15-411: Dynamic Semantics

Jan Hoffmann

Code review. Sign up today.

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
- \rightarrow Need precise ways of defining the meaning of programs

Types of (Formal) Dynamic 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

Gordon Plotkin

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

How to pick the right dynamic semantics?

Frank Pfenning

Evaluating Expressions

Continuations In an *abstract machine semantics*, which is a form of so-called *small-step operational* **semantically** separate the evaluation of an expression \mathcal{L} and \mathcal{L} and \mathcal{L} we have reached rea

Want to model a single evaluation step
Mant to model a single evaluation step a value *v*. So the basic judgment might be written *e* ! *e*⁰

$$
e\to e'
$$

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

How can we find the right place at which to make the step? Consider the expression *e*¹ + *e*2. By the left-to-right evaluation rule, we first How can we find the right place at which to make the step?

. However, the contract of the

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

which we read as "*evaluate expression e and pass the result to the continuation K*". In "Evaluate expression e and pass the result to K"

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

Evaluation Rules: Addition *e* B *K* Evaluation Duloc: Addition

Evaluation Rules: Binary Operations In the last rule we appeal to the mathematical operation of addition modulo 2³² on Γ **Dividendial Property** Sporters is

Arithmetic energtione are treated like addition. arithmetic operations are frequently addition Arithmetic operations are treated like addition are 32-bit words in two's complement representation. All other binary modular Arithmetic operations are treated like addition

 $e_1 \oplus e_2 \triangleright K$ \longrightarrow $e_1 \triangleright (_ \oplus e_2, K)$ $c_1 \triangleright (_ \oplus e_2 , K) \longrightarrow e_2 \triangleright (c_1 \oplus _ , K)$ $\mathcal{F}(\mathbf{r})$ $c_2 \triangleright (c_1 \oplus _ , \mathbf{\Lambda}) \longrightarrow c \triangleright \mathbf{\Lambda}$ ($c = c_1 \oplus c_2 \mod 2^{\circ -}$) $e_1 \oplus e_2 \triangleright K$ \longrightarrow $e_1 \triangleright (\oplus e_2 K)$ *e*¹ *e*² B *K* ! *e*¹ B (_ *e*² *, K*) arithmetic operations operations in a similar way, so we summarize the main $\sqrt{2}$ $e_1 \triangleright (I \oplus e_2, I \cap I) \longrightarrow e_2 \triangleright (e_1 \oplus I \cap I \cap I)$ *c*¹ B (_ *e*² *, K*) ! *e*² B (*c*¹ _ *, K*) $c_2 \triangleright (c_1 \oplus _ , K) \longrightarrow c \triangleright K \qquad (c = c_1 \oplus c_2 \mod 2^{32})$

state where the final outcome is reported as an arithmetic exception. We describe Aritrimetic Λ with matic is maadule 232 to matabaus $\sqrt{26}$ arabitacture $\frac{1}{2}$ state is reported as an arithmetic exception. We describe the set of $\frac{1}{2}$ Arithmetic is modulo 2³² to match our x86 architecture

e2 What about *c*¹ B (_ ↵ *e*² *, K*) ! *e*² B (*c*¹ ↵ _ *, K*) *e*¹ ↵ *e*² B *K* ! *e*¹ B (_ ↵ *e*² *, K*) effects? $\frac{1}{2}$ **c**2 B (*c*₁ $\frac{1}{2}$ $\frac{1}{2$ *c*¹ B (_ ↵ *e*² *, K*) ! *e*² B (*c*¹ ↵ _ *, K*) \mathbf{S}

Evaluation Rules: Binops with Effects $\frac{1}{2}$ **B** (*k*) $\frac{1}{2}$ **c** $\frac{1$ **aluation Rules: Binops with Effects** *e*¹ *e*² B *K* ! *e*¹ B (_ *e*² *, K*) aluation Rules: Binops with E1 *^c*² ^B (*c*¹ _ *, K*) ! *^c* ^B *^K* (*^c* ⁼ *^c*¹ *^c*² mod 232) ϵ ¹ Rides Binons with Fi *c*¹ B (_ *e*² *, K*) ! *e*² B (*c*¹ _ *, K*)

