15-411: Dynamic Semantics

Jan Hoffmann

15-411: Dynamic Semantics

Jan Hoffmann

Sign up for code review!

Dynamic Semantics

- Static semantics: definition of valid programs
- Dynamic semantics: definition of how programs are executed
- So far: Dynamic semantics is given in English on lab handouts
 - This only works since you know how C programs should behave
 - Sometimes needed to consult the reference compiler
- A description in English will always be ambiguous
- Need precise ways of defining the meaning of programs

- Denotational Semantics: Abstract and elegant.
 - Each part of a program is associated with a denotation (math. object)
 - For example: a procedure is associated with a mathematical function
- Axiomatic Semantics: Strongly related to program logic.
 - Gives meaning to phrases using logical axioms
 - The meaning is identical to the set of properties that can be proved
- Operational Semantics: Describes how programs are executed
 - Related to interpreters and abstract machines
 - Most popular and flexible form of semantics

- Denotational Semantics: Abstract and elegant.
 Dana Scott
 - Each part of a program is associated with a denotation (math. object)
 - For example: a procedure is associated with a mathematical function
- Axiomatic Semantics: Strongly related to program logic.
 - Gives meaning to phrases using logical axioms
 - The meaning is identical to the set of properties that can be proved
- Operational Semantics: Describes how programs are executed
 - Related to interpreters and abstract machines
 - Most popular and flexible form of semantics

- Denotational Semantics: Abstract and elegant.
 Dana Scott
 - Each part of a program is associated with a denotation (math. object)
 - For example: a procedure is associated with a mathematical function

Tony Hoare

- Axiomatic Semantics: Strongly related to program logic.
 - Gives meaning to phrases using logical axioms
 - The meaning is identical to the set of properties that can be proved
- Operational Semantics: Describes how programs are executed
 - Related to interpreters and abstract machines
 - Most popular and flexible form of semantics

- Denotational Semantics: Abstract and elegant.
 - Each part of a program is associated with a denotation (math. object)
 - For example: a procedure is associated with a mathematical function
- Axiomatic Semantics: Strongly related to program logic.
 - Gives meaning to phrases using logical axioms
 - The meaning is identical to the set of properties that can be proved
- Operational Semantics: Describes how programs are executed
 - Related to interpreters and abstract machines
 - Most popular and flexible form of semantics

Tony Hoare



t. Dana Scott

Operational Semantics

Many different styles

- Natural semantics (or big-step semantics or evaluation dynamics)
- Structural operational semantics
- Substructural operational semantics
- Abstract machine (or small-step with continuation)

• We will use an abstract machine

- Very general: can describe non-termination, concurrency, …
- Low-level and elaborate

Operational Semantics

Many different styles

- Natural semantics (or big-step semantics or evaluation dynamics)
- Structural operational semantics
- Substructural operational semantics
- Abstract machine (or small-step with continuation)

We will use an abstract machine

- Very general: can describe non-termination, concurrency, …
- Low-level and elaborate

Frank Pfenning

Operational Semantics

Many different styles

- Natural semantics (or big-step semantics or evaluation dynamics)
- Structural operational semantics
- Substructural operational semantics
- Abstract machine (or small-step with continuation)

We will use an abstract machine

- Very general: can describe non-termination, concurrency, …
- Low-level and elaborate

How to pick the right dynamic semantics?

Frank Pfenning

Evaluating Expressions

Want to model a single evaluation step

$$e \to e'$$

Want to model a single evaluation step

$$e \to e'$$

For example: $((4+5) * 10 + 2) \rightarrow (9 * 10 + 2)$

Want to model a single evaluation step

$$e \rightarrow e'$$

For example: $((4+5) * 10 + 2) \rightarrow (9 * 10 + 2)$

How can we find the right place at which to make the step?

Want to model a single evaluation step

$$e \to e'$$

For example: $((4+5) * 10 + 2) \rightarrow (9 * 10 + 2)$

How can we find the right place at which to make the step?

Use a continuation K:

$e \triangleright K$

"Evaluate expression e and pass the result to K"

The continuation has a 'hole' for the result value of e.

Want to model a single evaluation step

$$e \to e'$$

For example: $((4+5) * 10 + 2) \rightarrow (9 * 10 + 2)$

How can we find the right place at which to make the step?

Use a continuation K: $e \triangleright K$ A stack of partial computations.

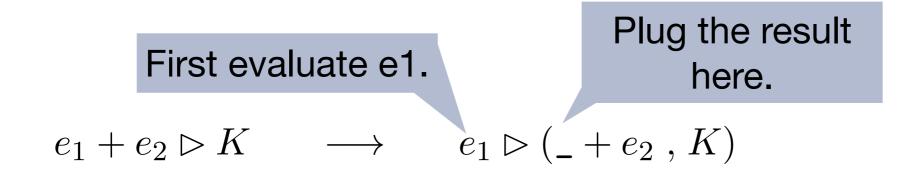
"Evaluate expression e and pass the result to K"

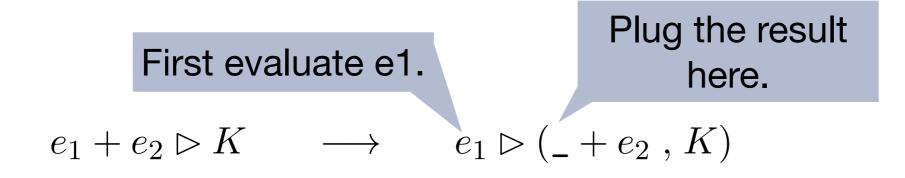
The continuation has a 'hole' for the result value of e.

$e_1 + e_2 \triangleright K \longrightarrow e_1 \triangleright (- + e_2, K)$

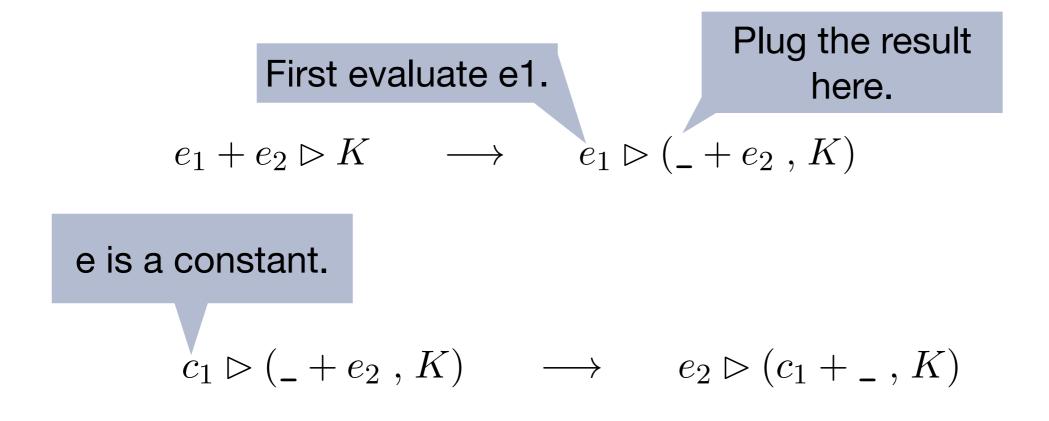
First evaluate e1.

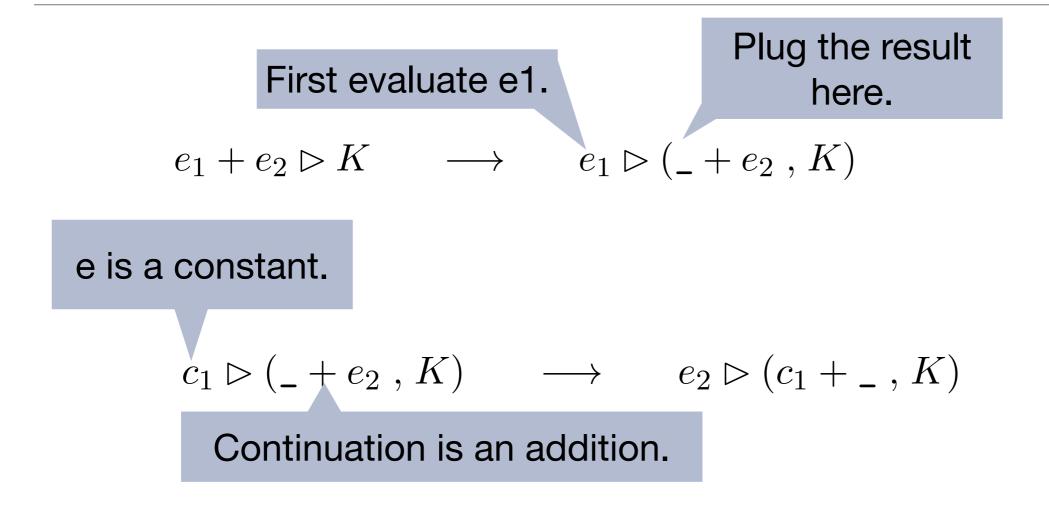
 $e_1 + e_2 \triangleright K \longrightarrow e_1 \triangleright (- + e_2, K)$

