;x := 3 + 2, [x=2]&gt;</td>
<td>x = o;</td>
<td>3 + 2</td>
</tr>
<tr>
<td>&lt;x := 5, [x=2]&gt;</td>
<td>o</td>
<td>x:=5</td>
</tr>
<tr>
<td>&lt;skip, [x=5]&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
Call By Value evaluation (CBV)

• Redex

\[ ((\lambda x.e_1)V_2) \rightarrow e_1[\alpha(V_2)/x] \]

Contexts

\[
\begin{align*}
H & ::= \ o \ | \ e_1 \ H \ | \ H \ V \\
V & ::= \ \lambda x \ e_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda s ((\lambda t \ s) s)( \lambda x. \ y. \ x) (\lambda w \ w) (\lambda z \ z) ))</td>
<td>((\lambda s ((\lambda t \ s) s)( \ o(\lambda z \ z) ))</td>
<td>((\lambda x. \ y. \ x) (\lambda w \ w))</td>
</tr>
</tbody>
</table>
• 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

\[ \tau ::= \ldots \mid \tau \text{ ref} \]
\[ e ::= \ldots \mid \text{ref } e : \tau \mid e_1 := e_2 \mid e_1; e_2 \mid ! e \]

• Example:

\[ (\lambda f : \text{int } \to (\text{int ref}) \cdot ! (f \text{ 5})) \ (\lambda x : \text{int. ref } x) \]

\[ (\lambda x : \text{int ref} \cdot x := 7 ; ! x) \text{ ref } x \]

• 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
  - \( p ::= heap\ h\ in\ e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- New contexts
  - \( H := \text{ref } H \mid H := e \mid a := H \mid !H \)

- No new local reduction rules

- New global reduction rules
  - \( \text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \to v) : \tau \text{in } H[a] \)
  
  - \( \text{heap } h \text{ in } H[! a] \rightarrow \text{heap } h \text{ in } H[v] \)
    
    • As long as \( a \to v \in h \)
    
  - \( \text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \to v] : \tau \text{in } H[*] \)
Typing Rules

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\text{ref } e : \tau) : \tau \text{ ref}} \quad \frac{\Gamma \vdash e : \tau \text{ ref}}{\Gamma \vdash !e : \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \tau \text{ ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \text{unit}}
\]
References and polymorphism

• \( \text{let } x : \forall t. (t \to t) \text{ref} = \Lambda t. \text{ref} (\lambda x : t. x) \)
• \( \text{in} \)
• \( x[\text{bool}] = \lambda x : \text{bool}. \text{not } x \)
• \( (! x[\text{int}]) 5 \)

• This is a big problem

• Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011

Big Step OS for $\lambda$ calculus

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

- Inductive definition: Base case
  \[
  x \rightarrow x
  \]

- Inductive definition: recursive cases
  \[
  e \rightarrow e' \\
  \lambda x. e \rightarrow \lambda x. e'
  \]

  \[
  ?? \\
  e_1 e_2 \rightarrow e_3
  \]
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

  \[
  e := n \mid x \mid e_1 + e_2 \\
  c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
  \]

- Now we need two types of judgments
  expressions result in values
  commands change the state

\[
\langle e, \sigma \rangle \rightarrow n \hspace{2cm} \langle c, \sigma \rangle \rightarrow \sigma'
\]
Big Step OS for Imperative Programs

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

\[
\begin{align*}
\langle N, \sigma \rangle & \rightarrow n \\
\langle e_1, \sigma \rangle & \rightarrow n_1 \quad \langle e_2, \sigma \rangle \rightarrow n_2 \quad n = n_1 + n_2 \\
\langle e_1 + e_2, \sigma \rangle & \rightarrow n \\
\langle x, \sigma \rangle & \rightarrow \sigma(x)
\end{align*}
\]

- We need a rule to assign values to variables
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\begin{align*}
\langle e, \sigma \rangle & \rightarrow e' \\
\langle X := e, \sigma \rangle & \rightarrow \sigma[X \rightarrow e']
\end{align*}
\]

\[
\begin{align*}
\langle c_1, \sigma \rangle & \rightarrow \sigma'' \\
\langle c_2, \sigma'' \rangle & \rightarrow \sigma' \\
\langle c_1; c_2, \sigma \rangle & \rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle & \rightarrow \text{false} \\
\langle c_f, \sigma \rangle & \rightarrow \sigma' \\
\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle & \rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle & \rightarrow \text{true} \\
\langle c_t, \sigma \rangle & \rightarrow \sigma' \\
\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle & \rightarrow \sigma'
\end{align*}
\]

- What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow false \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow true \\
\langle c; \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow true \\
\langle c, \sigma \rangle &\rightarrow \sigma'' \\
\langle \text{while } e_1 \text{ then } c, \sigma'' \rangle &\rightarrow \sigma' \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]
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: \( n_1 + n_2 \mid x = n \mid \text{skip}; c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \)
- In \( \lambda \)-calculus: \((\lambda \ x. \ v) \ e_2, (\lambda \ x. \ e_1) \ e_2\)
Local reduction rules

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

- \( <x, \sigma[x=n]> \rightarrow <n, \sigma> \)
- \( <n1+n2, \sigma> \rightarrow <n, \sigma> \) where \( n = n1 + n2 \)
- \( <x = n, \sigma> \rightarrow <\text{skip}, \sigma[x \mapsto n]> \)
- \( <\text{skip}; c, \sigma> \rightarrow <c, \sigma> \)
- \( <\text{if true then c1 else c2}, \sigma> \rightarrow <c1, \sigma> \)
- \( <\text{if false then c1 else c2}, \sigma> \rightarrow <c2, \sigma> \)
- \( <\text{while b do c}, \sigma> \rightarrow <\text{if b then (c; while b do c) else skip}, \sigma> \)
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

The context is a program with a hole

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x := (x + 1) + 2, [x=2]&gt;</td>
<td>x = (o + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>&lt;x := (2 + 1) + 2, [x=2]&gt;</td>
<td>x = o + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>&lt;x := 3 + 2, [x=2]&gt;</td>
<td>x = o;</td>
<td>3 + 2</td>
</tr>
<tr>
<td>&lt;x := 5, [x=2]&gt;</td>
<td>o</td>
<td>x:=5</td>
</tr>
<tr>
<td>&lt;skip, [x=5]&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contexts

- Contexts are defined by a grammar

\[ H ::= o \mid n + H \mid H + e \mid x := H \]
\[
| \text{if } H \text{ then } c1 \text{ else } c2 \mid H; c
\]
- 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
Call By Value evaluation (CBV)

- **Redex**

\[ ((\lambda x. e_1)V_2) \rightarrow e_1[\alpha(V_2)/x] \]

- **Contexts**

\[
H ::= \text{ o } | \text{ e}_1 \text{ H } | \text{ H V }\\
V ::= \lambda x \text{ e}_1
\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda s ((\lambda t \text{ s})\text{ s})(\lambda x. \lambda y. x)(\lambda w \text{ w})(\lambda z z z)))</td>
<td>((\lambda s ((\lambda t \text{ s})\text{ s})(\text{ o } \lambda z z z)))</td>
<td>((\lambda x. \lambda y. x)(\lambda w \text{ w}))</td>
</tr>
</tbody>
</table>
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

\[
\begin{align*}
\tau &::= \ldots \mid \tau \text{ ref} \\
\tau &::= \ldots \mid \text{ref } e: \tau \mid e_1 := e_2 \mid e_1; e_2 \mid ! e
\end{align*}
\]

• Example:

\[
(\lambda f: \text{int } \rightarrow (\text{int ref}). \,(f\,5))
(\lambda x: \text{int. ref } x)
\]

\[
(\lambda x: \text{int ref} \cdot \,x := 7; \! x) \, \text{ref } x
\]

• Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset | h, a \rightarrow \text{val: } \tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap } h \text{ in } e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- New contexts
  - \( H := \text{ref } H \mid H := e \mid a := H \mid !H \)
- No new local reduction rules
- New global reduction rules
  - \( \text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{in } H[a] \)
  - \( \text{heap } h \text{ in } H[!a] \rightarrow \text{heap } h \text{ in } H[v] \)
    - As long as \( a \rightarrow v \in h \)
  - \( \text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{in } H[*] \)
Typing Rules

\[
\begin{align*}
\Gamma &\vdash e : \tau \\
\Gamma &\vdash (\text{ref } e : \tau) : \tau \text{ref} \\
\Gamma &\vdash e : \tau \text{ref} \\
\Gamma &\vdash !e : \tau \\
\Gamma &\vdash e_1 : \tau \text{ref} \Gamma \vdash e_2 : \tau \\
\Gamma &\vdash e_1 := e_2 : \text{unit}
\end{align*}
\]
References and polymorphism

- \( \text{let } x : \forall t. (t \to t) \text{ref} = \Lambda t. \text{ref} \ (\lambda x : t. x) \)
- \( \text{in} \)
- \( x[\text{bool}]: = \lambda x : \text{bool}. \text{not } x \)
- \( (! x[\text{int}]) 5 \)

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

Inductive definition: Base case

\[
\frac{x \to x}{\lambda x. e \to \lambda x. e'}
\]

Inductive definition: recursive cases

\[
\frac{e \to e'}{\lambda x. e \to \lambda x. e'}
\]

\[
\frac{?\?}{e_1 e_2 \to e_3}
\]
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
  
  $e := n \mid x \mid e_1 + e_2$

  $c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c$

• Now we need two types of judgments
  expressions result in values commands change the state

\[ \langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma' \]
Big Step OS for Imperative Programs