In case of an arithmetic exception: Abort the computation and report and error In case of an arithmetic exception: Abort the computation and report and error state where the final outcome is reported as an arithmetic exception. We describe $\frac{1}{2}$ and a fragmential operation $\frac{1}{2}$ of $\frac{1}{2}$ raise $\frac{1}{2}$ raise $\frac{1}{2}$ m base or an antimitent exception. Note the computation and $\frac{1}{2}$ report and choi In case of an arithmetic exception: Abort the computation and

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

\n
$$
c_1 \triangleright (- \oslash e_2, K) \longrightarrow e_2 \triangleright (c_1 \oslash ... K)
$$

\n
$$
c_2 \triangleright (c_1 \oslash ... K) \longrightarrow c \triangleright K \qquad (c = c_1 \oslash c_2)
$$

\n
$$
c_2 \triangleright (c_1 \oslash ... K) \longrightarrow \text{exception}(\text{arith}) (c_1 \oslash c_2 \text{ undefined})
$$

For an effectful operation such as division, the last step could also raise an arith-

*^c*² ^B (*c*¹ _ *, K*) ! *^c* ^B *^K* (*^c* ⁼ *^c*¹ *^c*² mod 232)

Thare is no rule for further avaluating an avecation conditions here. I here is no rule for further evaluating an exception. There is no rule for further evaluating an exception. division and modulus, and the conditions under which the result is mathematiracions no function function cyaluding an exception. There is no rule for further evaluating an exception.

Example Evaluation **Example** Consider the expression ((4 + 5) ⇤ 10) + 2. Using our evaluation rules,

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

Evaluation Rules: End of and Evaluation have specified this in previous lectures and assignments, so we won't detail the specified theory is the specifi
In previous lectures and assignments, so we won't detail the specified the specified the specified theory is

continuation we stop the abstract machine and return value(*c*) If we reach a constant and the empty continuation then we stop

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

What happens when evaluation finishes normally? In the case of the empty state of the empty state of the empty
In the empty state of the empty st

Evaluation Rules: Boolean Expressions Evaluation Duloo: Pooloon Evaropoiono continuation we stop the abstract machine and return value(*c*) **Evalua** What happens when evaluation finishes normally? In the case of the empty significant case of the empty cally "undefined" (like division by zero) and therefore must raise an exception. We Evaluation Rules. Boolean Expressions

$$
e_1 \& e_2 \triangleright K \longrightarrow e_1 \triangleright (_ \& e_2 \ , K)
$$
\n
$$
\text{false} \triangleright (_ \& e_2 \ , K) \longrightarrow \text{false} \triangleright K \longrightarrow \text{Notice the short-}
$$
\n
$$
\text{true} \triangleright (_ \& e_2 \ , K) \longrightarrow e_2 \triangleright K
$$

true and *false* are also values that and factor and books and be confused at the confused at α which encodes the short-circuiting behavior of the short-circuiting behavior of the conjunction. Also note tha
Also note that we have a short-circuit in the conjunction. Also note that we have a short-circuit in the conju true and false are also values false and true and true and true as well as well as well as well as well as integer values of α

conditions here.

(We could also use 1 and 0 but distinguishing helps detect errors.) ture that we expect integers and boolean expect integers and boolean expect integers and boolean expressions to

Variables and Environments **Dynamic Semantics L13.4**

How do we evaluate variable?

3 Variables
3 Variables – Variables III (1990)
3 Variables – Variables II (1990)

$$
x \triangleright K \quad \longrightarrow \quad ?
$$

 $x \triangleright n \quad \longrightarrow \quad .$ booleans. We can continue along the continue along the studies of variables. When we do the studies of variables or $x \triangleright K \rightarrow ?$ booleans.