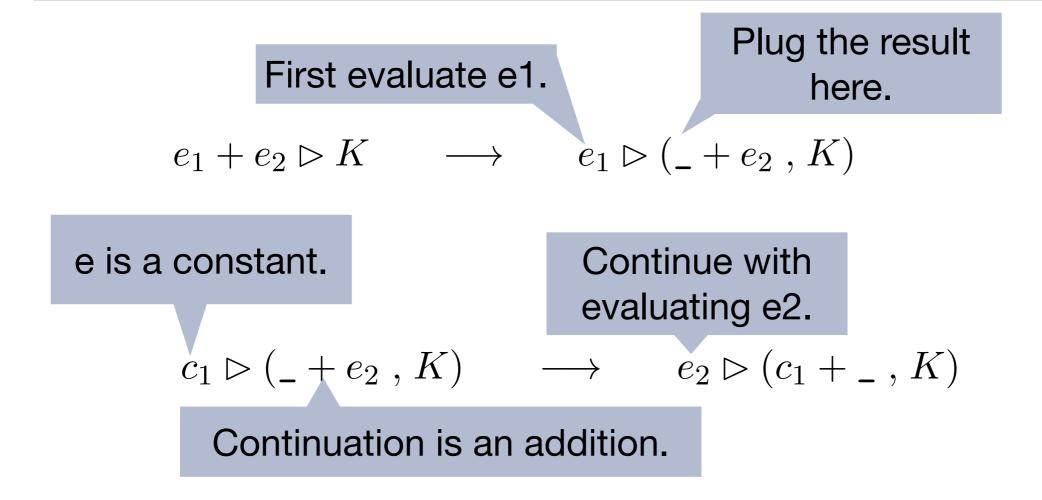


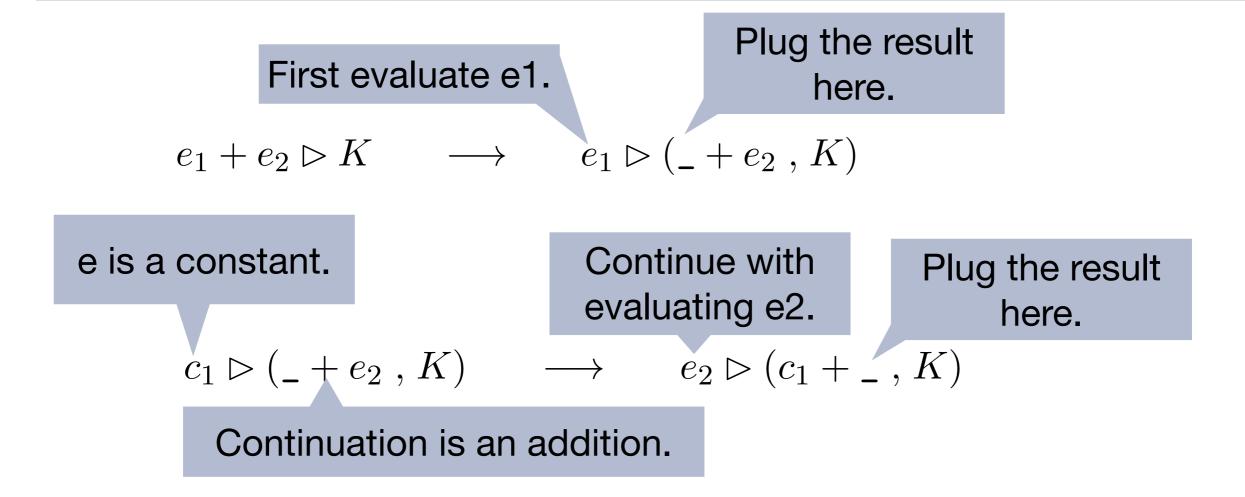


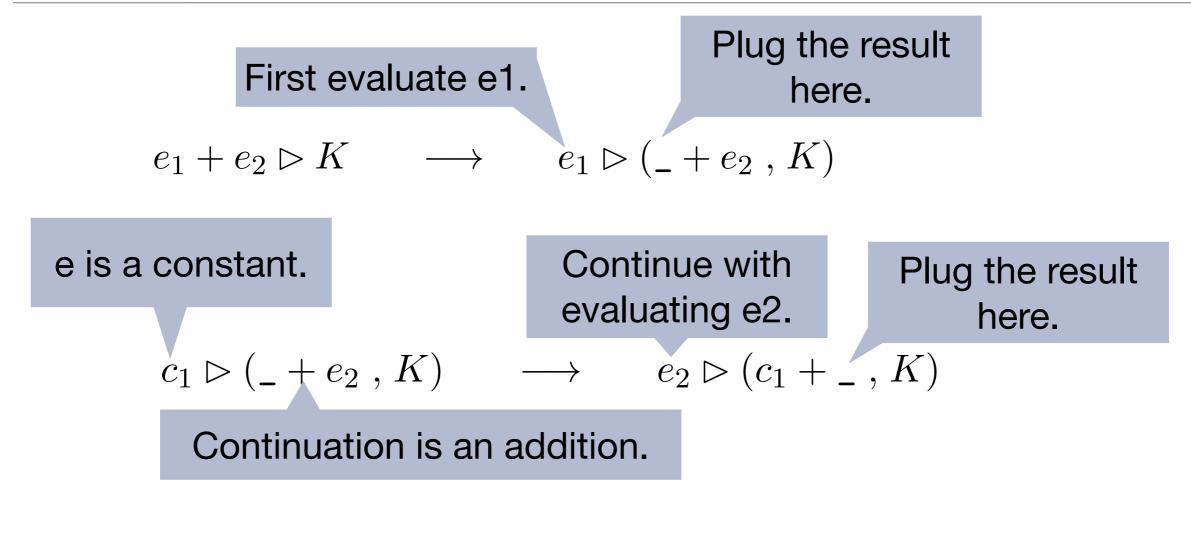
$$c_1 \triangleright (-+e_2, K) \longrightarrow e_2 \triangleright (c_1 + -, K)$$