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

$$\langle N, \sigma \rangle \rightarrow n$$

$$\langle e_1, \sigma \rangle \rightarrow n_1 \quad \langle e_2, \sigma \rangle \rightarrow n_2 \quad n = n_1 + n_2$$

$$\langle e_1 + e_2, \sigma \rangle \rightarrow n$$

- We need a rule to assign values to variables

$$\langle x, \sigma \rangle \rightarrow \sigma(x)$$
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\begin{align*}
&\langle e, \sigma \rangle \rightarrow e' \\
\frac{\langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e']}{\langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e']}
\end{align*}
\]

\[
\begin{align*}
&\langle c_1, \sigma \rangle \rightarrow \sigma'' \\
&\langle c_2, \sigma'' \rangle \rightarrow \sigma' \\
\frac{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma'}{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma'}
\end{align*}
\]

\[
\begin{align*}
&\langle e_1, \sigma \rangle \rightarrow false \\
&\langle c_f, \sigma \rangle \rightarrow \sigma' \\
\frac{\langle if \ e_1 \text{then} \; c_t \ \text{else} \; c_f, \sigma \rangle \rightarrow \sigma'}{\langle if \ e_1 \text{then} \; c_t \ \text{else} \; c_f, \sigma \rangle \rightarrow \sigma'}
\end{align*}
\]

\[
\begin{align*}
&\langle e_1, \sigma \rangle \rightarrow true \\
&\langle c_t, \sigma \rangle \rightarrow \sigma' \\
\frac{\langle if \ e_1 \text{then} \; c_t \ \text{else} \; c_f, \sigma \rangle \rightarrow \sigma'}{\langle if \ e_1 \text{then} \; c_t \ \text{else} \; c_f, \sigma \rangle \rightarrow \sigma'}
\end{align*}
\]

- What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow false \\
\langle while e_1 then c, \sigma \rangle &\rightarrow \sigma \\
\langle e_1, \sigma \rangle &\rightarrow true \\
\langle c; while e_1 then c, \sigma \rangle &\rightarrow \sigma' \\
\langle while e_1 then c, \sigma \rangle &\rightarrow \sigma' \\
\langle e_1, \sigma \rangle &\rightarrow true \\
\langle c, \sigma \rangle &\rightarrow \sigma'' \\
\langle while e_1 then c, \sigma'' \rangle &\rightarrow \sigma' \\
\langle while e_1 then c, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]
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: $n_1 + n_2 | x := n | \text{skip}; \ c | \text{if true then } c_1 \ \text{else} \ c_2 | \text{if false then } c_1 \ \text{else} \ c_2 | \text{while } b \ \text{do} \ c$
  - In $\lambda$-calculus: $(\lambda x. v) \ e_2 , (\lambda x. e_1) \ e_2$
Local reduction rules

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

- \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\)
- \(<n1+n2, \sigma> \rightarrow <n, \sigma>\) where \(n = n1 + n2\)
- \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x\rightarrow n]>\)
- \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\)
- \(<\text{if true then c1 else c2}, \sigma> \rightarrow <c1, \sigma>\)
- \(<\text{if false then c1 else c2}, \sigma> \rightarrow <c2, \sigma>\)
- \(<\text{while b do c}, \sigma> \rightarrow <\text{if b then (c; while b do c) else skip}, \sigma>\)
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 follows:

• 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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x := (x + 1) + 2, [x=2]&gt;</td>
<td>x = (o + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>&lt;x := (2 + 1) + 2, [x=2]&gt;</td>
<td>x = o + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>&lt;x := 3 + 2, [x=2]&gt;</td>
<td>x = o;</td>
<td>3 + 2</td>
</tr>
<tr>
<td>&lt;x := 5, [x=2]&gt;</td>
<td>o</td>
<td>x:=5</td>
</tr>
<tr>
<td>&lt;skip, [x=5]&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole
Contexts

- Contexts are defined by a grammar

- $H ::= o \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c1 \text{ else } c2 \mid H; c$

- 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
Call By Value evaluation (CBV)

- Redex

\[ ((\lambda x.e_1)V_2) \rightarrow e_1[\alpha(V_2)/x] \]

- Contexts

\[
\begin{align*}
H & := o \mid e_1 H \mid H V \\
V & := \lambda x e_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda s ((\lambda t) s) s) (\lambda x. \lambda y. x) (\lambda w) (\lambda z z)</td>
<td>(\lambda s ((\lambda t) s) s) (o (\lambda z z))</td>
<td>(\lambda x. \lambda y. x) (\lambda w)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
  \[
  \tau ::= \ldots | \tau \text{ ref} \]
  \[
  e ::= \ldots | \text{ref } e : \tau | e_1 := e_2 | e_1 ; e_2 |! e
  \]

- Example:
  \[
  (\lambda f : \text{int} \rightarrow (\text{int} \text{ ref}). \ f(5)) (\lambda x : \text{int}. \text{ ref } x)
  \]
  \[
  (\lambda x : \text{int ref} . \ x := 7 ;! x) \text{ ref } x
  \]

- 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
  - \( p ::= heap \ h \ in \ e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

• New contexts
  - \( H ::= \text{ref } H | H := e | a := H | !H \)

• No new local reduction rules

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

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\text{ref } e : \tau) : \tau \text{ ref}} \quad \frac{\Gamma \vdash e : \tau \text{ ref}}{\Gamma \vdash !e : \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \tau \text{ ref} \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \text{unit}}
\]
References and polymorphism

- \(\text{let } x: \forall t. (t \to t) \text{ref} = \Lambda t. \text{ref} (\lambda x: t.x)\)
- \(\text{in}\)
  - \(x[\text{bool}]: = \lambda x: \text{bool}. \text{not } x\)
  - \((!x[\text{int}])\) 5

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

Inductive definition: Base case

$$x \rightarrow x$$

Inductive definition: recursive cases

$$e \rightarrow e'$$

$$\frac{\lambda x. e \rightarrow \lambda x. e'}{\frac{e \rightarrow e'}{e_1 e_2 \rightarrow e_3}}$$
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
  
  \[
  \begin{align*}
  e &:= n \mid x \mid e_1 + e_2 \\
  c &:= x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
  \end{align*}
  \]

- Now we need two types of judgments
  expressions result in values
  commands change the state

\[
\langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma'
\]
Big Step OS for Imperative Programs

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

\[
\begin{align*}
\langle N, \sigma \rangle &\rightarrow n \\
\langle e_1, \sigma \rangle &\rightarrow n_1 \quad \langle e_2, \sigma \rangle &\rightarrow n_2 \quad n = n_1 + n_2 \\
\langle e_1 + e_2, \sigma \rangle &\rightarrow \eta
\end{align*}
\]

- We need a rule to assign values to variables

\[
\langle x, \sigma \rangle \rightarrow \sigma(x)
\]
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\begin{align*}
  \langle e, \sigma \rangle & \rightarrow e' \\
  \langle X := e, \sigma \rangle & \rightarrow \sigma[X \rightarrow e'] \\
  \langle c_1, \sigma \rangle & \rightarrow \sigma'' \\
  \langle c_2, \sigma'' \rangle & \rightarrow \sigma' \\
  \langle c_1; c_2, \sigma \rangle & \rightarrow \sigma' \\
  \langle e_1, \sigma \rangle & \rightarrow \text{false} \\
  \langle c_f, \sigma \rangle & \rightarrow \sigma' \\
  \langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle & \rightarrow \sigma' \\
  \langle e_1, \sigma \rangle & \rightarrow \text{true} \\
  \langle c_t, \sigma \rangle & \rightarrow \sigma' \\
  \langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle & \rightarrow \sigma'
\end{align*}
\]

- What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } e_1 \text{ then } c \ , \sigma \rangle \rightarrow \sigma}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{true} \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c \ , \sigma \rangle \rightarrow \sigma'}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'' 
\langle \text{while } e_1 \text{ then } c, \sigma'' \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c \ , \sigma \rangle \rightarrow \sigma'}
\]
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: $n_1 + n_2 \mid x^{\neq} = n \mid \text{skip; c} \mid \text{if true then c1 else c2} \mid \text{if false then c1 else c2} \mid \text{while b do c}$
  - In $\lambda$-calculus: $(\lambda x. v) e_2 \, , \, (\lambda x. e_1) e_2$
Local reduction rules

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

- \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\)
- \(<n1+n2, \sigma> \rightarrow <n, \sigma>\) where \(n = n1 + n2\)
- \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x \rightarrow n]>\)
- \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\)
- \(<\text{if true then c1 else c2, } \sigma> \rightarrow <\text{c1, } \sigma>\)
- \(<\text{if false then c1 else c2, } \sigma> \rightarrow <\text{c2, } \sigma>\)
- \(<\text{while } b \text{ do } c, \sigma> \rightarrow <\text{if } b \text{ then (c; while } b \text{ do c) else skip, } \sigma>\)
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.
The context is a program with a hole

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x := (x + 1) + 2, [x=2]&gt;</td>
<td>x = (o + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>&lt;x := (2 + 1) + 2, [x=2]&gt;</td>
<td>x = o + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>&lt;x := 3 + 2, [x=2]&gt;</td>
<td>x = o;</td>
<td>3 + 2</td>
</tr>
<tr>
<td>&lt;x := 5, [x=2]&gt;</td>
<td>o</td>
<td>x:=5</td>
</tr>
<tr>
<td>&lt;skip, [x=5]&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Contexts

• Contexts are defined by a grammar

\[ H ::= o \mid n + H \mid H + e \mid x := H \]

\[ \mid \text{if } H \text{ then } c1 \text{ else } c2 \mid H; c \]

• 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
# Call By Value evaluation (CBV)

- **Redex**

\[ \langle (\lambda x. e_1)V_2 \rangle \rightarrow \langle e_1[\alpha(V_2)/x] \rangle \]

- **Contexts**

\[
\begin{align*}
H & ::= \ o | e_1 \ H | \ H \ V \\
V & ::= \ \lambda x \ e_1
\end{align*}
\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda s (\langle \lambda t \ t \rangle \ s)\ s)(\ \langle \lambda x. \ \lambda y. \ x \rangle \ (\lambda w \ w) \ (\lambda z \ z)))</td>
<td>((\lambda s (\langle \lambda t \ t \rangle \ s)\ s)(\ o \ (\lambda z \ z)))</td>
<td>((\lambda x. \ \lambda y. \ x) \ (\lambda w \ w))</td>
</tr>
</tbody>
</table>
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
  \[
  \tau ::= \ldots | \tau \text{ref}
  \]
  \[
  e ::= \ldots | \text{ref } e : \tau | e_1 ::= e_2 | e_1; e_2 |! e
  \]