Need to be your on environment that maps variables to values values to the Need to have an environment that maps variables to values : that maps varia

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

The machine state consists now of an expression, a continuation, and an **v** (*i*) is already defined the state of the state of the abstract matrix of the ab The machine state consists now of an expression, a continuation, and an environment

 $\eta \vdash e \rhd K$ $\eta \vdash e \triangleright K$

Variables and Environments II *v* (if ⌘(*x*) is already defined). The state of the abstract machine now contains the variables and Environments II and ⌘[*x* 7! *v*] for either adding *x* 7! *v* to ⌘ or overwriting the current value of *x* by *v ariables and Environments II* environment are separate by a turnstile (the expression to evaluate and the expression to evaluate and the eva
The evaluate and the evaluate and the expression to evaluate and the evaluate and the evaluate and the evaluat *v* and *is already defined the state of the state of the state of the abstract matrix the state of the state* Variables and Environments II α (if α) is already defined of the state of the abstract machine now contains the abstract machine now contains the abstract machine α

The rules we have seen so far just carry over The rules we have soon so far just carry over The rules we have seen so far just carry o κ carry σ The r

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

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

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

environment are separate by a turnstile (\sim) from the expression to evaluate and the expression to evaluate a
The expression to evaluate and the expression to evaluate and the expression to evaluate and the expression to

⌘ ` *x* B *K* ! ⌘ ` ⌘(*x*) B *K* X^2 \longrightarrow X Variables are just looked up in the environment. Variables are simply looked up

its continuation.

$$
\eta \vdash x \rhd K \quad \longrightarrow \quad \eta \vdash \eta(x) \rhd K
$$

4 Executive Statement Concernsive Statements The environment pever ebengee when evaluating everessione variables must be initialized before they are used). The environment never changes when evaluating expressions

Executing Statements

Executing Statements I **4 Executing Statements Dynamic Semantics L13.5 4 Executing Statements** Executing statements in L3, the fragment of C0 we have considered so far, can

Executions of statements don't pass values to the continuation or raise an exception. The "normal" execution of a statement does not pass a value Executions of statements don't pass values to the continuation either complete normally, return from the current function with a return statement, Executions or statements don't pass values to the continuation **Executions of statements don**

Statements have usually an effect on the environment Statements have usually an effect on the environment Statements have usually an effect on the environment

Executing statements in L3, the fragment of C0 we have considered so far, can

Executing Statements II ⌘ ` nop I (*s,K*) ! ⌘ ` *s* I *K* Γ_{λ} (e.g., Γ_{λ}) so eventually becomes a normalized becomes a normalized becomes a normalized becomes a nonpolarized bec ⌘ ` seq(*s*1*, s*2) I *K* ! ⌘ ` *s*¹ I (*s*² *, K*)

Interaction with expressions is straightforward h expressions is straightforward

Assignments: first statement from the continuation. When executing an assignment we first have to evaluate the statement from the environment. When executing an assignment we first have the environment we first have the environment of the environment we first have the continuation. When executing an assignment we fi The last line codifies that if there no further statement to execution, we grab the

$$
\eta \vdash \operatorname{assign}(x, e) \blacktriangleright K \longrightarrow \eta \vdash e \rhd (\operatorname{assign}(x, _), K)
$$
\n
$$
\eta \vdash v \rhd (\operatorname{assign}(x, _), K) \longrightarrow
$$
\nUpdate the environment with new mapping.