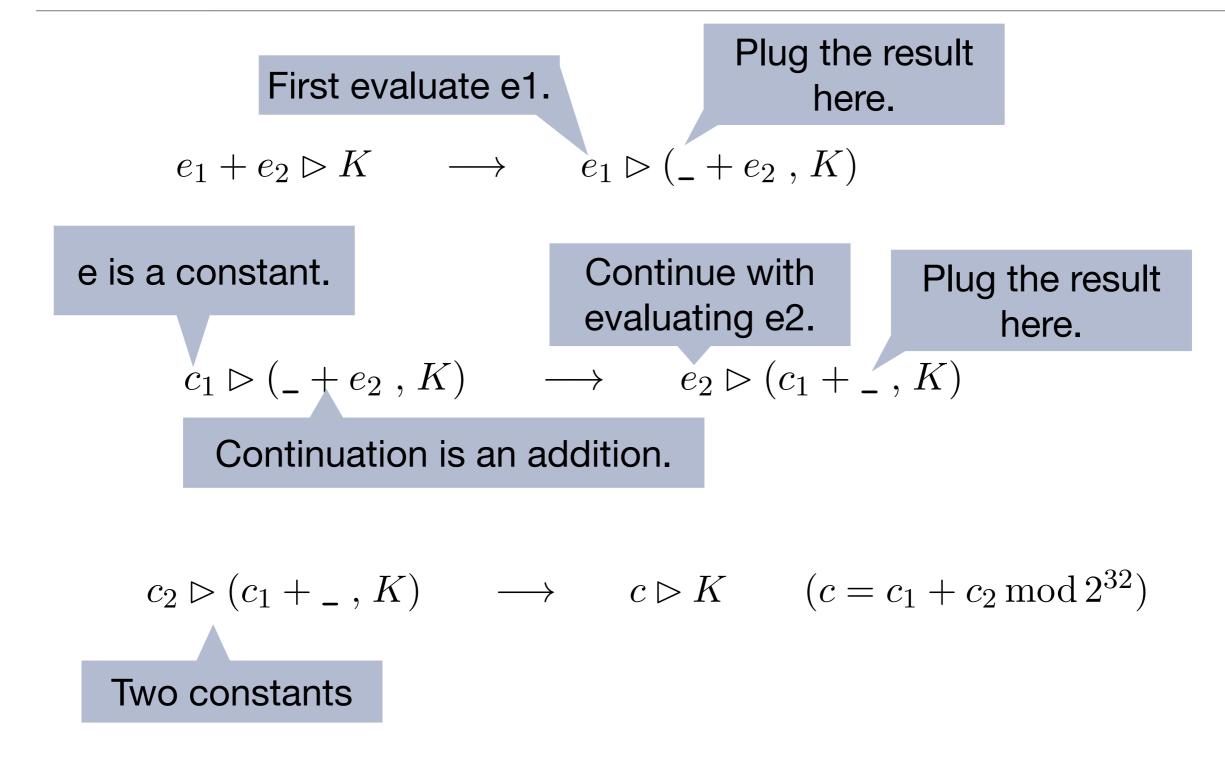


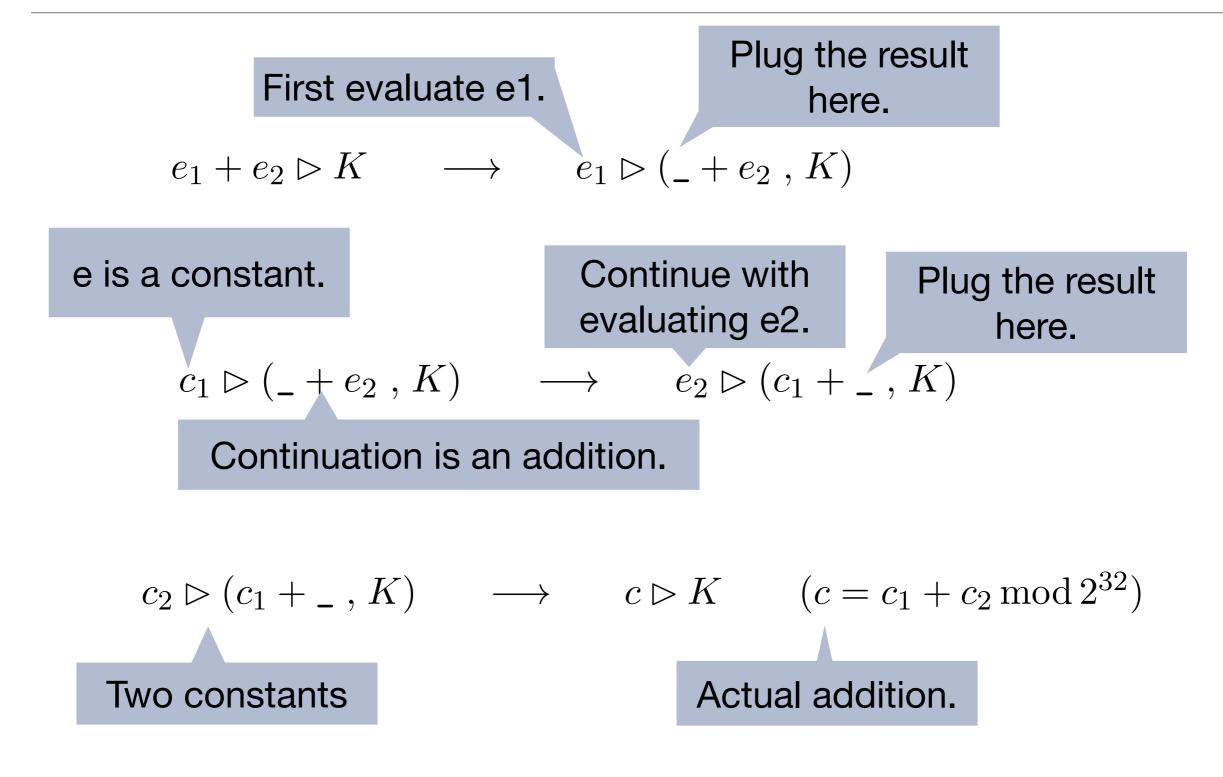






$$c_2 \triangleright (c_1 + _, K) \longrightarrow c \triangleright K \qquad (c = c_1 + c_2 \mod 2^{32})$$





Evaluation Rules: Binary Operations

Arithmetic operations are treated like addition

 $e_{1} \oplus e_{2} \triangleright K \longrightarrow e_{1} \triangleright (_ \oplus e_{2}, K)$ $c_{1} \triangleright (_ \oplus e_{2}, K) \longrightarrow e_{2} \triangleright (c_{1} \oplus _, K)$ $c_{2} \triangleright (c_{1} \oplus _, K) \longrightarrow c \triangleright K \qquad (c = c_{1} \oplus c_{2} \mod 2^{32})$

Arithmetic is modulo 2³² to match our x86 architecture

Evaluation Rules: Binary Operations

Arithmetic operations are treated like addition

 $e_{1} \oplus e_{2} \triangleright K \longrightarrow e_{1} \triangleright (_ \oplus e_{2}, K)$ $c_{1} \triangleright (_ \oplus e_{2}, K) \longrightarrow e_{2} \triangleright (c_{1} \oplus _, K)$ $c_{2} \triangleright (c_{1} \oplus _, K) \longrightarrow c \triangleright K \qquad (c = c_{1} \oplus c_{2} \mod 2^{32})$

Arithmetic is modulo 2³² to match our x86 architecture

What about effects?

Evaluation Rules: Binops with Effects

In case of an arithmetic exception: Abort the computation and report and error

$$e_{1} \oslash e_{2} \triangleright K \longrightarrow e_{1} \triangleright (_ \oslash e_{2}, K)$$

$$c_{1} \triangleright (_ \oslash e_{2}, K) \longrightarrow e_{2} \triangleright (c_{1} \oslash _, K)$$

$$c_{2} \triangleright (c_{1} \oslash _, K) \longrightarrow c \triangleright K \qquad (c = c_{1} \oslash c_{2})$$

$$c_{2} \triangleright (c_{1} \oslash _, K) \longrightarrow \text{exception(arith)} \qquad (c_{1} \oslash c_{2} \text{ undefined})$$

There is no rule for further evaluating an exception.

 $((4+5)*10)+2 \triangleright \cdot$

$$((4+5)*10) + 2 \vartriangleright \cdot$$
$$\longrightarrow (4+5)*10 \Join -+2$$

$$((4+5)*10) + 2 \triangleright \cdot$$

$$\longrightarrow (4+5)*10 \qquad \triangleright -+2$$

$$\longrightarrow 4+5 \qquad \triangleright -*10, -+2$$

Evaluation Rules: End of and Evaluation

If we reach a constant and the empty continuation then we stop

$$c \triangleright \cdot \quad \longrightarrow \quad \mathsf{value}(c)$$

Evaluation Rules: Boolean Expressions

$$e_1 \&\& e_2 \triangleright K \longrightarrow e_1 \triangleright (_\&\& e_2, K)$$
false $\triangleright (_\&\& e_2, K) \longrightarrow false \triangleright K$
true $\triangleright (_\&\& e_2, K) \longrightarrow e_2 \triangleright K$

true and false are also values

(We could also use 1 and 0 but distinguishing helps detect errors.)

Evaluation Rules: Boolean Expressions

$$e_1 \&\& e_2 \triangleright K \longrightarrow e_1 \triangleright (_\&\& e_2, K)$$
false $\triangleright (_\&\& e_2, K) \longrightarrow false \triangleright K \longrightarrow cutting.$

$$true \triangleright (_\&\& e_2, K) \longrightarrow e_2 \triangleright K$$

true and false are also values

(We could also use 1 and 0 but distinguishing helps detect errors.)

How do we evaluate variable?

How do we evaluate variable?

$$x \triangleright K \longrightarrow ?$$

How do we evaluate variable?

$$x \triangleright K \longrightarrow ?$$

Need to have an environment that maps variables to values

How do we evaluate variable?

$$x \triangleright K \longrightarrow ?$$

Need to have an environment that maps variables to values

 $\eta ::= \cdot \mid \eta, x \mapsto v$

How do we evaluate variable?

$$x \triangleright K \longrightarrow ?$$

Need to have an environment that maps variables to values

$$\eta ::= \cdot \mid \eta, x \mapsto v$$

The machine state consists now of an expression, a continuation, and an environment

How do we evaluate variable?

$$x \triangleright K \longrightarrow ?$$

Need to have an environment that maps variables to values

$$\eta ::= \cdot \mid \eta, x \mapsto v$$

The machine state consists now of an expression, a continuation, and an environment

 $\eta \vdash e \vartriangleright K$

How do we evaluate variable?

$$x \triangleright K \longrightarrow ?$$

Integers or booleans.

Need to have an environment that maps variables to values

 $\eta ::= \cdot \mid \eta, x \mapsto v$

The machine state consists now of an expression, a continuation, and an environment

 $\eta \vdash e \vartriangleright K$

The rules we have seen so far just carry over

$$\eta \vdash e_1 \oplus e_2 \triangleright K \longrightarrow \eta \vdash e_1 \triangleright (_ \oplus e_2, K)$$

$$\eta \vdash c_1 \triangleright (_ \oplus e_2, K) \longrightarrow \eta \vdash e_2 \triangleright (c_1 \oplus _, K)$$

$$\eta \vdash c_2 \triangleright (c_1 \oplus _, K) \longrightarrow \eta \vdash c \triangleright K \qquad (c = c_1 \oplus c_2 \mod 2^{32})$$

The rules we have seen so far just carry over

$$\eta \vdash e_1 \oplus e_2 \triangleright K \qquad \longrightarrow \qquad \eta \vdash e_1 \triangleright (_ \oplus e_2 , K)$$

$$\eta \vdash c_1 \triangleright (_ \oplus e_2 , K) \qquad \longrightarrow \qquad \eta \vdash e_2 \triangleright (c_1 \oplus _ , K)$$

$$\eta \vdash c_2 \triangleright (c_1 \oplus _ , K) \qquad \longrightarrow \qquad \eta \vdash c \triangleright K \qquad (c = c_1 \oplus c_2 \mod 2^{32})$$

Variables are simply looked up

$$\eta \vdash x \vartriangleright K \qquad \longrightarrow \qquad \eta \vdash \eta(x) \vartriangleright K$$

The rules we have seen so far just carry over

$$\eta \vdash e_1 \oplus e_2 \triangleright K \qquad \longrightarrow \qquad \eta \vdash e_1 \triangleright (_ \oplus e_2 , K)$$

$$\eta \vdash c_1 \triangleright (_ \oplus e_2 , K) \qquad \longrightarrow \qquad \eta \vdash e_2 \triangleright (c_1 \oplus _ , K)$$

$$\eta \vdash c_2 \triangleright (c_1 \oplus _ , K) \qquad \longrightarrow \qquad \eta \vdash c \triangleright K \qquad (c = c_1 \oplus c_2 \mod 2^{32})$$

Variables are simply looked up

$$\eta \vdash x \vartriangleright K \qquad \longrightarrow \qquad \eta \vdash \eta(x) \vartriangleright K$$

The environment never changes when evaluating expressions

Executions of statements don't pass values to the continuation

Statements have usually an effect on the environment

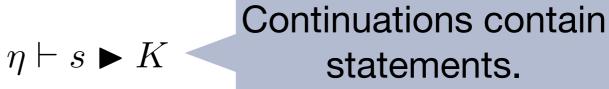
Machine configurations:

 $\eta \vdash s \blacktriangleright K$

Executions of statements don't pass values to the continuation

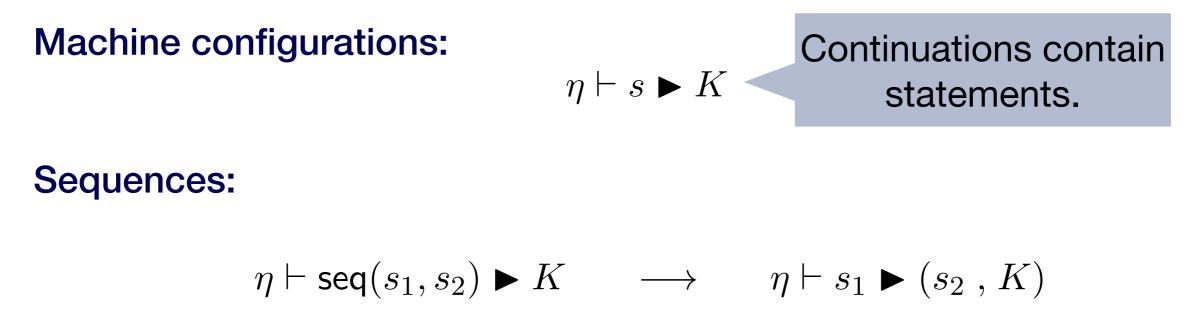
Statements have usually an effect on the environment

Machine configurations:



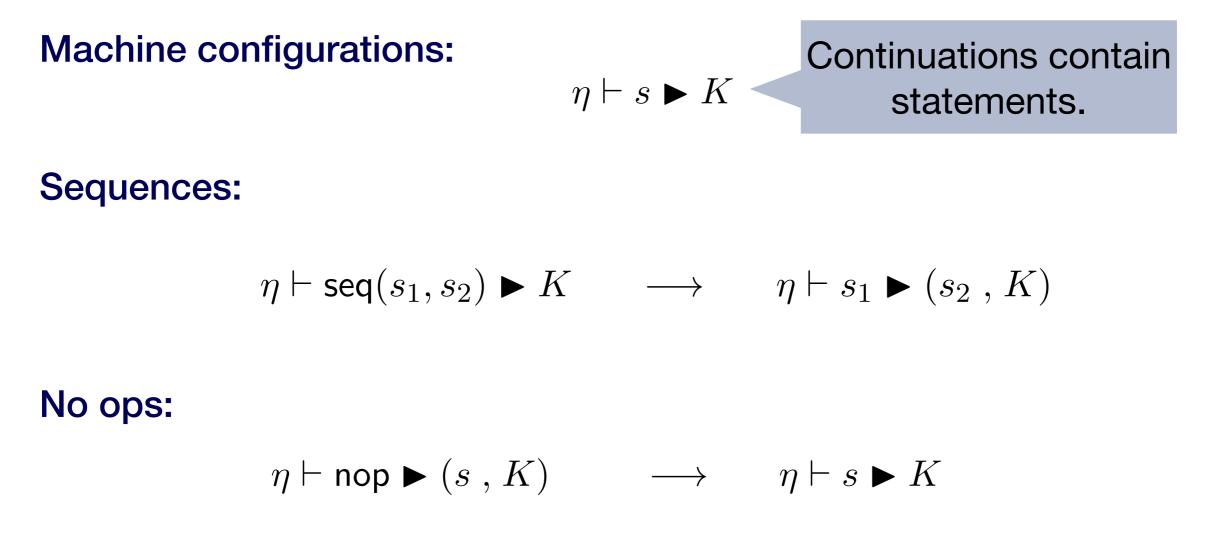
Executions of statements don't pass values to the continuation

Statements have usually an effect on the environment



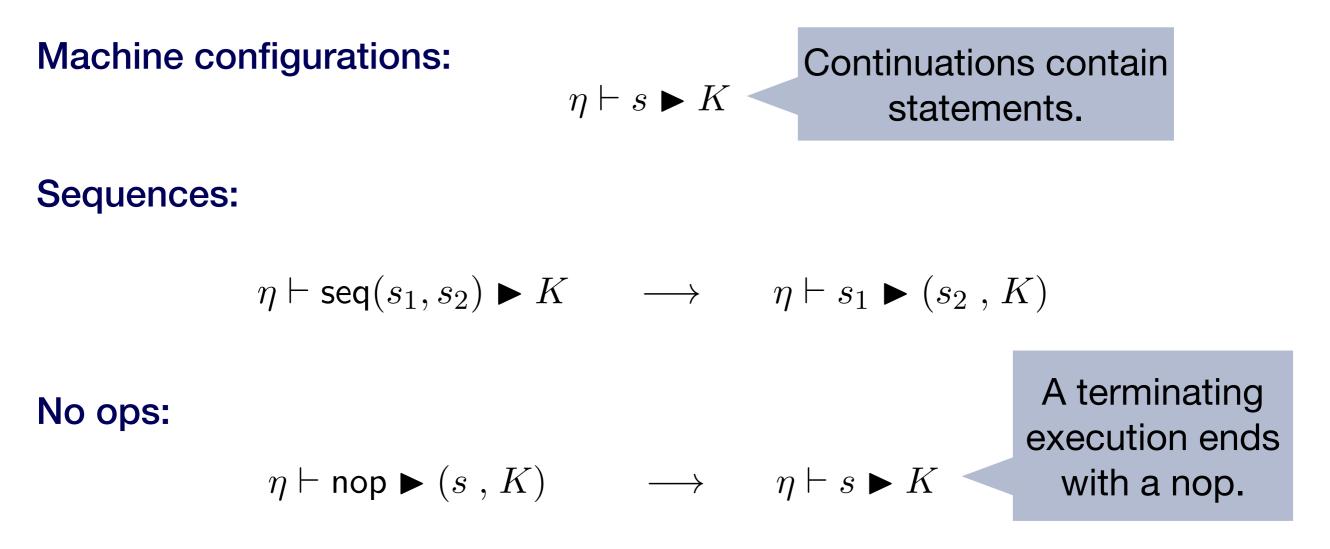
Executions of statements don't pass values to the continuation

Statements have usually an effect on the environment



Executions of statements don't pass values to the continuation

Statements have usually an effect on the environment



Interaction with expressions is straightforward

Assignments:

$$\eta \vdash \operatorname{assign}(x, e) \blacktriangleright K \qquad \longrightarrow \qquad \eta \vdash e \triangleright (\operatorname{assign}(x, _), K)$$
$$\eta \vdash v \triangleright (\operatorname{assign}(x, _), K) \qquad \longrightarrow$$

Interaction with expressions is straightforward

Assignments:

$$\begin{split} \eta \vdash \operatorname{assign}(x, e) \blacktriangleright K & \longrightarrow & \eta \vdash e \triangleright (\operatorname{assign}(x, _), K) \\ \eta \vdash v \triangleright (\operatorname{assign}(x, _), K) & \longrightarrow & \eta[x \mapsto v] \vdash \operatorname{nop} \blacktriangleright K \end{split}$$

Interaction with expressions is straightforward

Assignments:

$$\eta \vdash \operatorname{assign}(x, e) \blacktriangleright K \longrightarrow \eta \vdash e \triangleright (\operatorname{assign}(x, _), K)$$
$$\eta \vdash v \triangleright (\operatorname{assign}(x, _), K) \longrightarrow \eta[x \mapsto v] \vdash \operatorname{nop} \blacktriangleright K$$
$$Update the environment with new mapping.$$

Conditionals:

$$\eta \vdash \mathsf{if}(e, s_1, s_2) \blacktriangleright K \qquad \longrightarrow \qquad \eta \vdash e \triangleright (\mathsf{if}(_, s_1, s_2), K)$$

$$\eta \vdash \mathsf{true} \triangleright (\mathsf{if}(_, s_1, s_2), K) \longrightarrow \eta \vdash s_1 \blacktriangleright K$$

 $\eta \vdash \mathsf{false} \rhd (\mathsf{if}(_, s_1, s_2), K) \longrightarrow \eta \vdash s_2 \triangleright K$

Loops:

$$\eta \vdash \mathsf{while}(e, s) \triangleright K \longrightarrow ?$$

Loops:

$$\eta \vdash \mathsf{while}(e, s) \triangleright K \longrightarrow ?$$

Not that the following statements are equivalent:

 $\mathsf{while}(e,s) \equiv \mathsf{if}(e,\mathsf{seq}(s,\mathsf{while}(e,s)),\mathsf{nop})$

Loops:

$$\eta \vdash \mathsf{while}(e, s) \triangleright K \longrightarrow ?$$

Not that the following statements are equivalent:

while
$$(e, s) \equiv if(e, seq(s, while(e, s)), nop)$$

 $\eta \vdash \mathsf{while}(e,s) \blacktriangleright K \qquad \longrightarrow \qquad \eta \vdash \mathsf{if}(e,\mathsf{seq}(s,\mathsf{while}(e,s)),\mathsf{nop}) \blacktriangleright K$

Loops:

$$\eta \vdash \mathsf{while}(e, s) \triangleright K \longrightarrow ?$$

Not that the following statements are equivalent:

while
$$(e, s) \equiv if(e, seq(s, while(e, s)), nop)$$

$$\eta \vdash \mathsf{while}(e, s) \blacktriangleright K \qquad \longrightarrow \qquad \eta \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{nop}) \blacktriangleright K$$

Non-termination:

$$s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots$$

We can make an infinite number of steps without reaching a final state

Assertions:

$$\begin{split} \eta \vdash \mathsf{assert}(e) \blacktriangleright K & \longrightarrow & \eta \vdash e \triangleright (\mathsf{assert}(_), K) \\ \eta \vdash \mathsf{true} \triangleright (\mathsf{assert}(_), K) & \longrightarrow & \eta \vdash \mathsf{nop} \blacktriangleright K \\ \eta \vdash \mathsf{false} \triangleright (\mathsf{assert}(_), K) & \longrightarrow & \mathsf{exception}(\mathsf{abort}) \end{split}$$