- Example:
  \[
  (\lambda f : \text{int \to (int ref)}. \,(f\,5))\,(\lambda x : \text{int}. \text{ref}\,x)
  \]
  \[
  (\lambda x : \text{int ref}. \,x ::= 7;!x)\,\text{ref}\,x
  \]

- Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset | h, a \rightarrow val: \tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap } h \text{ in } e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- New contexts
  - \( H := \text{ref } H | H := e | a := H | !H \)

- No new local reduction rules

- New global reduction rules
  - \( \text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{ in } H[a] \)

  - \( \text{heap } h \text{ in } H[! a] \rightarrow \text{heap } h \text{ in } H[v] \)
    - As long as \( a \rightarrow v \in h \)
  - \( \text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[*] \)
Typing Rules

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad & \Gamma \vdash e : \tau \, ref \\
\frac{}{\Gamma \vdash (ref \ e : \tau) : \tau \, ref} & & \frac{}{\Gamma \vdash !e : \tau} \\
\Gamma \vdash e_1 : \tau \, ref \Gamma \vdash e_2 : \tau & & \frac{}{\Gamma \vdash e_1 := e_2 : unit}
\end{align*}
\]
References and polymorphism

• \( \text{let } x : \forall t. (t \rightarrow t) \text{ref} = \Lambda t. \text{ref} (\lambda x : t.x) \)
• \( \text{in} \)
• \( x[\text{bool}]: = \lambda x : \text{bool}. \text{not } x \)
• \( (! x[\text{int}]) 5 \)

• This is a big problem

• Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for \( \lambda \) calculus

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

- Inductive definition: Base case
  \[
  x \rightarrow x
  \]

- Inductive definition: recursive cases
  \[
  e \rightarrow e' \\
  \frac{\lambda x. e \rightarrow \lambda x. e'}{e_1 e_2 \rightarrow e_3}
  \]
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
  
  \( e := n \mid x \mid e_1 + e_2 \)
  
  \( c := \ x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \)

• Now we need two types of judgments
  
  expressions result in values \( \langle e, \sigma \rangle \rightarrow n \)
  
  commands change the state \( \langle c, \sigma \rangle \rightarrow \sigma' \)
Big Step OS for Imperative Programs

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

\[
\frac{\langle e_1, \sigma \rangle \rightarrow n_1 \quad \langle e_2, \sigma \rangle \rightarrow n_2 \quad n = n_1 + n_2}{\langle e_1 + e_2, \sigma \rangle \rightarrow n}
\]

- We need a rule to assign values to variables

\[
\frac{\langle x, \sigma \rangle \rightarrow \sigma(x)}{
}
\]
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\begin{align*}
\langle e, \sigma \rangle & \rightarrow e' \\
\frac{\langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e']}{\langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e']} \\
\langle c_1, \sigma \rangle & \rightarrow \sigma'' \\
\frac{\langle c_2, \sigma'' \rangle \rightarrow \sigma'}{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma'} \\
\langle e_1, \sigma \rangle & \rightarrow false \\
\frac{\langle c_f, \sigma \rangle \rightarrow \sigma'}{\langle if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rangle \rightarrow \sigma'} \\
\langle e_1, \sigma \rangle & \rightarrow true \\
\frac{\langle c_t, \sigma \rangle \rightarrow \sigma'}{\langle if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rangle \rightarrow \sigma'}
\end{align*}
\]

- What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{true} \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle \text{while } e_1 \text{ then } c, \sigma'' \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\]
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: \( n_1 + n_2 \mid x = n \mid \text{skip}; c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \)
  - In \( \lambda \)-calculus: \( (\lambda x. v) e_2 \), \( (\lambda x. e_1) e_2 \)
Local reduction rules

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

- \( <x, \sigma[x=n]> \rightarrow <n, \sigma> \)
- \( <n_1+n_2, \sigma> \rightarrow <n, \sigma> \) where \( n = n_1 + n_2 \)
- \( <x = n, \sigma> \rightarrow <\text{skip}, \sigma[x\rightarrow n]> \)
- \( <\text{skip}; c, \sigma> \rightarrow <c, \sigma> \)
- \( <\text{if true then c1 else c2}, \sigma> \rightarrow <c_1, \sigma> \)
- \( <\text{if false then c1 else c2}, \sigma> \rightarrow <c_2, \sigma> \)
- \( <\text{while b do c}, \sigma> \rightarrow <\text{if b then (c; while b do c) else skip}, \sigma> \)
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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x := (x + 1) + 2, [x=2])&gt;</td>
<td>(x = (o + 1) + 2)</td>
<td>(x)</td>
</tr>
<tr>
<td>(&lt;x := (2 + 1) + 2, [x=2])&gt;</td>
<td>(x = o + 2)</td>
<td>(2 + 1)</td>
</tr>
<tr>
<td>(&lt;x := 3 + 2, [x=2])&gt;</td>
<td>(x = o;)</td>
<td>(3 + 2)</td>
</tr>
<tr>
<td>(&lt;x := 5, [x=2])&gt;</td>
<td>(o)</td>
<td>(x:=5)</td>
</tr>
<tr>
<td>(&lt;skip, [x=5])&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole
Contexts

• Contexts are defined by a grammar

\[ H ::= o \mid n + H \mid H + e \mid x := H \]

\[ \mid \text{if } H \text{ then } c1 \text{ else } c2 \mid H; c \]

• 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
**Call By Value evaluation (CBV)**

- **Redex**

  \[\langle (\lambda x. e_1)V_2 \rangle \rightarrow \langle e_1[\alpha(V_2)/x] \rangle\]

- **Contexts**

  
  \[
  H ::= \ o \mid e_1 \ H \mid H\ V
  \]

  \[
  V ::= \ \lambda x \ e_1
  \]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda s ((\lambda t) s) s)((\lambda x. \lambda y. x)(\lambda w w)(\lambda z z )))</td>
<td>((\lambda s ((\lambda t) s)( o (\lambda z z )))</td>
<td>((\lambda x. \lambda y. x)(\lambda w w))</td>
</tr>
</tbody>
</table>
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

\[
\begin{align*}
\tau & ::= \ldots \mid \tau \text{ref} \\
e & ::= \ldots \mid \text{ref} \ e : \tau \mid e_1 := e_2 \mid e_1; e_2 \mid! e
\end{align*}
\]

• Example:

\[
(\lambda f : \text{int} \to (\text{int ref}). \ (! (f \ 5))) (\lambda x : \text{int. ref} \ x)
\]

\[
(\lambda x : \text{int ref} \ . \ x := 7; ! x) \text{ ref } x
\]

• Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - $h ::= \emptyset \mid h, a \to val : \tau$

- A Program is an expression + a heap
  - $p ::= \text{heap } h \text{ in } e$
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- **New contexts**
  - \( H := \text{ref } H \mid H := e \mid a := H \mid !H \)

- **No new local reduction rules**

- **New global reduction rules**
  - \( \text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{ in } H[a] \)

  - \( \text{heap } h \text{ in } H[! a] \rightarrow \text{heap } h \text{ in } H[v] \)
    - **As long as** \( a \rightarrow v \in h \)

  - \( \text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[\ast] \)
Typing Rules

\[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash (\text{ref } e : \tau) : \tau \text{ref} \]
\[ \Gamma \vdash e : \tau \text{ref} \]
\[ \Gamma \vdash !e : \tau \]

\[ \Gamma \vdash e_1 : \tau \text{ref} \quad \Gamma \vdash e_2 : \tau \]
\[ \Gamma \vdash e_1 := e_2 : \text{unit} \]
References and polymorphism

- \textit{let} \( x : \forall t. (t \rightarrow t) \text{ref} = \Lambda t. \text{ref} \ (\lambda x : t. x) \)
- \textit{in}
- \( x[\text{bool}] : = \lambda x : \text{bool}. \text{not} \ x \)
- \( (! x[\text{int}]) \ 5 \)

- This is a big problem

- Solution: Disallow side effects in \textit{let}.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

- Inductive definition: Base case
  $x \rightarrow x$

- Inductive definition: recursive cases

\[
\frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}
\]

??

\[
\frac{e_1 e_2 \rightarrow e_3}{
}\]
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
  
  \[
  \begin{align*}
  e &:= n \mid x \mid e_1 + e_2 \\
  c &:= x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
  \end{align*}
  \]

- Now we need two types of judgments
  expressions result in values commands change the state

\[
\langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma'
\]
Big Step OS for Imperative Programs

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

\[
\frac{\langle N, \sigma \rangle \rightarrow n}{\langle e_1, \sigma \rangle \rightarrow n_1 \quad \langle e_2, \sigma \rangle \rightarrow n_2 \quad n = n_1 + n_2}
\]

\[
\langle e_1 + e_2, \sigma \rangle \rightarrow \eta
\]

• We need a rule to assign values to variables

\[
\frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)}
\]
Big Step OS for Imperative Programs

• Commands mutate the state

\[
\begin{align*}
\langle e, \sigma \rangle & \to e' \\
\hline
\langle X := e, \sigma \rangle & \to \sigma[X \to e']
\end{align*}
\]

\[
\begin{align*}
\langle c_1, \sigma \rangle & \to \sigma'' \\
\langle c_2, \sigma'' \rangle & \to \sigma' \\
\hline
\langle c_1 ; c_2, \sigma \rangle & \to \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle & \to false \\
\langle c_f, \sigma \rangle & \to \sigma' \\
\hline
\langle if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rangle & \to \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle & \to true \\
\langle c_t, \sigma \rangle & \to \sigma' \\
\hline
\langle if \ e_1 \ then \ c_t \ else \ c_f, \sigma \rangle & \to \sigma'
\end{align*}
\]

• What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow \text{false} \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow \text{true} \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow \text{true} \quad \langle c, \sigma \rangle &\rightarrow \sigma'' \quad \langle \text{while } e_1 \text{ then } c, \sigma'' \rangle &\rightarrow \sigma' \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]
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: \( n_1 + n_2 \mid x^* = n \mid \text{skip}; c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \)
  - In \( \lambda \)-calculus: \((\lambda x. v) e_2, (\lambda x. e_1) e_2\)
Local reduction rules

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

- \( <x, \sigma[x=n]> \rightarrow <n, \sigma> \)
- \( <n_1+n_2, \sigma> \rightarrow <n, \sigma> \) where \( n = n_1 + n_2 \)
- \( <x = n, \sigma> \rightarrow <\text{skip}, \sigma[x \rightarrow n]> \)
- \( <\text{skip}; c, \sigma> \rightarrow <c, \sigma> \)
- \( <\text{if true then c1 else c2}, \sigma> \rightarrow <c_1, \sigma> \)
- \( <\text{if false then c1 else c2}, \sigma> \rightarrow <c_2, \sigma> \)
- \( <\text{while b do c}, \sigma> \rightarrow <\text{if b then } (c; \text{while b do c}) \text{ else skip}, \sigma> \)
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
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 follows.