Executing Statements III first statement from the continuation. When executing an assignment we first have first h to evaluate the assignment of the value in the value in the environment. The value is the environment of the e **The last is that if the line codifierer statements iii** the statement of the statement of the statement of the s first statement from the continuation. When executing an assignment we first have first have first have first
The continuation of the continuation and assignment we first have first have first have first have first have ng Statements III

Conditionals: Conditionals follow the pattern of the short-circuiting conjunction. ${\sf a}$ assign(${\sf a}$) ${\sf a}$ it is easily ${\sf a}$ as ${\sf a}$ ${\sf b}$ ${\sf c}$ ${\$ ⌘ ` *v* B (assign(*x,* _) *, K*) ! ⌘[*x* 7! *v*] ` nop I *K* Conditionalet

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

The last line codifies that if the last line codifies that if there no further statement to execution, we grab
The last line codifies the last line codifies the last line codifies the last line codifies the last line codi

Executing Statements IV while(*e, s*) ⌘ if(*e,*seq(*s,*while(*e, s*))*,* nop) E *v* Ω uting Ctatamenta II / ⌘ ` if(*e, s*1*, s*2) I *K* ! ⌘ ` *e* B (if(_*, s*1*, s*2) *, K*) Conditionals follow the pattern of the short-circuiting conjunction. EXUUUIII IY ULULUITIUITIU TV

Loops: directly. S : s₁, s₂) I *k* $\frac{1}{2}$ *e* B (if($\frac{1}{2}$ $\frac{1}{2}$ ⌘ ` true B (if(_*, s*1*, s*2)) ! ⌘ ` *s*¹ I *K* \blacksquare **false B** \blacksquare $\$

Loops:

$$
\eta \vdash \text{while}(e, s) \blacktriangleright K \longrightarrow ?
$$

Vot that the following statements are equivalent: What the following statements are equivalent: Not that the following statements are equivalent: Not that the following statements are eq

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

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

to avoid writing out several rules implementing the right-hand side of this identity α

⌘ ` true B (assert(_)*, K*) ! ⌘ ` nop I *K* $\mathsf{ion:}\quad\qquad \mathsf{non}\quad\qquad \mathsf{non}\quad\q$ which is the strain machine transition of non-termination, which is modeled naturally: we have a strain of non-termination, which is modeled naturally: we have a strain of non-termination, we have a strain of non-terminati ⌘ ` while(*e, s*) I *K* ! ⌘ ` if(*e,*seq(*s,*while(*e, s*))*,* nop) I *K*

$$
s_0 \hspace{0.2cm} \longrightarrow \hspace{0.2cm} s_1 \hspace{0.2cm} \longrightarrow \hspace{0.2cm} s_2 \hspace{0.2cm} \longrightarrow \hspace{0.2cm} \cdots
$$

We can make an infinite number of stens without reaching a final state We can make an infinite number of steps without reaching a final state just have abstract machine transitions *s*⁰ ! *s*¹ ! *s*² ! *···* without we can make an infinite number of steps without reaching a final we can make an immite namiocron steps without reaching a final We can make an infinite number of steps without reaching a final state

Executing Statements V
 Executing Statements V ever arriving at a final state. The final states are just nop I *·* and exception(*E*), Evoouting Statemante V just have abstract machine transitions *s*⁰ ! *s*¹ ! *s*² ! *···* without ⌘ ` while(*e, s*) I *K* ! ⌘ ` if(*e,*seq(*s,*while(*e, s*))*,* nop) I *K* Loops bring up the question of non-termination, which is modeled naturally: we **Executing Statements V**

While loops are a bit more complicated. We take a slight shortcut by using the slight shortcut by using t

where E can currently be either arithmetic by a failing by a fai

Assertions: Assertions: **E** just have abstract machine transitions *s*⁰ ! *s*¹ ! *s*² ! *···* without ever arriving at a final state α final state α *and exception i* α , and exception α \overline{A} **ASSEITIONS:**

Conditionals follow the pattern of the short-circuiting conjunction.

 \mathcal{L} are a bit more complicated. We take a slight shortcut by using the slight shortcut by using the

Conditionals follow the pattern of the short-circuiting conjunction.