Assertions:

$$\begin{split} \eta \vdash \mathsf{assert}(e) \blacktriangleright K & \longrightarrow & \eta \vdash e \triangleright (\mathsf{assert}(_), K) \\ \eta \vdash \mathsf{true} \triangleright (\mathsf{assert}(_), K) & \longrightarrow & \eta \vdash \mathsf{nop} \blacktriangleright K \\ \eta \vdash \mathsf{false} \triangleright (\mathsf{assert}(_), K) & \longrightarrow & \mathsf{exception}(\mathsf{abort}) \end{split}$$

Declarations:

$$\eta \vdash \mathsf{decl}(x,\tau,s) \blacktriangleright K \qquad \longrightarrow \qquad \eta[x \mapsto \mathsf{nothing}] \vdash s \blacktriangleright K$$

Assertions:

$$\begin{split} \eta \vdash \mathsf{assert}(e) \blacktriangleright K & \longrightarrow & \eta \vdash e \triangleright (\mathsf{assert}(_), K) \\ \eta \vdash \mathsf{true} \triangleright (\mathsf{assert}(_), K) & \longrightarrow & \eta \vdash \mathsf{nop} \blacktriangleright K \\ \eta \vdash \mathsf{false} \triangleright (\mathsf{assert}(_), K) & \longrightarrow & \mathsf{exception}(\mathsf{abort}) \end{split}$$

Declarations:

$$\eta \vdash \mathsf{decl}(x,\tau,s) \blacktriangleright K \longrightarrow \eta[x \mapsto \mathsf{nothing}] \vdash s \blacktriangleright K$$

If C0 had shadowing then we would have to be careful here.

Assertions:

- $$\begin{split} \eta \vdash \mathsf{assert}(e) \blacktriangleright K & \longrightarrow & \eta \vdash e \rhd (\mathsf{assert}(_), K) \\ \eta \vdash \mathsf{true} \triangleright (\mathsf{assert}(_), K) & \longrightarrow & \eta \vdash \mathsf{nop} \blacktriangleright K \end{split}$$
- $\eta \vdash \mathsf{false} \rhd (\mathsf{assert}(_), K) \longrightarrow \mathsf{exception}(\mathsf{abort})$

Declarations:

$$\eta \vdash \mathsf{decl}(x,\tau,s) \blacktriangleright K \qquad \longrightarrow \qquad \eta[x \mapsto \mathsf{nothing}] \vdash s \blacktriangleright K$$

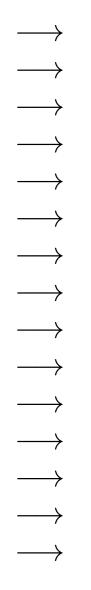
Final states:

exception(E) nop \blacktriangleright .

If C0 had shadowing then we would have to be careful here.

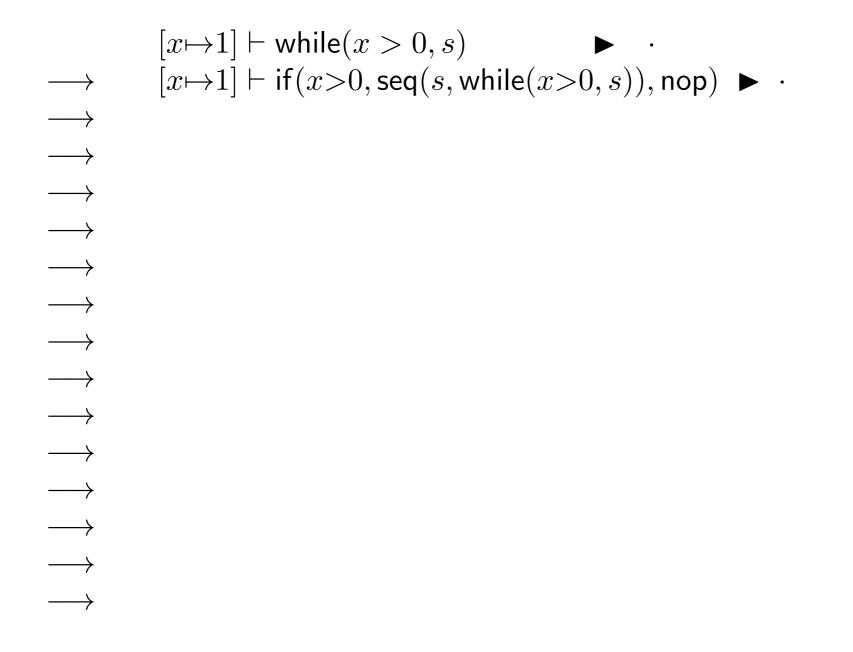
Example: Infinite Loop

 $\mathsf{while}(x \ > \ 0, \mathsf{assign}(x, x + 1)) \qquad \eta \ = \ [x \mapsto 1]$



 $\mathsf{while}(x \ > \ 0, \mathsf{assign}(x, x + 1)) \qquad \eta \ = \ [x \mapsto 1]$

 $[x \mapsto 1] \vdash \mathsf{while}(x > 0, s)$ ► · \longrightarrow \longrightarrow



while $(x > 0, \operatorname{assign}(x, x + 1))$ $\eta = [x \mapsto 1]$

$$[x \mapsto 1] \vdash \mathsf{while}(x > 0, s) \qquad \blacktriangleright \qquad \cdot$$

$$[x \mapsto 1] \vdash \mathsf{if}(x > 0, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \qquad \blacktriangleright \qquad \cdot$$

$$[x \mapsto 1] \vdash x > 0 \qquad \rhd \qquad \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop})$$

$$\rightarrow \qquad \rightarrow$$

 $\mathsf{while}(x > 0, \mathsf{assign}(x, x + 1)) \qquad \eta = [x \mapsto 1]$

$$\begin{array}{cccc} [x \mapsto 1] \vdash \mathsf{while}(x > 0, s) & \blacktriangleright & \cdot \\ \hline & & [x \mapsto 1] \vdash \mathsf{if}(x > 0, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) & \blacktriangleright & \cdot \\ \hline & & & [x \mapsto 1] \vdash x > 0 & \triangleright & \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\ \hline & & & & [x \mapsto 1] \vdash x & \triangleright & _ > 0; \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\ \hline & & & & & [x \mapsto 1] \vdash 1 & \triangleright & _ > 0; \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\ \hline & & & & & & [x \mapsto 1] \vdash 0 & \triangleright & 1 > _; \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \end{array}$$

$$\rightarrow$$

 \longrightarrow

 $\begin{array}{c} \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \end{array}$

 \longrightarrow

 \longrightarrow

$$\begin{array}{cccc} [x\mapsto 1] \vdash \mathsf{while}(x > 0, s) & \blacktriangleright & \cdot \\ & \longrightarrow & [x\mapsto 1] \vdash \mathsf{if}(x > 0, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) & \blacktriangleright & \cdot \\ & \longrightarrow & [x\mapsto 1] \vdash x > 0 & \rhd & \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\ & \longrightarrow & [x\mapsto 1] \vdash x & \rhd & _ > 0; \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\ & \longrightarrow & [x\mapsto 1] \vdash 1 & \rhd & _ > 0; \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\ & \rightarrow & [x\mapsto 1] \vdash 0 & \rhd & 1 > _; \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \\ & \rightarrow & [x\mapsto 1] \vdash \mathsf{true} & \triangleright & \mathsf{if}(_, \mathsf{seq}(s, \mathsf{while}(x > 0, s)), \mathsf{nop}) \end{array}$$

- \longrightarrow
- $\begin{array}{c} \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \end{array}$

- $\begin{array}{c} \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \end{array}$

 $\mathsf{while}(x > 0, \mathsf{assign}(x, x + 1)) \qquad \eta = [x \mapsto 1]$

 $\begin{array}{c} \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \\ \longrightarrow \end{array}$

 \longrightarrow

 \longrightarrow

 \rightarrow

while $(x > 0, \operatorname{assign}(x, x + 1))$ $\eta = [x \mapsto 1]$

• • •

Functions

What needs to happen at a function call?

• Evaluate the arguments in left-to-right order

- Evaluate the arguments in left-to-right order
- Save the environment of the caller to continue the execution after the function call

- Evaluate the arguments in left-to-right order
- Save the environment of the caller to continue the execution after the function call
- Save the continuation of the caller

- Evaluate the arguments in left-to-right order
- Save the environment of the caller to continue the execution after the function call
- Save the continuation of the caller
- Execute the body of the callee in a new environment that maps the formal parameters to the argument values

- Evaluate the arguments in left-to-right order
- Save the environment of the caller to continue the execution after the function call
- Save the continuation of the caller
- Execute the body of the callee in a new environment that maps the formal parameters to the argument values
- Pass the return value to the environment of the caller

We need to keep track of continuations and environment in stack frames

Call stack:

$$S ::= \cdot \mid S , \langle \eta, K \rangle$$

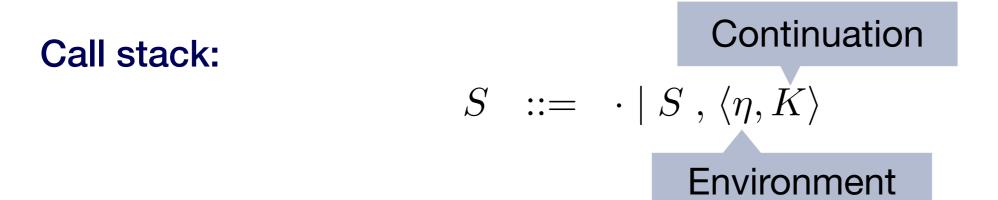
We need to keep track of continuations and environment in stack frames

Call stack:

$$S ::= \cdot \mid S , \langle \eta, K \rangle$$

Environment

We need to keep track of continuations and environment in stack frames



We need to keep track of continuations and environment in stack frames

Call stack:

$$S ::= \cdot \mid S, \langle \eta, K \rangle$$

Environment

Configurations:

Evaluation $S ; \eta \vdash e \triangleright K$ Execution $S ; \eta \vdash s \blacktriangleright K$

We need to keep track of continuations and environment in stack frames

Call stack:

$$S ::= \cdot \mid S, \langle \eta, K \rangle$$

Environment

Configurations:

Evaluation	$S ; \eta \vdash e \rhd K$
Execution	$S \ ; \eta \vdash s \blacktriangleright K$

Existing rules can be lifted to the new configurations by passing through the call stack

We only show the special case of 0 and 2 arguments

n args is similar.