- 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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x := (x + 1) + 2, [x=2]&gt;)</td>
<td>(x = (o + 1) + 2)</td>
<td>(x)</td>
</tr>
<tr>
<td>(&lt;x := (2 + 1) + 2, [x=2]&gt;)</td>
<td>(x = o + 2)</td>
<td>(2 + 1)</td>
</tr>
<tr>
<td>(&lt;x := 3 + 2, [x=2]&gt;)</td>
<td>(x = o;)</td>
<td>(3 + 2)</td>
</tr>
<tr>
<td>(&lt;x := 5, [x=2]&gt;)</td>
<td>(o)</td>
<td>(x := 5)</td>
</tr>
<tr>
<td>(&lt;\text{skip}, [x=5]&gt;)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole
Contexts

- Contexts are defined by a grammar

\[ H ::= o \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c_1 \text{ else } c_2 \mid H; c \]

- 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
Call By Value evaluation (CBV)

- Redex

\[ ((\lambda x. e_1)V_2) \rightarrow e_1[\alpha(V_2)/x] \]

- Contexts

\[
H ::= \ o \ | \ e_1 \ H \ | \ H \ V \\
V ::= \ \lambda x \ e_1
\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda s (\lambda t t) s)( (\lambda x. \lambda y. x) (\lambda w w) (\lambda z z) )</td>
<td>(\lambda s (\lambda t t) s)( o (\lambda z z) )</td>
<td>(\lambda x. \lambda y. x) (\lambda w w)</td>
</tr>
<tr>
<td>(\lambda s (\lambda t t) s)( (\lambda t t) s)( o (\lambda z z) )</td>
<td>(\lambda s (\lambda t t) s)( o (\lambda z z) )</td>
<td>(\lambda x. \lambda y. x) (\lambda w w)</td>
</tr>
</tbody>
</table>
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
  \[
  \tau ::= \ldots \mid \tau \text{ref} \\
  e ::= \ldots \mid \text{ref}\ e : \tau \mid e_1 := e_2 \mid e_1 ; e_2 \mid! e
  \]
- Example:
  \[
  (\lambda f : \text{int} \to (\text{int ref}). \ f\ 5) (\lambda x : \text{int}. \text{ref}\ x)
  \]
  \[
  (\lambda x : \text{int ref} . \ x := 7; !x) \text{ref}\ x
  \]
- Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  \[ h ::= \emptyset | h, a \rightarrow val : \tau \]

- A Program is an expression + a heap
  \[ p ::= heap \ h \ in \ e \]

  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- **New contexts**
  - \[ H ::= \text{ref } H \mid H := e \mid a := H \mid !H \]

- **No new local reduction rules**

- **New global reduction rules**
  - \[ \text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{ in } H[a] \]

  - \[ \text{heap } h \text{ in } H[! a] \rightarrow \text{heap } h \text{ in } H[v] \]
    - **As long as** \( a \rightarrow v \in h \)

  - \[ \text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[*] \]
Typing Rules

\[ \frac{ \Gamma \vdash e : \tau }{ \Gamma \vdash (\text{ref } e : \tau) : \tau \text{ ref} } \]

\[ \frac{ \Gamma \vdash e : \tau \text{ ref} }{ \Gamma \vdash !e : \tau } \]

\[ \frac{ \Gamma \vdash e_1 : \tau \text{ ref} \Gamma \vdash e_2 : \tau }{ \Gamma \vdash e_1 := e_2 : \text{unit} } \]
References and polymorphism

- \( \text{let } x: \forall t. (t \to t) \text{ref} = \Lambda t. \text{ref} \ (\lambda x: t. x) \)
- \( \text{in} \)
- \( x[\text{bool}]: = \lambda x: \text{bool}. \text{not } x \)
- \( (! x[\text{int}]) \ 5 \)

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for \( \lambda \) calculus

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

- Inductive definition: Base case
  \[
  x \rightarrow x
  \]

- Inductive definition: recursive cases
  \[
  e \rightarrow e' \quad \frac{\lambda x. e \rightarrow \lambda x. e'}{\lambda x. e \rightarrow \lambda x. e'}
  \]
  
  \[
  ??
  \]
  
  \[
  e_1 e_2 \rightarrow e_3
  \]
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

  \[ e := n \mid x \mid e_1 + e_2 \]

  \[ c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

- Now we need two types of judgments

  expressions result in values
  commands change the state

\[ \langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma' \]
Big Step OS for Imperative Programs

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

\[
\frac{(N, \sigma) \rightarrow n}{(e_1, \sigma) \rightarrow n_1 \quad (e_2, \sigma) \rightarrow n_2 \quad n = n_1 + n_2} \quad (e_1, \sigma) + (e_2, \sigma) \rightarrow n \quad (e_1 + e_2, \sigma) \rightarrow \eta
\]

• We need a rule to assign values to variables

\[
\frac{}{(x, \sigma) \rightarrow \sigma(x)}
\]
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\begin{align*}
\langle e, \sigma \rangle & \rightarrow e' \\
\langle X := e, \sigma \rangle & \rightarrow \sigma[X \rightarrow e'] \\
\langle c_1, \sigma \rangle & \rightarrow \sigma'' \\
\langle c_2, \sigma'' \rangle & \rightarrow \sigma' \\
\langle c_1; c_2, \sigma \rangle & \rightarrow \sigma' \\
\langle e_1, \sigma \rangle & \rightarrow \text{false} \\
\langle c_f, \sigma \rangle & \rightarrow \sigma' \\
\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle & \rightarrow \sigma' \\
\langle e_1, \sigma \rangle & \rightarrow \text{true} \\
\langle c_t, \sigma \rangle & \rightarrow \sigma' \\
\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle & \rightarrow \sigma'
\end{align*}
\]

- What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow false \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow true \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow true \quad \langle c, \sigma \rangle &\rightarrow \sigma'' \quad \langle \text{while } e_1 \text{ then } c, \sigma'' \rangle &\rightarrow \sigma' \\
\langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]
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: \( n_1 + n_2 \mid x \cdot = n \mid \text{skip; } c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \)
- In \( \lambda \)-calculus: \( (\lambda x. v) e_2, (\lambda x. e_1) e_2 \)
Local reduction rules

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

- \( <x, \sigma[x=n]> \rightarrow <n, \sigma> \)
- \( <n_1+n_2, \sigma> \rightarrow <n, \sigma> \) where \( n = n_1 + n_2 \)
- \( <x = n, \sigma> \rightarrow <\text{skip}, \sigma[x\rightarrow n]> \)
- \( <\text{skip}; c, \sigma> \rightarrow <c, \sigma> \)
- \( <\text{if true then } c_1 \text{ else } c_2, \sigma> \rightarrow <c_1, \sigma> \)
- \( <\text{if false then } c_1 \text{ else } c_2, \sigma> \rightarrow <c_2, \sigma> \)
- \( <\text{while } b \text{ do } c, \sigma> \rightarrow <\text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}, \sigma> \)
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 follows:

  - 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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x := (x + 1) + 2, [x=2])&gt;</td>
<td>(x = (o + 1) + 2)</td>
<td>(x)</td>
</tr>
<tr>
<td>(&lt;x := (2 + 1) + 2, [x=2])&gt;</td>
<td>(x = o + 2)</td>
<td>(2 + 1)</td>
</tr>
<tr>
<td>(&lt;x := 3 + 2, [x=2])&gt;</td>
<td>(x = o;)</td>
<td>(3 + 2)</td>
</tr>
<tr>
<td>(&lt;x := 5, [x=2])&gt;</td>
<td>(o)</td>
<td>(x:=5)</td>
</tr>
<tr>
<td>(&lt;\text{skip, [x=5]})&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole
Contexts

- Contexts are defined by a grammar

\[ H ::= o \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c1 \text{ else } c2 \mid H; c \]

- 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
Call By Value evaluation (CBV)

- **Redex**

\[ \langle (\lambda x. e_1)V_2 \rangle \rightarrow \langle e_1[\alpha(V_2)/x] \rangle \]

- **Contexts**

\[ H ::= \text{o} \mid e_1 H \mid H V \]

\[ V ::= \lambda x e_1 \]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda s ((\lambda t) s) s (\lambda x.\lambda y. x) (\lambda w) (\lambda z z) )</td>
<td>( \lambda s ((\lambda t) s) (\lambda z z) )</td>
<td>( \lambda x.\lambda y. x (\lambda w) )</td>
</tr>
</tbody>
</table>
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
  \[
  \tau ::= \ldots | \tau\ \text{ref}
  \]
  \[
  e ::= \ldots | \text{ref}\ e : \tau | e_1 := e_2 | e_1 ; e_2 | ! e
  \]