 $\eta \vdash \mathsf{assert}(e) \blacktriangleright K \longrightarrow \eta \vdash e \triangleright (\mathsf{assert}(_) , K)$ ⌘ ` true B (assert(_)*, K*) ! ⌘ ` nop I *K* $\eta \vdash \textsf{true} \rhd (\textsf{assert}(_), K) \qquad \longrightarrow \qquad \eta \vdash \textsf{nop} \blacktriangleright K$ $\frac{1}{2}$ ⌘ ` assert(*e*) I *K* ! ⌘ `*e* B (assert(_)*, K*) $E = \frac{E}{2\pi} E$ and E and E are E and E and E failing E and E \rightarrow E $\$ assert statement. \blacksquare if(, s) \blacksquare if(, s) \blacksquare \bl $\eta \vdash$ assert $(e) \blacktriangleright K$ ⌘ ` false B (if(_*, s*1*, s*2)) ! ⌘ ` *s*² I *K*

where E can currently be either arithmetic by a failure arithmetic by a failure \mathbb{R}^n

ever arriving at a final state. The final states are just nop I *·* and exception(*E*),

 $\eta \vdash \mathsf{false} \rhd (\mathsf{assert}(_), K) \qquad \longrightarrow \qquad \mathsf{exception}(\mathsf{abort})$ ⌘ ` false B (assert(_)*, K*) ! exception(abort) ⌘ ` true B (assert(_)*, K*) ! ⌘ ` nop I *K* $\eta \vdash \mathsf{false} \rhd (\mathsf{assert}(_), K) \qquad \longrightarrow \qquad \mathsf{exception}(\mathsf{abort})$ $y' = \tan \theta \cos \theta \cos \theta (-y) + y'$

Declarations: to avoid writing out several rules in place of the right-hand side of the right-hand side of the right-hand side of this identity in the right-hand side of the right-hand side of the right-hand side of this identity in the to avoid writing out the right-hand side of the right-hand side of the right-hand side of the right-hand side o Declarations:

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

Final states: Loops bring up the question of non-termination, which is modeled naturally: we just have abstract machine transitions *s*⁰ ! *s*¹ ! *s*² ! *···* without

 $exception(E)$ *inal state are just nop* **b** *·* and e *care* $\mathsf{excepuon}(E)$ nop \blacktriangleright . We can statically static static static statically statically

Loops bring up the question of non-termination of non-termination, which is modeled naturally: which is modele f in a states.
Then we will be abstract many states s^2 is a sense of the sense of s^2 is a sense $\sqrt{16}$ and other deviations in $\frac{16}{16}$ current value of **x** and restore it and restoruting. The structure in the structure in the structure of $\frac{1}{2}$ of a little would have to the value of the value of the rightmost of the value of the rightmost of the If C0 had shadowing then we would have to be careful here.

Example: Infinite Loop LACITIVIG. II IIII IILG L Γ Latin are tied to function calls. The model of $A_n = \frac{1}{n}$ that are the time of the next section calls. We discuss the next section calls. We discuss the next section \sim Γ Latin are that all the function calls. The next section calls to the next section calls. The next section of $A_n = \frac{1}{n}$ Train are the time of the next section calls. The next section calls the next section of the next section.

Using rules for statement execution, we obtion the following execution; where *s* ⌘

Example Consider the statement while(*x >* 0*,* assign(*x, x* + 1)) and ⌘ = [*x*7!1]. while $(x > 0$, $\text{assign}(x, x + 1))$ $\eta = [x \mapsto 1]$ $\text{s} = \text{assign}(x, x + 1)$

Using rules for statement execution, we obtion the following execution; where *s* ⌘

Using rules for statement execution, we obtion the following execution; where *s* ⌘

Using rules for statement execution, we obtion the following execution; where *s* ⌘

Functions

Function Calls

What needs to happen at a 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
- 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

Call Stack **Call Stack** we invoke the function, whose body starts to execute in an environment that \sim Call Stack to evaluate the function \int we invoke the function, whose body starts to execute in an environment that maps to execute in an environment
In an environment that maps to execute in an environment that maps to execute in an environment that maps that A function call first has to evaluate the function arguments, from left to right. Then

We need to keep track of continuations and environment in stack frames We need to keen track of continuations and environment in stack frames. have to save to save the current environment of the current environment of the caller somewhere. Similarly, we
The caller somewhere the caller somewhere the caller somewhere. Similarly, we also also also also be also also we invoke the function, whose body starts to execute in an environment that maps to execute in an environment that m we fleed to keep track of continuations and environment in stack frames.