We only show the special case of 0 and 2 arguments

We only show the special case of 0 and 2 arguments

No arguments:

 $\begin{array}{ll} S : \eta \vdash f() \triangleright K & \longrightarrow & (S , \langle \eta, K \rangle) : \cdot \vdash s \blacktriangleright \cdot \\ (given that f is defined as \ f()\{s\}) & \end{array}$

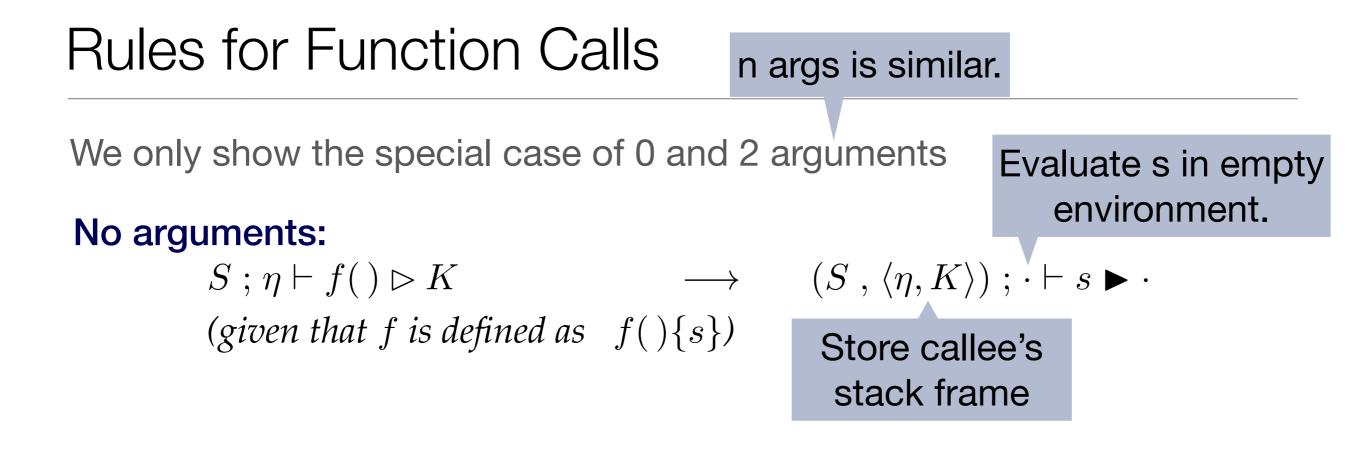
We only show the special case of 0 and 2 arguments

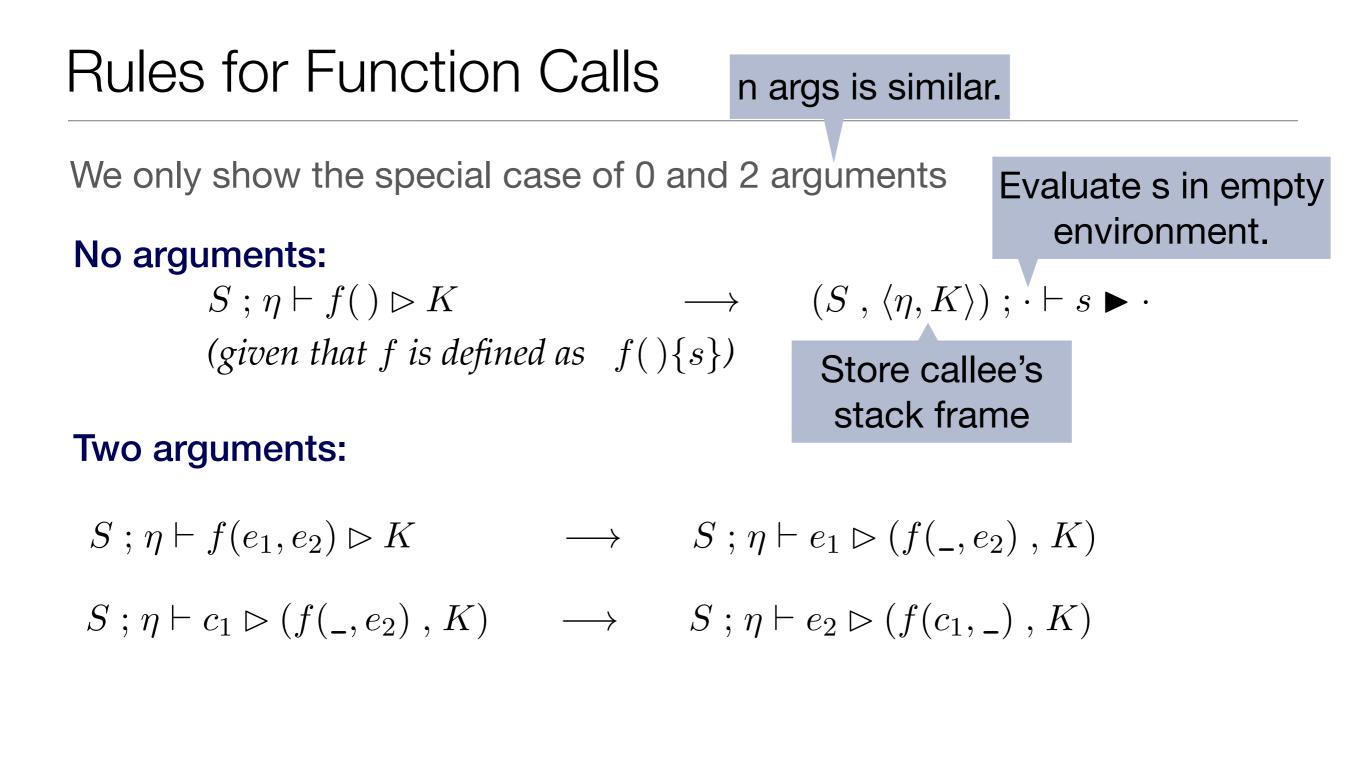
No arguments:

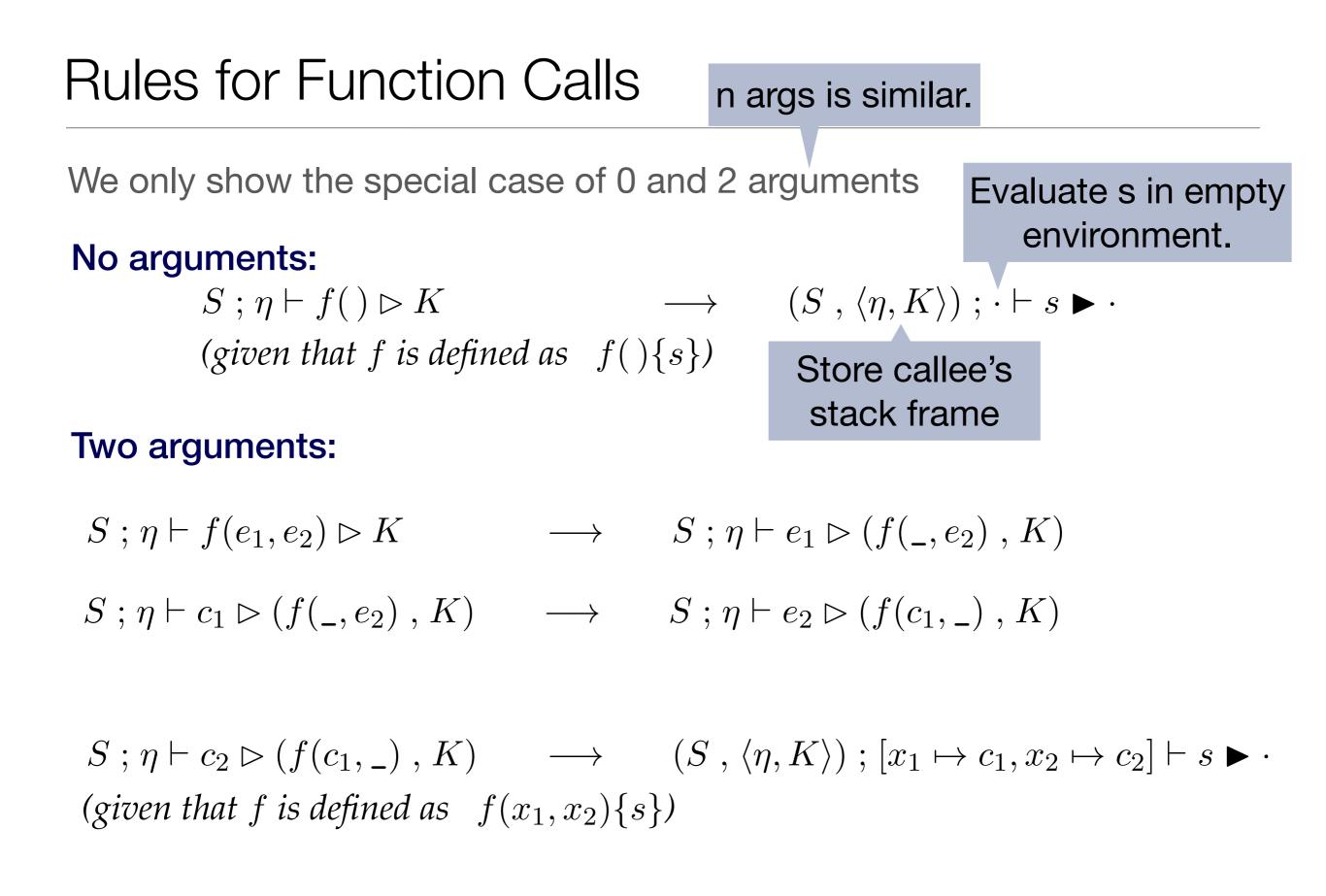
 $S ; \eta \vdash f() \triangleright K \longrightarrow$ (given that f is defined as $f()\{s\}$)