- Example:
  \[
  (\lambda f : \text{int} \to (\text{int\ ref}). \! (f\ 5))
  (\lambda x : \text{int}. \text{ref}\ x)
  \]
  \[
  (\lambda x : \text{int\ ref}. \ x := 7 ; ! x) \text{ref}\ x
  \]

- Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset \mid h, a \rightarrow \text{val}: \tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap } h \text{ in } e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- New contexts
  - $H := \text{ref } H \mid H := e \mid a := H \mid !H$

- No new local reduction rules

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

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{ref } e : \tau} \quad \frac{\Gamma \vdash e : \tau \text{ ref}}{\Gamma \vdash !e : \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \tau \text{ ref} \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \text{unit}}
\]
References and polymorphism

- \( \text{let } x : \forall t. (t \rightarrow t) \text{ref} = \Lambda t. \text{ref} (\lambda x : t. x) \)
- \( \text{in} \)
- \( x[\text{bool}]: = \lambda x : \text{bool}. \text{not } x \)
- \( (!x[\text{int}]) 5 \)

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

- Result is a different lambda expression

- Inductive definition: Base case
  \[
  x \rightarrow x
  \]

- Inductive definition: recursive cases
  \[
  e \rightarrow e' \\
  \frac{\lambda x. e \rightarrow \lambda x. e'}{??} \\
  e_1 e_2 \rightarrow e_3
  \]
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
  
  \[ e := n \mid x \mid e_1 + e_2 \]
  
  \[ c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

• Now we need two types of judgments
  expressions result in values
  commands change the state

\[ \langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma' \]
Big Step OS for Imperative Programs

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

\[
\begin{align*}
\langle e_1, \sigma \rangle \to n_1 & \quad \langle e_2, \sigma \rangle \to n_2 & \quad n = n_1 + n_2 \\
\langle e_1, \sigma \rangle + \langle e_2, \sigma \rangle \to n & \quad \langle e_1 + e_2, \sigma \rangle \to \eta
\end{align*}
\]

- We need a rule to assign values to variables

\[
\langle x, \sigma \rangle \to \sigma(x)
\]
Big Step OS for Imperative Programs

• Commands mutate the state

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

\[
\frac{\langle c_1, \sigma \rangle \rightarrow \sigma''}{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma'}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{false}}{\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle \rightarrow \sigma'}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{true}}{\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle \rightarrow \sigma'}
\]

• What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow false \\
\therefore \langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma 
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow true \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma' \\
\therefore \langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma' 
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow true \quad \langle c, \sigma \rangle &\rightarrow \sigma'' \quad \langle \text{while } e_1 \text{ then } c, \sigma'' \rangle &\rightarrow \sigma' \\
\therefore \langle \text{while } e_1 \text{ then } c, \sigma \rangle &\rightarrow \sigma' 
\end{align*}
\]
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: \( n_1 + n_2 \mid x_1^* = n \mid \text{skip}; c \mid \text{if true then c1 else c2} \mid \text{if false then c1 else c2} \mid \text{while b do c} \)
  - In \( \lambda \)-calculus: \((\lambda x. v) e_2, (\lambda x. e_1) e_2\)
Local reduction rules

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

- \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\)
- \(<n1+n2, \sigma> \rightarrow <n, \sigma>\)  where \(n = n1 + n2\)
- \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x \rightarrow n]>\)
- \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\)
- \(<\text{if true then c1 else c2}, \sigma> \rightarrow <c1, \sigma>\)
- \(<\text{if false then c1 else c2}, \sigma> \rightarrow <c2, \sigma>\)
- \(<\text{while b do c}, \sigma> \rightarrow <\text{if b then (c; while b do c) else skip}, \sigma>\)
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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;x := (x + 1) + 2, [x=2]&gt;</code></td>
<td><code>x = (o + 1) + 2</code></td>
<td><code>x</code></td>
</tr>
<tr>
<td><code>&lt;x := (2 + 1) + 2, [x=2]&gt;</code></td>
<td><code>x = o + 2</code></td>
<td><code>2 + 1</code></td>
</tr>
<tr>
<td><code>&lt;x := 3 + 2, [x=2]&gt;</code></td>
<td><code>x = o;</code></td>
<td><code>3 + 2</code></td>
</tr>
<tr>
<td><code>&lt;x := 5, [x=2]&gt;</code></td>
<td><code>o</code></td>
<td><code>x:=5</code></td>
</tr>
<tr>
<td><code>&lt;skip, [x=5]&gt;</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole
Contexts

- Contexts are defined by a grammar

- \( H ::= \ o \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c_1 \text{ else } c_2 \mid H; c \)

- 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
Call By Value evaluation (CBV)

- Redex

\[
((\lambda x. e_1)V_2) \rightarrow e_1[\alpha(V_2)/x]
\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda s\ ((\lambda t) \ s)\ s) ((\lambda x.\ \lambda y.\ x)\ \ (\lambda w\ w)\ \ (\lambda z\ z\ z)\ )\</td>
<td>(\lambda s\ ((\lambda t) \ s)\ \ (\lambda z\ z\ z)\ )\</td>
<td>((\lambda x.\ \lambda y.\ x)\ \ (\lambda w\ w))\</td>
</tr>
</tbody>
</table>
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
  \[ \tau ::= \ldots \mid \tau \text{ ref} \]
  \[ e ::= \ldots \mid \text{ref } e : \tau \mid e_1 := e_2 \mid e_1 ; e_2 \mid ! e \]

• Example:
  \[
  (\lambda f : \text{int} \rightarrow (\text{int ref}). \ ((f \ 5)) \ (\lambda x : \text{int}. \text{ ref } x))
  
  (\lambda x : \text{int ref} \ . \ x := 7 ; ! x) \text{ ref } x
  
  \]

• Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset | h, a \rightarrow \text{val}: \tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap } h \text{ in } e \)

- Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- New contexts
  - $H := \text{ref } H \mid H := e \mid a := H \mid !H$

- No new local reduction rules

- New global reduction rules
  - $\text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{ in } H[a]$

  - $\text{heap } h \text{ in } H[a] \rightarrow \text{heap } h \text{ in } H[v]$
    - As long as $a \rightarrow v \in h$

  - $\text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[*]$

October 13, 2011
Typing Rules

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \Rightarrow \quad \Gamma \vdash (\text{ref } e : \tau) : \tau \text{ ref} \\
\Gamma \vdash e : \tau \text{ ref} & \quad \Rightarrow \quad \Gamma \vdash !e : \tau \\
\Gamma \vdash e_1 : \tau \text{ ref} & \quad \Rightarrow \quad \Gamma \vdash e_2 : \tau \\
\Gamma \vdash e_1 \text{ := } e_2 : \text{unit} & \quad \Rightarrow \quad \Gamma \vdash e_2 : \text{unit}
\end{align*}
\]
References and polymorphism

- \( \text{let } x: \forall t. (t \to t) \text{ref} = \Lambda t. \text{ref} \ (\lambda x: t. x) \)
- \( \text{in} \)
- \( x[\text{bool}]: = \lambda x: \text{bool}. \text{not } x \)
- \( (! x[\text{int}]) \ 5 \)

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

- Inductive definition: Base case
  \[
  x \rightarrow x
  \]

- Inductive definition: recursive cases
  \[
  e \rightarrow e' \\
  \lambda x. e \rightarrow \lambda x. e' \\
  ?? \\
  e_1 e_2 \rightarrow e_3
  \]
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
  
  \[ e := n \mid x \mid e_1 + e_2 \]
  \[ c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

- Now we need two types of judgments
  expressions result in values
  commands change the state

\[ \langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma' \]
Big Step OS for Imperative Programs

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

\[
\frac{\langle e_1, \sigma \rangle \rightarrow n_1 \quad \langle e_2, \sigma \rangle \rightarrow n_2 \quad n = n_1 + n_2}{\langle e_1 + e_2, \sigma \rangle \rightarrow n}
\]

• We need a rule to assign values to variables

\[
\frac{}{\langle x, \sigma \rangle \rightarrow \sigma(x)}
\]
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\begin{align*}
\langle e, \sigma \rangle &\rightarrow e' \\
\langle X := e, \sigma \rangle &\rightarrow \sigma[X \rightarrow e'] \\
\langle c_1, \sigma \rangle &\rightarrow \sigma'' \\
\langle c_2, \sigma'' \rangle &\rightarrow \sigma' \\
\langle c_1; c_2, \sigma \rangle &\rightarrow \sigma' \\
\langle e_1, \sigma \rangle &\rightarrow \text{false} \\
\langle c_f, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle &\rightarrow \sigma' \\
\langle e_1, \sigma \rangle &\rightarrow \text{true} \\
\langle c_t, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

- What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\begin{align*}
\langle e_1, \sigma \rangle \rightarrow false \\
\hline
\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle \rightarrow true & \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma' \\
\hline
\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle \rightarrow true & \quad \langle c, \sigma \rangle \rightarrow \sigma'' & \quad \langle \text{while } e_1 \text{ then } c, \sigma'' \rangle \rightarrow \sigma' \\
\hline
\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'
\end{align*}
\]
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: $n_1 + n_2 \mid x^* = n \mid \text{skip; } c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$
  - In λ-calculus: $(\lambda x. v)\ e_2 \ , \ (\lambda x. e_1)\ e_2$
Local reduction rules

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

- \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\)
- \(<n1+n2, \sigma> \rightarrow <n, \sigma>\) where \(n = n1 + n2\)
- \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x\rightarrow n]>\)
- \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\)
- \(<\text{if true then c1 else c2}, \sigma> \rightarrow <c1, \sigma>\)
- \(<\text{if false then c1 else c2}, \sigma> \rightarrow <c2, \sigma>\)
- \(<\text{while } b \text{ do } c, \sigma> \rightarrow <\text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}, \sigma>\)
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 follows:

  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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x := (x + 1) + 2, [x=2]&gt;</td>
<td>x = (o + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>&lt;x := (2 + 1) + 2, [x=2]&gt;</td>
<td>x = o + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>&lt;x := 3 + 2, [x=2]&gt;</td>
<td>x = o;</td>
<td>3 + 2</td>
</tr>
<tr>
<td>&lt;x := 5, [x=2]&gt;</td>
<td>o</td>
<td>x:=5</td>
</tr>
<tr>
<td>&lt;skip, [x=5]&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
Call By Value evaluation (CBV)

- Redex

\[((\lambda . e_1) V_2) \rightarrow e_1[\alpha(V_2)/x]\]

- Contexts

\[H ::= \ o \ | \ e_1 \ H \ | \ H \ V\]
\[V ::= \ \lambda x \ e_1\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda s ((\lambda t) s)\ s)\ ((\lambda x. \lambda y. x)\ \lambda w\ w)\ (\lambda z\ z\ z))</td>
<td>((\lambda s ((\lambda t) s)\ s)\ (\ o\ (\lambda z\ z\ z))</td>
<td>((\lambda x. \lambda y. x)\ (\lambda w\ w))</td>
</tr>
</tbody>
</table>
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

\[ \tau ::= ... \mid \tau \text{ref} \]
\[ e ::= ... \mid \text{ref } e : \tau \mid e_1 := e_2 \mid e_1 ; e_2 \mid ! e \]

• Example:

\[ (\lambda f : \text{int } \rightarrow (\text{int ref}). \, ! (f \, 5)) \, (\lambda x : \text{int. ref } x) \]
\[ (\lambda x : \text{int ref} \, . \, x := 7 ; ! x) \, \text{ref } x \]

• Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset \mid h, a \rightarrow \text{val: } \tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap } h \text{ in } e \)

  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- New contexts
  - $H := \text{ref } H \mid H := e \mid a := H \mid \text{!H}$

- No new local reduction rules

- New global reduction rules
  - $\text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{ in } H[a]

  - $\text{heap } h \text{ in } H[\text{! } a] \rightarrow \text{heap } h \text{ in } H[v]$
    - As long as $a \rightarrow v \in h$

  - $\text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[*]$
Typing Rules

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\text{ref } e : \tau) : \tau \text{ref}} \quad \frac{\Gamma \vdash e : \tau \text{ref}}{\Gamma \vdash !e : \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \tau \text{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \text{unit}}
\]
References and polymorphism

- \( \text{let } x : \forall t. (t \rightarrow t) \text{ref} = \Lambda t. \text{ref} \ (\lambda x : t. x) \)
- \( \text{in} \)
- \( x[\text{bool}] : = \lambda x : \text{bool. not } x \)
- \( (! x[\text{int}]) \ 5 \)

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

- Inductive definition: Base case
  \[
  \frac{x \rightarrow x}{\lambda x. e \rightarrow \lambda x. e'}
  \]

- Inductive definition: recursive cases
  \[
  \frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}
  \]

  \[
  \frac{e_1 e_2 \rightarrow e_3}{? ?}
  \]
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
  c := x := e | c_1 ; c_2 | if e then c_1 else c_2 | while e do c

• Now we need two types of judgments
  expressions result in valuescommands change the state

\[ \langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma' \]
Big Step OS for Imperative Programs

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

\[
\begin{align*}
\langle N, \sigma \rangle &\rightarrow n \\
\langle e_1, \sigma \rangle &\rightarrow n_1 \\
\langle e_2, \sigma \rangle &\rightarrow n_2 \\
\langle e_1 + e_2, \sigma \rangle &\rightarrow n
\end{align*}
\]

- We need a rule to assign values to variables

\[
\langle x, \sigma \rangle \rightarrow \sigma(x)
\]
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\begin{align*}
\langle e, \sigma \rangle &\rightarrow e' \\
\frac{\langle X := e, \sigma \rangle}{\langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e']} \\
\langle c_1, \sigma \rangle &\rightarrow \sigma'' \\
\frac{\langle c_2, \sigma'' \rangle}{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma'}
\end{align*}
\]

- What about loops?

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow \text{false} \\
\frac{\langle f, \sigma \rangle \rightarrow \sigma' \quad \langle c_f, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle \rightarrow \sigma'} \\
\langle e_1, \sigma \rangle &\rightarrow \text{true} \\
\frac{\langle c_t, \sigma \rangle \rightarrow \sigma' \quad \langle c_f, \sigma \rangle \rightarrow \sigma'}{\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle \rightarrow \sigma'}
\end{align*}
\]
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow \text{false} \\
\langle \text{while } e_1 \text{ then } c , \sigma \rangle &\rightarrow \sigma
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow \text{true} \\
\langle c; \text{while } e_1 \text{ then } c , \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle \text{while } e_1 \text{ then } c , \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle e_1, \sigma \rangle &\rightarrow \text{true} \\
\langle c, \sigma \rangle &\rightarrow \sigma'' \\
\langle \text{while } e_1 \text{ then } c , \sigma'' \rangle &\rightarrow \sigma'
\end{align*}
\]

\[
\begin{align*}
\langle \text{while } e_1 \text{ then } c , \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]
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: \( n_1 + n_2 \mid x := n \mid \text{skip}; c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \)
  - In \( \lambda \)-calculus: \( (\lambda x. v) e_2, (\lambda x. e_1) e_2 \)
Local reduction rules

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

- $\langle x, \sigma[x=n]\rangle \rightarrow \langle n, \sigma\rangle$
- $\langle n_1+n_2, \sigma\rangle \rightarrow \langle n, \sigma\rangle$ where $n = n_1 + n_2$
- $\langle x = n, \sigma\rangle \rightarrow \langle \text{skip}, \sigma[x\rightarrow n]\rangle$
- $\langle \text{skip}; c, \sigma\rangle \rightarrow \langle c, \sigma\rangle$
- $\langle \text{if true then c1 else c2}, \sigma\rangle \rightarrow \langle c1, \sigma\rangle$
- $\langle \text{if false then c1 else c2}, \sigma\rangle \rightarrow \langle c2, \sigma\rangle$
- $\langle \text{while b do c}, \sigma\rangle \rightarrow \langle \text{if b then (c; while b do c) else skip}, \sigma\rangle$
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
• 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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x := (x + 1) + 2, [x=2]&gt;</td>
<td>x = (o + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>&lt;x := (2 + 1) + 2, [x=2]&gt;</td>
<td>x = o + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>&lt;x := 3 + 2, [x=2]&gt;</td>
<td>x = o;</td>
<td>3 + 2</td>
</tr>
<tr>
<td>&lt;x := 5, [x=2]&gt;</td>
<td>o</td>
<td>x:=5</td>
</tr>
<tr>
<td>&lt;skip, [x=5]&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
Call By Value evaluation (CBV)

- **Redex**

\[ (\lambda x. e_1) V_2 \rightarrow e_1[\alpha(V_2)/x] \]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\lambda s ((\lambda t) s) s)(\lambda x. \lambda y. x) (\lambda w w) (\lambda z z))</td>
<td>((\lambda s ((\lambda t) s) (\lambda z z)))</td>
<td>((\lambda x. \lambda y. x) (\lambda w w))</td>
</tr>
</tbody>
</table>
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

\[ \tau ::= \ldots \mid \tau \text{ref} \]

\[ e ::= \ldots \mid \text{ref} \ e : \tau \mid e_1 := e_2 \mid e_1 ; e_2 \mid ! e \]

• Example:

\[(\lambda f : \text{int} \rightarrow (\text{int ref}). \! (f \ 5)) \ (\lambda x : \text{int. ref} \ x)\]

\[(\lambda x : \text{int} \text{ref} . \ x := 7 ; ! x) \text{ref} \ x\]

• Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset \mid h, a \rightarrow \text{val:}\tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap}\ h\ \text{in}\ e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

• New contexts
  - \( H := \text{ref } H \mid H := e \mid a := H \mid \text{!}H \)

• No new local reduction rules

• New global reduction rules
  - \( \text{heap } h \text{ in } H[\text{ref } v : \tau] \rightarrow \text{heap } h, (a \rightarrow v) : \tau \text{ in } H[a] \)

  - \( \text{heap } h \text{ in } H[\text{!} a] \rightarrow \text{heap } h \text{ in } H[v] \)
    • As long as \( a \rightarrow v \in h \)
  - \( \text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[*] \)
Typing Rules

\[\Gamma \vdash e : \tau \quad \Gamma \vdash e : \tau \text{ref} \]

\[\Gamma \vdash \text{!}e : \tau \]

\[\Gamma \vdash e_1 : \tau \text{ref} \quad \Gamma \vdash e_2 : \tau \]

\[\Gamma \vdash e_1 := e_2 : \text{unit} \]
References and polymorphism

• $\text{let } x : \forall t. (t \rightarrow t) \text{ref} = \Lambda t. \text{ref} (\lambda x : t.x)$
• $\text{in}$
• $x[\text{bool}]: = \lambda x : \text{bool. not } x$
• $(! x[\text{int}]) 5$

• This is a big problem

• Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

- Inductive definition: Base case
  \[ x \rightarrow x \]

- Inductive definition: recursive cases
  \[
  e \rightarrow e' \\
  \lambda x. e \rightarrow \lambda x. e'
  \]
  \[
  e_1 e_2 \rightarrow e_3
  \]
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
  \[ e := n \mid x \mid e_1 + e_2 \]
  \[ c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

- Now we need two types of judgments
  - expressions result in values
  - commands change the state

\[ \langle e, \sigma \rangle \rightarrow n \] \[ \langle c, \sigma \rangle \rightarrow \sigma' \]
Big Step OS for Imperative Programs

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

\[
\begin{align*}
\langle N, \sigma \rangle &\rightarrow n \\
\langle e_1, \sigma \rangle &\rightarrow n_1 \\
\langle e_2, \sigma \rangle &\rightarrow n_2 \\
\langle e_1, \sigma \rangle + \langle e_2, \sigma \rangle &\rightarrow n \\
\langle e_1 + e_2, \sigma \rangle &\rightarrow \eta
\end{align*}
\]