Call stack:	S	:= $\cdot S, \langle \eta, K \rangle$
Environment	Environment	

have to save the current environment of the current environment of the caller somewhere. Similarly, we also ca

Configurations: $Confinurations$ Now states representing evaluation of expression and execution of statements have

S EXISUITY TURS CAN DE INTED TO THE HEW CONFIGURATIONS **R** in a series through the call stack **S** $\frac{1}{2}$ $\frac{1}{2}$ Existing rules can Existing rules can be lifted to the new configurations by passing through the call stack

Rules for Returns D_{11} D_{22} for D_{21} F_{12} as for Returns a return statement we simply have to restore the called \sim ronment and continuation from the stack and pass the return value to the caller's

Need to restore continuation and environment and pass return value eed to restore continuation and environment and pass return value coal to rester

 \mathcal{M}_{max} and the caller statement we simply have to restore the caller to restore the caller the caller

and a control only be reached by a control return to the end of the end of end of end of every void function. As an end of end of the end of end of end of example, and the end of end of end of every void function. As an en Special case: returning void functions without return. S ; η \vdash return(*e*) \blacktriangleright *K* S , $\langle \eta', K'\rangle$; $\eta \vdash v \rhd (\mathsf{return}(_) \, , K)$ and elaborate "return;" as return;" as return;" as return;" as returning;" as return;" as return;" as return;
" as returns;" as r \overline{a} **Dummy value** S ; $\eta \vdash$ return $(e) \blacktriangleright K$ *S* ; ⌘ ` return(*e*) I *K* ! *S* ; ⌘ ` *e* B (return(_) *, K*) In order to support functions returning void, we can use a useless value, nothing, and electronical world. Will only be reached by a representation as returning world. approvide returning void transformations without return. $S: \langle n', K' \rangle : n \vdash \text{non} \blacktriangleright \cdot \longrightarrow S : n' \vdash \text{nothing}$ Special case: returning void In order to support functions returning void, we can use a useless value, nothing, $M\llap{/}$ illevely ha rasched by S , $\langle \eta', K' \rangle$; $\eta \vdash$ nop \blacktriangleright *·* \longrightarrow *S*; $\eta' \vdash$ nothing $\triangleright K'$

i ; ⌘ ` nop I *·* ! *S* ; ⌘⁰ ` nothing B *K*⁰ Alternative: elaborate each function that returns void with return(nothing) statements.

Execution of the Main Function Execution of the Main Function **6 Statically individual dynamics, and Safety Safety and Safety Safety Safety**

How can we execute a program? \mathcal{L}_{max} the mathematic initially initially initially initially initially initially initially initially in How can we execute a program! \mathbb{R}^n start the machine initially initially initially initially initially initially initially initially initially in

 \cdot ; $\cdot \vdash \textsf{main}() \triangleright \cdot$ (initial state) \mathcal{C} , \mathcal{C} , which will eventually \mathcal{C} , which \mathcal{C} is returned by the main function. We have \mathcal{C} $\frac{1}{\sqrt{2\pi}}$ *·* ; *·* ` main() B *·* (initial state) \cdot ; $\eta \vdash c \rhd \cdot \quad \longrightarrow \quad \text{value}(c) \qquad \text{(final state)}$ *·* ; *·* ` main() B *·* (initial state) \cdot ; $\eta \vdash c \rhd \cdot \quad \longrightarrow \quad \text{value}(c) \qquad \text{(final s)}$ *·* ; *·* ` main() B *·* (initial state) \cdot ; $\eta \vdash c \rhd \cdot \quad \longrightarrow \quad$ value (c) (final state)