 $(S, \langle \eta, K \rangle); \cdot \vdash s \blacktriangleright \cdot$

Store callee's stack frame







Need to restore continuation and environment and pass return value

 $S ; \eta \vdash \mathsf{return}(e) \blacktriangleright K$

Need to restore continuation and environment and pass return value

$$S ; \eta \vdash \mathsf{return}(e) \blacktriangleright K \qquad \longrightarrow \qquad S ; \eta \vdash e \triangleright (\mathsf{return}(_), K)$$

Need to restore continuation and environment and pass return value

$$\begin{array}{ll} S ; \eta \vdash \operatorname{return}(e) \blacktriangleright K & \longrightarrow & S ; \eta \vdash e \triangleright (\operatorname{return}(_), K) \\ \\ S , \langle \eta', K' \rangle ; \eta \vdash v \triangleright (\operatorname{return}(_), K) \end{array}$$

Need to restore continuation and environment and pass return value

$$\begin{array}{lll} S \ ; \ \eta \vdash \mathsf{return}(e) \blacktriangleright K & \longrightarrow & S \ ; \ \eta \vdash e \triangleright (\mathsf{return}(_) \ , K) \\ \\ S \ , \ \langle \eta', K' \rangle \ ; \ \eta \vdash v \triangleright (\mathsf{return}(_) \ , K) & \longrightarrow & S \ ; \ \eta' \vdash v \triangleright K' \end{array}$$

Need to restore continuation and environment and pass return value

$$\begin{array}{lll} S \ ; \eta \vdash \operatorname{return}(e) \blacktriangleright K & \longrightarrow & S \ ; \eta \vdash e \triangleright (\operatorname{return}(_) \ , K) \\ \\ S \ , \langle \eta', K' \rangle \ ; \eta \vdash v \triangleright (\operatorname{return}(_) \ , K) & \longrightarrow & S \ ; \eta' \vdash v \triangleright K' \end{array}$$

Special case: returning void

$$S, \langle \eta', K' \rangle; \eta \vdash \mathsf{nop} \blacktriangleright \cdot \qquad \longrightarrow \qquad S; \eta' \vdash \mathsf{nothing} \triangleright K'$$

Need to restore continuation and environment and pass return value

 $S : \eta \vdash \operatorname{return}(e) \blacktriangleright K \qquad \longrightarrow \qquad S : \eta \vdash e \triangleright (\operatorname{return}(_), K)$ $S , \langle \eta', K' \rangle : \eta \vdash v \triangleright (\operatorname{return}(_), K) \qquad \longrightarrow \qquad S : \eta' \vdash v \triangleright K'$ Special case: returning void $S , \langle \eta', K' \rangle : \eta \vdash \operatorname{nop} \triangleright \cdot \qquad \longrightarrow \qquad S : \eta' \vdash \operatorname{nothing} \triangleright K'$

Need to restore continuation and environment and pass return value

 $\begin{array}{cccc} S \ ; \eta \vdash \operatorname{return}(e) \blacktriangleright K & \longrightarrow & S \ ; \eta \vdash e \triangleright (\operatorname{return}(_) \ , K) \\ S \ , \langle \eta', K' \rangle \ ; \eta \vdash v \triangleright (\operatorname{return}(_) \ , K) & \longrightarrow & S \ ; \eta' \vdash v \triangleright K' \\ \end{array}$ $\begin{array}{cccc} \text{Special case: returning void} & & \\ S \ , \langle \eta', K' \rangle \ ; \eta \vdash \operatorname{nop} \blacktriangleright & & \longrightarrow & S \ ; \eta' \vdash \operatorname{nothing} \triangleright K' \\ S \ , \langle \eta', K' \rangle \ ; \eta \vdash \operatorname{nop} \blacktriangleright & & \longrightarrow & S \ ; \eta' \vdash \operatorname{nothing} \triangleright K' \\ \end{array}$

Need to restore continuation and environment and pass return value

 $\begin{array}{cccc} S \ ; \eta \vdash \operatorname{return}(e) \blacktriangleright K & \longrightarrow & S \ ; \eta \vdash e \triangleright (\operatorname{return}(_) \ , K) \\ S \ , \langle \eta', K' \rangle \ ; \eta \vdash v \triangleright (\operatorname{return}(_) \ , K) & \longrightarrow & S \ ; \eta' \vdash v \triangleright K' \\ \end{array}$ $\begin{array}{cccc} \text{Special case: returning void} & & \\ S \ , \langle \eta', K' \rangle \ ; \eta \vdash \operatorname{nop} \blacktriangleright & & \longrightarrow & S \ ; \eta' \vdash \operatorname{nothing} \triangleright K' \\ & & \\$

Alternative: elaborate each function that returns void with return(nothing) statements.

Execution of the Main Function

How can we execute a program?

Execution of the Main Function

How can we execute a program?

```
\cdot; \cdot \vdash main() \triangleright \cdot (initial state)
```

Execution of the Main Function

How can we execute a program?

 $\cdot ; \cdot \vdash \mathsf{main}() \vartriangleright \cdot \quad (\text{initial state})$ $\cdot ; \eta \vdash c \vartriangleright \cdot \quad \longrightarrow \quad \mathsf{value}(c) \qquad (\text{final state})$

Statics, Dynamics, and Safety

- S; $\eta \vdash e \triangleright K$ Evaluating the expression e with the continuation K
- $S ; \eta \vdash s \triangleright K$ Evaluating the statement *s* with the continuation *K*
- value(c) Final state, return a value
- exception(E) Final state, report an error

- $S ; \eta \vdash e \triangleright K$ Evaluating the expression *e* with the continuation *K*
- $S ; \eta \vdash s \triangleright K$ Evaluating the statement *s* with the continuation *K*
- value(c) Final state, return a value
- exception(E) Final state, report an error

What do we expect from the transitions?

$S; \eta \vdash e \triangleright K$ –	Evaluating the expression e with the continuation K
---------------------------------------	---

 $S ; \eta \vdash s \triangleright K$ – Evaluating the statement *s* with the continuation *K*

value(c) – Final state, return a value

exception(E) – Final state, report an error

What do we expect from the transitions?

There shouldn't be more steps after reaching a final state

 $S ; \eta \vdash s \triangleright K$ – Evaluating the statement *s* with the continuation *K*

value(c) – Final state, return a value

exception(E) – Final state, report an error

What do we expect from the transitions?

There shouldn't be more steps after reaching a final state

The language should be deterministic: there at most one transition per state



There are many non-final states that don't have transitions, e.g.

There are many non-final states that don't have transitions, e.g.

$$S; \eta \vdash 42 \rhd (\mathsf{if}(_, s_1, s_2); K)$$

There are many non-final states that don't have transitions, e.g.

 $S; \eta \vdash 42 \triangleright (\mathsf{if}(_, s_1, s_2); K) \qquad \quad \cdot; \cdot \vdash \mathsf{nop} \triangleright \cdot$

There are many non-final states that don't have transitions, e.g.

$$S; \eta \vdash 42 \triangleright (if(_, s_1, s_2); K)$$
 $\cdot; \cdot \vdash nop \blacktriangleright \cdot$
Stuck states.

There are many non-final states that don't have transitions, e.g.

$$S; \eta \vdash 42 \triangleright (if(_, s_1, s_2); K) \qquad \cdot; \cdot \vdash nop \blacktriangleright \cdot \qquad \text{The behavior of these states is undefined.}$$

There are many non-final states that don't have transitions, e.g.

$$S; \eta \vdash 42 \triangleright (if(_, s_1, s_2); K) \qquad :; \cdot \vdash nop \blacktriangleright \cdot \qquad \text{The behavior of these states is undefined.}$$

Central relationship between static and dynamic semantics:

Programs that are well-defined according to the static semantics should be free of undefined behavior.

There are many non-final states that don't have transitions, e.g.

$$S; \eta \vdash 42 \triangleright (if(_, s_1, s_2); K) \qquad \cdot; \cdot \vdash nop \blacktriangleright \cdot \qquad \text{The behavior of these states is undefined.}$$

Central relationship between static and dynamic semantics:

Programs that are well-defined according to the static semantics should be free of undefined behavior.

Theorem 1 (No undefined behavior) *If a program passes all the static semantics, and*

 $:: \vdash \mathsf{main}() \longrightarrow ST_1 \longrightarrow \ldots \longrightarrow ST_n$ then either ST_n is a final state or else ST_n is not-stuck because there exists a state ST'such that $ST_n \longrightarrow ST'$.

Well-typed programs don't go wrong!

There are many non-final states that don't have transitions, e.g.

$$S; \eta \vdash 42 \triangleright (if(_, s_1, s_2); K) \qquad \cdot; \cdot \vdash nop \blacktriangleright \cdot \qquad \text{The behavior of these states is undefined.}$$

Central relationship between static and dynamic semantics:

Programs that are well-defined according to the static semantics should be free of undefined behavior.