- We need a rule to assign values to variables

\[
\langle x, \sigma \rangle \rightarrow \sigma(x)
\]
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\frac{\langle e, \sigma \rangle \rightarrow e' \quad \langle c_1, \sigma \rangle \rightarrow \sigma'' \quad \langle c_2, \sigma'' \rangle \rightarrow \sigma'}{\langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e'] \quad \langle c_1 ; c_2, \sigma \rangle \rightarrow \sigma'}
\]

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

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

- What about loops?
Big Step OS for Imperative Programs

• The definition for loops must be recursive

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{false}}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{true} \quad \langle c; \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\]

\[
\frac{\langle e_1, \sigma \rangle \rightarrow \text{true} \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle \text{while } e_1 \text{ then } c, \sigma'' \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\]
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: $n_1 + n_2 \mid x_{\lambda^1} \triangleq n \mid \text{skip}; \ c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$
- In $\lambda$-calculus: $(\lambda \ x. \ v) \ e_2 , (\lambda \ x. \ e_1) \ e_2$
Local reduction rules

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

  – \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\>
  – \(<n1+n2, \sigma> \rightarrow <n, \sigma>\) where \(n = n1 + n2\)
  – \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x\rightarrow n]>\>
  – \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\>
  – \(<\text{if true then c1 else c2}, \sigma> \rightarrow <c1, \sigma>\>
  – \(<\text{if false then c1 else c2}, \sigma> \rightarrow <c2, \sigma>\>
  – \(<\text{while b do c}, \sigma> \rightarrow <\text{if b then (c; while b do c) else skip}, \sigma>\>

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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x := (x + 1) + 2, [x=2]&gt;)</td>
<td>(x = (o + 1) + 2)</td>
<td>(x)</td>
</tr>
<tr>
<td>(&lt;x := (2 + 1) + 2, [x=2]&gt;)</td>
<td>(x = o + 2)</td>
<td>(2 + 1)</td>
</tr>
<tr>
<td>(&lt;x := 3 + 2, [x=2]&gt;)</td>
<td>(x = o;)</td>
<td>(3 + 2)</td>
</tr>
<tr>
<td>(&lt;x := 5, [x=2]&gt;)</td>
<td>(o)</td>
<td>(x:=5)</td>
</tr>
<tr>
<td>(&lt;\text{skip}, [x=5]&gt;)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole
Contexts

• Contexts are defined by a grammar

• \( H ::= o \mid n + H \mid H + e \mid x := H \)
  \( \mid \text{if } H \text{ then } c1 \text{ else } c2 \mid H; c \)

• 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
Call By Value evaluation (CBV)

- **Redex**

  \[ ((\lambda x.e_1)V_2) \rightarrow e_1[\alpha(V_2)/x] \]

- **Contexts**

  \[
  H ::= \ o \mid e_1 \ | \ H \ V \\
  V ::= \ \lambda x \ e_1
  \]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
</table>
| (\lambda s ((\lambda t \ s) \ s) (\lambda x. \ y. \ x) (\lambda w \ w) (\lambda z \ z) ) | (\lambda s ((\lambda t \ s) \ s)( \ o (\lambda z \ z) ) | (\lambda x. \ y. \ x) (\lambda w \ w) )

\[ ((\lambda x.e_1)V_2) \rightarrow e_1[\alpha(V_2)/x] \]
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

\[
\begin{align*}
\tau & ::= \ldots | \tau \text{ ref} \\
e & ::= \ldots | \text{ref } e : \tau | e_1 := e_2 | e_1 ; e_2 |! e
\end{align*}
\]

• Example:

\[
(\lambda f : \text{int } \to (\text{int ref}). \, !(f \, 5)) \, (\lambda x : \text{int}. \, \text{ref } x)
\]

\[
(\lambda x : \text{int ref}. \, x := 7 ;! x) \, \text{ref } x
\]

• Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset \mid h, a \rightarrow \text{val:} \tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap } h \text{ in } e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- **New contexts**
  - $H := \text{ref } H \mid H := e \mid a := H \mid !H$

- **No new local reduction rules**

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

  - $\text{heap } h \text{ in } H[! a] \rightarrow \text{heap } h \text{ in } H[v]$
    - As long as $a \rightarrow v \in h$
  
    - $\text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v] : \tau \text{ in } H[*]$
Typing Rules

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \Gamma \vdash (\text{ref } e : \tau) : \tau \text{ref} \\
\Gamma \vdash e : \tau \text{ref} & \quad \Gamma \vdash !e : \tau \\
\Gamma \vdash e_1 : \tau \text{ref} \quad \Gamma \vdash e_2 : \tau & \quad \Gamma \vdash e_1 := e_2 : \text{unit}
\end{align*}
\]
References and polymorphism

- \( \text{let } x : \forall t. (t \to t) \text{ref } = \Lambda t. \text{ref } (\lambda x : t. x) \)
- \( \text{in } \)
  - \( x[\text{bool}]{\textcolon} = \lambda x : \text{bool}. \text{not } x \)
  - \( (! x[\text{int}]) 5 \)

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for \(\lambda\) calculus

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

- Inductive definition: Base case
  \[
  \frac{x \rightarrow x}{\lambda x. e \rightarrow \lambda x. e'}
  \]

- Inductive definition: recursive cases

\[
\frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'} \quad ?? \quad \frac{e_1 e_2 \rightarrow e_3}{\lambda x. e \rightarrow \lambda x. e'}
\]
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
  \[ e := n \mid x \mid e_1 + e_2 \]
  \[ c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \]

- Now we need two types of judgments
  expressions result in values
  commands change the state

\[ \langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma' \]
Big Step OS for Imperative Programs

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

\[
\begin{align*}
\langle N, \sigma \rangle &\rightarrow n \\
\langle e_1, \sigma \rangle &\rightarrow n_1 \\
\langle e_2, \sigma \rangle &\rightarrow n_2 \\
\langle e_1 + e_2, \sigma \rangle &\rightarrow n \\
\langle e_1, \sigma \rangle + \langle e_2, \sigma \rangle &\rightarrow n \\
\langle e_1 + e_2, \sigma \rangle &\rightarrow n \\
\langle e_1, \sigma \rangle &\rightarrow n_1 + n_2
\end{align*}
\]

- We need a rule to assign values to variables

\[
\langle x, \sigma \rangle \rightarrow \sigma(x)
\]
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\frac{\langle e, \sigma \rangle \rightarrow e' \quad \langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e']}{\langle X := e, \sigma \rangle \rightarrow \sigma[X \rightarrow e']} \quad \frac{\langle c_1, \sigma \rangle \rightarrow \sigma'' \quad \langle c_2, \sigma'' \rangle \rightarrow \sigma'}{\langle c_1; c_2, \sigma \rangle \rightarrow \sigma'}
\]

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

- What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[
\begin{align*}
&\langle e_1, \sigma \rangle \rightarrow false \\
&\frac{}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma}
\end{align*}
\]

\[
\begin{align*}
&\langle e_1, \sigma \rangle \rightarrow true \\
&\frac{\langle c; \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\end{align*}
\]

\[
\begin{align*}
&\langle e_1, \sigma \rangle \rightarrow true \\
&\langle c, \sigma \rangle \rightarrow \sigma'' \\
&\frac{\langle \text{while } e_1 \text{ then } c, \sigma'' \rangle \rightarrow \sigma'}{\langle \text{while } e_1 \text{ then } c, \sigma \rangle \rightarrow \sigma'}
\end{align*}
\]
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: $n_1 + n_2 \mid x \triangleleft = n \mid \text{skip} \mid c \mid \text{if true then } c_1 \text{ else } c_2 \mid \text{if false then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c$
  - In $\lambda$-calculus: $(\lambda x. \nu) e_2, (\lambda x. e_1) e_2$
Local reduction rules

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

- $<x, \sigma[x=n]> \rightarrow <n, \sigma>$
- $<n_1+n_2, \sigma> \rightarrow <n, \sigma>$ where $n = n_1 + n_2$
- $<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x\rightarrow n]>$
- $<\text{skip}; c, \sigma> \rightarrow <c, \sigma>$
- $<\text{if true then } c_1 \text{ else } c_2, \sigma> \rightarrow <c_1, \sigma>$
- $<\text{if false then } c_1 \text{ else } c_2, \sigma> \rightarrow <c_2, \sigma>$
- $<\text{while } b \text{ do } c, \sigma> \rightarrow <\text{if } b \text{ then } (c; \text{ while } b \text{ do } c) \text{ else skip}, \sigma>$
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 follows

• 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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;x := (x + 1) + 2, [x=2]&gt;</td>
<td>x = (o + 1) + 2</td>
<td>x</td>
</tr>
<tr>
<td>&lt;x := (2 + 1) + 2, [x=2]&gt;</td>
<td>x = o + 2</td>
<td>2 + 1</td>
</tr>
<tr>
<td>&lt;x := 3 + 2, [x=2]&gt;</td>
<td>x = o;</td>
<td>3 + 2</td>
</tr>
<tr>
<td>&lt;x := 5, [x=2]&gt;</td>
<td>o</td>
<td>x:=5</td>
</tr>
<tr>
<td>&lt;skip, [x=5]&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The context is a program with a hole
Contexts

• Contexts are defined by a grammar

\[ H ::= o \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c_1 \text{ else } c_2 \mid H; c \]

• 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
Call By Value evaluation (CBV)

- **Redex**
  \[
  \langle (\lambda x.e_1)V_2 \rangle \to \langle e_1[\alpha(V_2)/x] \rangle
  \]