Statics, Dynamics, and Safety

Overview of Machine States (Configurations) *·* ; ⌘ ` *c* B *·* ! value(*c*) (final state) which will eventually step to value(*c*), where *c* is returned by the main function. We

 S ; η \vdash *s* \blacktriangleright *K* – Evaluating the statement *s* with the continuation *K*

 $value(c)$ – Final state, return a value

have defined four kinds of *machine state ST* :

 $\mathsf{exception}(E)$ – 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 In the other direction, the other direction, there are many examples of non-final states that have $n=1$ per state

Progress don't go wrong!

Sition. On that any rule of the value of the t thing about whether those rules are reasonable or notation $\mathcal{L}_\mathbf{z}$ obviously we do be on the left-hand side of a trans-hand side of a trans-hand side of a trans-hand side of a tra thing about whether those rules are reasonable or not? Well-typed programs don't go wrong!

unreasonable rule. We might also care that the language is deterministic: that the language is deterministic:
That every that the language is deterministic: that every the language is deterministic: that every the langua

There are many non-final states that don't have transitions, e.g. In the other direction, there may be are many examples of an intervention of the states of the state of t There are many non-final states that don't have tra Increase are many non-man otates that don't nave transitions, org

unreasonable rule. We might also care that the language is deterministic: that the language is deterministic:
That every care that the language is deterministic: that every care that every care that every care that every

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

Central relationship between static and dynamic semantics:

Should be free of underlined behavior. 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*

 \cdot ; \cdot \vdash main() \longrightarrow ST_1 \longrightarrow $\cdot \cdot \cdot$ \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 $ST_n \longrightarrow ST'$.

In a course like 15-312, we would learn how to prove the sorts of the sorts of the sorts of the original α How to prove this? 15-312

Summary I

- $S; \eta \vdash c_2 \rhd (c_1 \odot \ldots K)$ \longrightarrow $S; \eta \vdash c \rhd K$ $(c = c_1 \odot c_2)$
 $S; \eta \vdash c_2 \rhd (c_1 \odot \ldots K)$ \longrightarrow exception(arith) $(c_1 \odot c_2 \text{ un})$
- S ; $\eta \vdash e_1$ && $e_2 \rhd K$ \longrightarrow S ; η \vdash false \triangleright (\bot && e_2 , K) S ; η \vdash true \triangleright (\bot && e_2 , K)

- $S; \eta \vdash e_1 \odot e_2 \rhd K$
 $S; \eta \vdash e_1 \rhd (_ \odot e_2, K)$
 $S; \eta \vdash e_2 \rhd (c_1 \odot _ K)$ $S; \eta \vdash c_1 \rhd (_ \circ e_2, K) \longrightarrow S; \eta \vdash e_2 \rhd (c_1 \circ _ , K)$
 $S; \eta \vdash c_2 \rhd (c_1 \circ _ , K) \longrightarrow S; \eta \vdash c \rhd K \qquad (c = c)$
	- \longrightarrow exception(arith) $(c_1 \odot c_2 \text{ undefined})$

$$
S; \eta \vdash e_1 \rhd (_ \&\& e_2 \; , K)
$$

$$
\rightarrow \qquad S \mathrel{;} \eta \vdash \mathsf{false} \rhd K
$$

$$
\rightarrow \qquad S \mathrel{;} \eta \vdash e_2 \rhd K
$$

 S ; $\eta \vdash x \triangleright K$ \longrightarrow S ; $\eta \vdash \eta(x) \triangleright K$