Theorem 1 (No undefined behavior) *If a program passes all the static semantics, and*

 $:: \vdash \min() \longrightarrow ST_1 \longrightarrow \dots \longrightarrow ST_n$ then either ST_n is a final state or else ST_n is not-stuck because there exists a state ST'such that $ST_n \longrightarrow ST'$.

Well-typed programs don't go wrong!

There are many non-final states that don't have transitions, e.g.

$$S; \eta \vdash 42 \triangleright (if(_, s_1, s_2); K) \qquad \cdot; \cdot \vdash nop \blacktriangleright \cdot \qquad \text{The behavior of these states is undefined.}$$

Central relationship between static and dynamic semantics:

Programs that are well-defined according to the static semantics should be free of undefined behavior.

Theorem 1 (No undefined behavior) *If a program passes all the static semantics, and*

 $:: \vdash \mathsf{main}() \longrightarrow ST_1 \longrightarrow \ldots \longrightarrow ST_n$ then either ST_n is a final state or else ST_n is not-stuck because there exists a state ST'such that $ST_n \longrightarrow ST'$.

How to prove this?

Well-typed programs don't go wrong!

There are many non-final states that don't have transitions, e.g.

$$S; \eta \vdash 42 \triangleright (if(_, s_1, s_2); K) \qquad \cdot; \cdot \vdash nop \blacktriangleright \cdot \qquad \mbox{The behavior of these states is undefined.} \label{eq:generalized_states}$$

Central relationship between static and dynamic semantics:

Programs that are well-defined according to the static semantics should be free of undefined behavior.

Theorem 1 (No undefined behavior) *If a program passes all the static semantics, and*

 $:: \vdash \mathsf{main}() \longrightarrow ST_1 \longrightarrow \ldots \longrightarrow ST_n$ then either ST_n is a final state or else ST_n is not-stuck because there exists a state ST'such that $ST_n \longrightarrow ST'$.

How to prove this?

15-312

Expressions	e	::=	$c \mid e_1 \odot e_2 \mid {\sf true} \mid {\sf false} \mid e_1$ && $e_2 \mid x \mid f(e_1, e_2) \mid f()$
Statements	S	::= 	$\begin{array}{l} nop \mid seq(s_1, s_2) \mid assign(x, e) \mid decl(x, \tau, s) \\ if(e, s_1, s_2) \mid while(e, s) \mid return(e) \mid assert(e) \end{array}$
Values	v	::=	$c \mid true \mid false \mid nothing$
Environments	η	::=	$\cdot \mid \eta, x \mapsto c$
Stacks	S	::=	$\cdot \mid S \;, \langle \eta, K angle$
Cont. frames	ϕ	::= 	$ _ \odot e \mid c \odot _ \mid _ \&\& e \mid f(_, e) \mid f(c, _) \\ s \mid \operatorname{assign}(x, _) \mid \operatorname{if}(_, s_1, s_2) \mid \operatorname{return}(_) \mid \operatorname{assert}(_) $
Continuations	K	::=	$\cdot \mid \phi \;, K$
Exceptions	E	::=	arith abort mem

Summary I

			All ops.
Expressions	ρ	••—	$c \mid e_1 \odot e_2 \mid true \mid false \mid e_1 \&\& e_2 \mid x \mid f(e_1, e_2) \mid f()$
	C	••	$c \mid c_1 \cup c_2 \mid c_1 dc \mid d_1 dc \mid c_1 dd c_2 \mid d \mid j (c_1, c_2) \mid j ()$
Statements	S	::=	$nop \mid seq(s_1, s_2) \mid assign(x, e) \mid decl(x, \tau, s)$
			$if(e, s_1, s_2) \mid while(e, s) \mid return(e) \mid assert(e)$
Values	v	::=	$c \mid true \mid false \mid nothing$
Environments	η	::=	$\cdot \mid \eta, x \mapsto c$
Stacks	S	::=	$\cdot \mid S \;, \langle \eta, K angle$
Cont. frames	ϕ	::=	$_\odot e \mid c \odot _ \mid _$ && $e \mid f(_, e) \mid f(c, _)$
			$s \mid \operatorname{assign}(x, _) \mid \operatorname{if}(_, s_1, s_2) \mid \operatorname{return}(_) \mid \operatorname{assert}(_)$
Continuations	K	::=	$\cdot \mid \phi \;, K$
Exceptions	E	::=	arith abort mem

Summary I

- $S ; \eta \vdash e_1 \odot e_2 \triangleright K$ $S ; \eta \vdash c_1 \triangleright (_ \odot e_2 , K)$ $S ; \eta \vdash c_2 \triangleright (c_1 \odot _, K)$ $S ; \eta \vdash c_2 \triangleright (c_1 \odot _, K)$
- $S ; \eta \vdash e_1 \&\& e_2 \triangleright K$ $S ; \eta \vdash \mathsf{false} \triangleright (_\&\& e_2 , K)$ $S ; \eta \vdash \mathsf{true} \triangleright (_\&\& e_2 , K)$

 $S ; \eta \vdash x \rhd K$

- $\begin{array}{ll} \longrightarrow & S ; \eta \vdash e_1 \triangleright (_ \odot e_2 , K) \\ \longrightarrow & S ; \eta \vdash e_2 \triangleright (c_1 \odot _ , K) \\ \longrightarrow & S ; \eta \vdash c \triangleright K & (c = c_1 \odot c_2) \\ \longrightarrow & \text{exception(arith)} & (c_1 \odot c_2 \text{ undefined}) \end{array}$
- $\longrightarrow \quad S ; \eta \vdash e_1 \triangleright (_\&\& e_2, K)$
- $\longrightarrow \qquad S \ ; \ \eta \vdash \mathsf{false} \vartriangleright K$
- \longrightarrow $S; \eta \vdash e_2 \triangleright K$
- $\longrightarrow S ; \eta \vdash \eta(x) \triangleright K$

Summary: Expressions

 $S : \eta \vdash seq(s_1, s_2) \triangleright K$ $S : \eta \vdash nop \triangleright (s, K)$ $S : \eta \vdash assign(x, e) \triangleright K$ $S : \eta \vdash c \triangleright (assign(x, _), K)$

 $S \ ; \eta \vdash \mathsf{decl}(x,\tau,s) \blacktriangleright K$

- $\begin{array}{l} S \ ; \eta \vdash \mathsf{assert}(e) \blacktriangleright K \\ S \ ; \eta \vdash \mathsf{true} \rhd (\mathsf{assert}(_) \ , K) \\ S \ ; \eta \vdash \mathsf{false} \rhd (\mathsf{assert}(_) \ , K) \end{array}$
- $S ; \eta \vdash if(e, s_1, s_2) \blacktriangleright K$ $S ; \eta \vdash true \triangleright (if(_, s_1, s_2), K)$ $S ; \eta \vdash false \triangleright (if(_, s_1, s_2), K)$

 $S ; \eta \vdash \mathsf{while}(e, s) \blacktriangleright K$

 $\begin{array}{ll} \longrightarrow & S ; \eta \vdash s_1 \blacktriangleright (s_2, K) \\ \longrightarrow & S ; \eta \vdash s \blacktriangleright K \\ \longrightarrow & S ; \eta \vdash e \triangleright (\operatorname{assign}(x, _), K) \\ \longrightarrow & S ; \eta [x \mapsto c] \vdash \operatorname{nop} \blacktriangleright K \end{array}$

$$\longrightarrow$$
 $S; \eta[x \mapsto \text{nothing}] \vdash s \triangleright K$

$$S ; \eta \vdash e \rhd (\mathsf{assert}(_), K)$$
$$S ; \eta \vdash \mathsf{nop} \blacktriangleright K$$
$$\mathsf{exception}(\mathsf{abort})$$

$$S ; \eta \vdash e \triangleright (if(_, s_1, s_2) , K)$$

$$S ; \eta \vdash s_1 \blacktriangleright K$$

$$S ; \eta \vdash s_2 \blacktriangleright K$$

 \longrightarrow

 \longrightarrow

 $\longrightarrow \hspace{1.5cm} S \ ; \eta \vdash \mathsf{if}(e, \mathsf{seq}(s, \mathsf{while}(e, s)), \mathsf{nop}) \blacktriangleright K$

Summary: Statements

$$S ; \eta \vdash f(e_1, e_2) \triangleright K$$

$$S ; \eta \vdash c_1 \triangleright (f(_, e_2), K)$$

$$S ; \eta \vdash c_2 \triangleright (f(c_1, _), K)$$

 $S ; \eta \vdash f() \vartriangleright K$

$$\begin{array}{ll} S : \eta \vdash \mathsf{return}(e) \blacktriangleright K & \longrightarrow \\ (S , \langle \eta', K' \rangle) : \eta \vdash v \triangleright (\mathsf{return}(_), K) & \longrightarrow \\ \cdot : \eta \vdash c \triangleright (\mathsf{return}(_), K) & \longrightarrow \end{array}$$

 $\begin{array}{ll} \longrightarrow & S ; \eta \vdash e_1 \triangleright (f(_, e_2) , K) \\ \longrightarrow & S ; \eta \vdash e_2 \triangleright (f(c_1, _) , K) \\ \longrightarrow & (S , \langle \eta, K \rangle) ; [x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright \cdot \\ & (given that f is defined as f(x_1, x_2) \{s\}) \\ \longrightarrow & (S , \langle \eta, K \rangle) ; \cdot \vdash s \blacktriangleright \cdot \\ & (given that f is defined as f() \{s\}) \end{array}$

$$S ; \eta \vdash e \triangleright (\mathsf{return}(_) , K)$$
$$S ; \eta' \vdash v \triangleright K'$$
$$\mathsf{value}(c)$$

Summary: Functions