- **Contexts**
  \[
  H ::= \ o \ | \ e_1 \ H \ | \ H \ V
  \]
  \[
  V ::= \ \lambda x \ e_1
  \]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda s ((\lambda t) s) (\lambda x. \lambda y. x) (\lambda w) (\lambda z z))</td>
<td>(\lambda s ((\lambda t) s) (\ o \ (\lambda z z)))</td>
<td>((\lambda x. \lambda y. x) (\lambda w w))</td>
</tr>
</tbody>
</table>
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
  \[
  \tau ::= \ldots | \tau \text{ ref} \\
  e ::= \ldots | \text{ref } e : \tau | e_1 ::= e_2 | e_1; e_2 | e
  \]

- Example:
  \[
  (\lambda f : \text{int } \to (\text{int ref}) \cdot !(f \ 5)) \ (\lambda x : \text{int. ref } x) \\
  \\
  (\lambda x : \text{int ref } \cdot x ::= 7; !x) \ \text{ref } x
  \]

- 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
  - \( p ::= heap h \ in e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

• New contexts
  - \( H := \text{ref } H \mid H := e \mid a := H \mid \neg H \)

• No new local reduction rules

• New global reduction rules
  - \( \text{heap } h \text{ in } H[\text{ref } v: \tau] \rightarrow \text{heap } h, (a \rightarrow v): \tau \text{ in } H[a] \)
  - \( \text{heap } h \text{ in } H[\neg a] \rightarrow \text{heap } h \text{ in } H[v] \)
    • As long as \( a \rightarrow v \in h \)
  - \( \text{heap } h \text{ in } H[a := v] \rightarrow \text{heap } h[a \rightarrow v]: \tau \text{ in } H[*] \)
Typing Rules

\[
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\text{ref } e : \tau) : \tau \text{ ref}} \quad \frac{\Gamma \vdash e : \tau \text{ ref}}{\Gamma \vdash !e : \tau}
\]

\[
\frac{\Gamma \vdash e_1 : \tau \text{ ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \text{unit}}
\]
References and polymorphism

- \( \text{let } x : \forall t. (t \to t) \text{ref} = \Lambda t. \text{ref} (\lambda x : t.x) \)
- \( \text{in} \)
- \( x[\text{bool}]: = \lambda x : \text{bool}. \text{not} x \)
- \( (! x[\text{int}]) 5 \)

- This is a big problem

- Solution: Disallow side effects in let.
Types for Imperative Programs

Armando Solar-Lezama
Computer Science and Artificial Intelligence Laboratory
MIT

Derived from slides by George Necula

October 13, 2011
Big Step OS for $\lambda$ calculus

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

- Inductive definition: Base case
  \[
  \frac{\lambda x. e \rightarrow \lambda x. e'}{x \rightarrow x}
  \]

- Inductive definition: recursive cases
  \[
  \frac{e \rightarrow e'}{\lambda x. e \rightarrow \lambda x. e'}
  \]

  \[
  \frac{e_1 e_2 \rightarrow e_3}{??}
  \]
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
  \[
  e := n \mid x \mid e_1 + e_2 \\
  c := x := e \mid c_1 ; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c
  \]

• Now we need two types of judgments
  expressions result in values commands change the state

\[
\langle e, \sigma \rangle \rightarrow n \quad \langle c, \sigma \rangle \rightarrow \sigma'
\]
Big Step OS for Imperative Programs

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

\[
\begin{align*}
\langle N, \sigma \rangle &\rightarrow n \\
\langle e_1, \sigma \rangle &\rightarrow n_1 \\
\langle e_2, \sigma \rangle &\rightarrow n_2 \\
\langle e_1 + e_2, \sigma \rangle &\rightarrow n \\
\langle \eta, \sigma \rangle &\rightarrow n
\end{align*}
\]

- We need a rule to assign values to variables

\[
\langle x, \sigma \rangle \rightarrow \sigma(x)
\]
Big Step OS for Imperative Programs

- Commands mutate the state

\[
\begin{align*}
\langle e, \sigma \rangle &\rightarrow e' \\
\langle X := e, \sigma \rangle &\rightarrow \sigma[X \rightarrow e'] \\
\langle c_1, \sigma \rangle &\rightarrow \sigma'' \\
\langle c_2, \sigma'' \rangle &\rightarrow \sigma' \\
\langle e_1, \sigma \rangle &\rightarrow \text{false} \\
\langle c_f, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle &\rightarrow \sigma' \\
\langle e_1, \sigma \rangle &\rightarrow \text{true} \\
\langle c_t, \sigma \rangle &\rightarrow \sigma' \\
\langle \text{if } e_1 \text{ then } c_t \text{ else } c_f, \sigma \rangle &\rightarrow \sigma'
\end{align*}
\]

- What about loops?
Big Step OS for Imperative Programs

- The definition for loops must be recursive

\[ \langle e_1, \sigma \rangle \rightarrow false \quad \frac{\langle while \ e_1 \ then \ c \ , \sigma \rangle \rightarrow \sigma}{\langle while \ e_1 \ then \ c \ , \sigma \rangle \rightarrow \sigma} \]

\[ \langle e_1, \sigma \rangle \rightarrow true \quad \langle c; while \ e_1 \ then \ c, \sigma \rangle \rightarrow \sigma' \quad \frac{\langle while \ e_1 \ then \ c \ , \sigma \rangle \rightarrow \sigma'}{\langle while \ e_1 \ then \ c \ , \sigma \rangle \rightarrow \sigma'} \]

\[ \langle e_1, \sigma \rangle \rightarrow true \quad \langle c, \sigma \rangle \rightarrow \sigma'' \quad \langle while \ e_1 \ then \ c, \sigma'' \rangle \rightarrow \sigma' \quad \frac{\langle while \ e_1 \ then \ c \ , \sigma \rangle \rightarrow \sigma'}{\langle while \ e_1 \ then \ c \ , \sigma \rangle \rightarrow \sigma'} \]
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: $n_1 + n_2 \mid x \cdot = n \mid \text{skip;} \ c \mid \text{if true then } c_1 \ \text{else } c_2 \mid \text{if false then } c_1 \ \text{else } c_2 \mid \text{while } b \ \text{do } c$
  - In $\lambda$-calculus: $(\lambda \ x \ . \ v) \ e_2 \ , \ (\lambda \ x \ . \ e_1) \ e_2$
Local reduction rules

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

- \(<x, \sigma[x=n]> \rightarrow <n, \sigma>\)
- \(<n1+n2, \sigma> \rightarrow <n, \sigma>\) where \(n = n1 + n2\)
- \(<x = n, \sigma> \rightarrow <\text{skip}, \sigma[x \rightarrow n]>\)
- \(<\text{skip}; c, \sigma> \rightarrow <c, \sigma>\)
- \(<\text{if true then c1 else c2}, \sigma> \rightarrow <c1, \sigma>\)
- \(<\text{if false then c1 else c2}, \sigma> \rightarrow <c2, \sigma>\)
- \(<\text{while b do c}, \sigma> \rightarrow <\text{if b then (c; while b do c) else skip}, \sigma>\)
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 follows:

- 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

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(&lt;x := (x + 1) + 2, [x=2])&gt;</td>
<td>(x = (o + 1) + 2)</td>
<td>(x)</td>
</tr>
<tr>
<td>(&lt;x := (2 + 1) + 2, [x=2])&gt;</td>
<td>(x = o + 2)</td>
<td>(2 + 1)</td>
</tr>
<tr>
<td>(&lt;x := 3 + 2, [x=2])&gt;</td>
<td>(x = o;)</td>
<td>(3 + 2)</td>
</tr>
<tr>
<td>(&lt;x := 5, [x=2])&gt;</td>
<td>(o)</td>
<td>(x:=5)</td>
</tr>
<tr>
<td>(&lt;\text{skip, [x=5]})&gt;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
Call By Value evaluation (CBV)

- Redex

\[ \langle (\lambda x. e_1) V_2 \rangle \rightarrow \langle e_1[\alpha(V_2)/x] \rangle \]

- Contexts

\[
H ::= \ o \mid e_1 \ H \mid H V \\
V ::= \lambda x \ e_1
\]

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Context</th>
<th>Redex</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda s ((\lambda t) s) s)( (\lambda x. \lambda y. x) (\lambda w w) (\lambda z z z) ))</td>
<td>(\lambda s ((\lambda t) s)( o (\lambda z z z) ))</td>
<td>((\lambda x. \lambda y. x) (\lambda w w))</td>
</tr>
</tbody>
</table>
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
  \[ \tau ::= \ldots | \tau \text{ ref} \]
  \[ e ::= \ldots | \text{ ref}\ e : \tau | e_1 ::= e_2 | e_1 ; e_2 \mid ! e \]

- Example:
  \[ (\lambda f : \text{ int} \to (\text{ int ref}). ! (f\ 5)) (\lambda x : \text{ int}. \text{ ref}\ x) \]
  \[ (\lambda x : \text{ int ref}. x ::= 7; ! x) \text{ ref}\ x \]

- Equational reasoning is gone!
Modeling the Heap

- Heap is a map from addresses to values
  - \( h ::= \emptyset \mid h, a \rightarrow \text{val:}\tau \)

- A Program is an expression + a heap
  - \( p ::= \text{heap } h \text{ in } e \)
  - Heap addresses act as bound variables in expression
Small Step Semantics with Heap

- **New contexts**
  - \( H := \text{ref } H \mid H := e \mid a := H \mid !H \)

- **No new local reduction rules**

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

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \Gamma \vdash (\text{ref } e : \tau) : \tau \text{ ref} \\
\Gamma \vdash e : \tau \text{ ref} & \quad \Gamma \vdash !e : \tau \\
\Gamma \vdash e_1 : \tau \text{ ref} & \quad \Gamma \vdash e_2 : \tau \\
\Gamma \vdash e_1 := e_2 : \text{unit}
\end{align*}
\]
References and polymorphism

- \( \text{let } x : \forall t. (t \to t) \text{ref} = \Lambda t. \text{ref} \ (\lambda x : t. x) \)
- \( \text{in} \)
- \( x[\text{bool}] : = \lambda x : \text{bool}. \text{not } x \)
- \( (! x[\text{int}]) \ 5 \)

- This is a big problem

- Solution: Disallow side effects in let.