Summary: Expressions *S* ; ⌘ ` true B (assert(_) *, K*) ! *S* ; ⌘ ` nop I *K*

-
- S ; $\eta \vdash \mathsf{nop} \blacktriangleright (s, K)$
 S ; $\eta \vdash \mathsf{assign}(x, e) \blacktriangleright K$
-
-

 S ; $\eta \vdash$ decl $(x, \tau, s) \blacktriangleright K$ \longrightarrow

- S ; η \vdash assert $(e) \blacktriangleright K$ S ; $\eta \vdash \textsf{true} \rhd (\textsf{assert}(_), K) \longrightarrow$
 $S : \eta \vdash \textsf{false} \rhd (\textsf{assert}(_), K) \longrightarrow$ S ; η \vdash false \triangleright (assert(_), K) \longrightarrow exception(abort)
- S ; η \vdash true \triangleright (if(_, s₁, s₂), K) S ; η \vdash s₁ \blacktriangleright K
 S ; η \vdash false \triangleright (if(_, s₁, s₂), K) S ; η \vdash s₂ \blacktriangleright K S ; η \vdash false \triangleright (if(_*, s*₁*, s*₂*), K*)

 $S; \eta \vdash \mathsf{seq}(s_1, s_2) \blacktriangleright K$ \longrightarrow $S; \eta \vdash s_1 \blacktriangleright (s_2, K)$
 $S; \eta \vdash \mathsf{nop} \blacktriangleright (s, K)$ \longrightarrow $S; \eta \vdash s \blacktriangleright K$ $S; \eta \vdash \mathsf{assign}(x, e) \blacktriangleright K$ $S; \eta \vdash e \rhd (\mathsf{assign}(x, _), K)$
 $S; \eta \vdash c \rhd (\mathsf{assign}(x, _), K)$ \longrightarrow $S; \eta[x \mapsto c] \vdash \mathsf{nop} \blacktriangleright K$ \longrightarrow S ; $\eta[x \mapsto c]$ \vdash nop \blacktriangleright *K*

$$
S: \eta[x \mapsto \text{nothing}] \vdash s \blacktriangleright K
$$

$$
S; \eta \vdash e \rhd (\text{assert}(_), K)
$$

$$
S; \eta \vdash \text{nop} \blacktriangleright K
$$

excentration(short)

 $S: \eta \vdash \text{if}(e, s_1, s_2) \blacktriangleright K$
 $S: \eta \vdash t$ rue \triangleright (if(_, s₁, s₂), K)
 \longrightarrow $S: \eta \vdash s_1 \blacktriangleright K$

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

Summary: Statements S , *c*₁ $\frac{1}{2}$, *c*₁

$$
S; \eta \vdash f(e_1, e_2) \rhd K S; \eta \vdash c_1 \rhd (f(., e_2), K) S; \eta \vdash c_2 \rhd (f(c_1, .), K)
$$

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

$$
S; \eta \vdash \text{return}(e) \blacktriangleright K \longrightarrow S; \eta \vdash e \triangleright (\text{return}(_) , K) (S, \langle \eta', K' \rangle) ; \eta \vdash v \triangleright (\text{return}(_) , K) \longrightarrow S; \eta' \vdash v \triangleright K' \cdot; \eta \vdash c \triangleright (\text{return}(_) , K) \longrightarrow \text{value}(c)
$$

 \longrightarrow *S* ; $\eta \vdash e_1 \triangleright (f(_, e_2) , K)$ \longrightarrow *S*; $\eta \vdash e_2 \rhd (f(c_1, _) , K)$ \longrightarrow $(S, \langle \eta, K \rangle)$; $[x_1 \mapsto c_1, x_2 \mapsto c_2] \vdash s \blacktriangleright$ *(given that f is defined as* $f(x_1, x_2)$ $\{s\}$ *) (given that f is defined as* $f(\left\{s\right\})$

$$
S; \eta \vdash e \rhd (\text{return}(_), K)
$$

$$
S; \eta' \vdash v \rhd K'
$$

value(c)

Summary: Functions