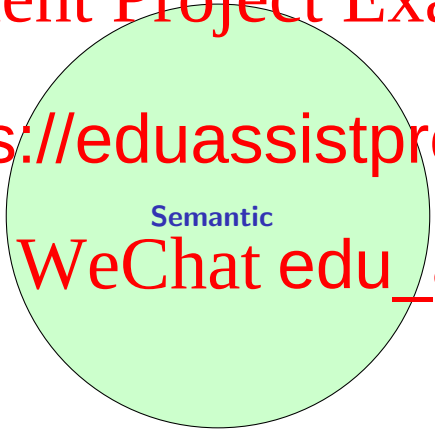
# Semantic

Dr. Liam O'Connor
University of Edinburgh LFCS
UNSW, Term 3 2020

## Semantics

σημαντιχως

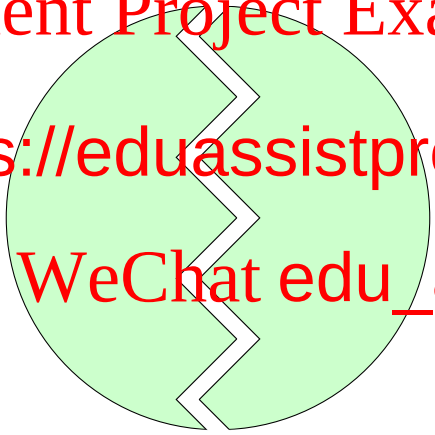




## Semantics

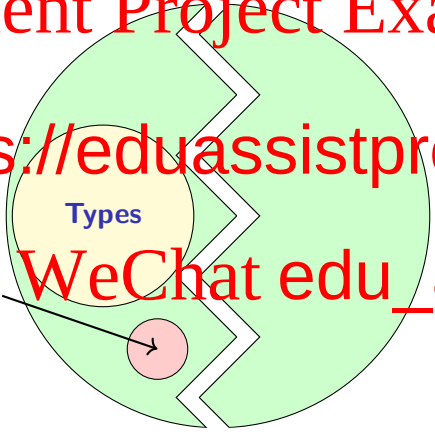
$\sigma\eta\mu\alpha\nu\tau\iota\chi\omega\varsigma$





# Semantics

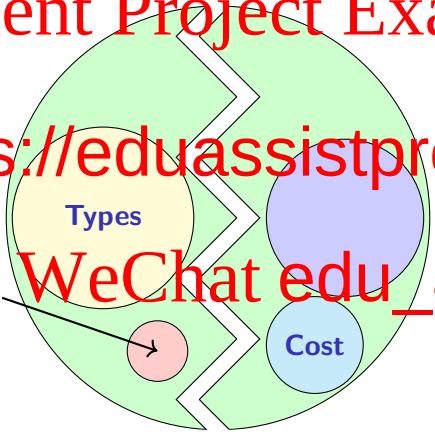
$\sigma\eta\mu\alpha\nu\tau\iota\chi\omega\varsigma$



# Semantics

$\sigma\eta\mu\alpha\nu\tau\iota\chi\omega\varsigma$





## Static Semantics



**Definition**

The *static sema* *P* that can be determine program.



Recall our arithmetic expression language. What propertie about those terms?

## Static Semantics



**Definition**

The *static sema* *P* that can be determine program.



Recall our arithmetic expression language. What propertie about those terms?
The only thing we can check is that the program is well-scoped (as

## Scope-Checking



$$\frac{}{x \ \mathbf{ok}} \qquad \frac{e_1 \ \mathbf{ok} \qquad e_2 \ \mathbf{ok}}{\lambda e_1 \, e_2 \ \mathbf{ok}} \qquad \frac{e_1 \ \mathbf{ok} \qquad e_2 \ \mathbf{ok}}{e_1 \, e_2 \ \mathbf{ok}}$$



**Key Idea**



We keep a *context* $\Gamma$, a set of assumptions, on the left hand of $e$ **ok**, indicating what is required in order for $e$ to be *well-scoped*.

This could be read as hypothetical derivations for the judgement $e$ **ok** or as a binary judgement $\Gamma \vdash e$ **ok**; whichever you prefer.

## Scope-Checking



$$\frac{}{\Gamma \vdash ()} \qquad \frac{\Gamma \vdash e_1 \ \mathbf{ok} \qquad \Gamma \vdash e_2 \ \mathbf{ok}}{\Gamma \vdash (e_1, e_2) \ \mathbf{ok}}$$



**Key Idea**



We keep a *context* $\Gamma$, a set of assumptions, on the left hand of $e$ **ok**, indicating what is required in order for $e$ to be *well-scoped*.

This could be read as hypothetical derivations for the judgement $e$ **ok** or as a binary judgement $\Gamma \vdash e$ **ok**; whichever you prefer.

## Scope-Checking



$$\frac{}{\Gamma \vdash (} \qquad \frac{\Gamma \vdash e_1 \textbf{ ok} \qquad \Gamma \vdash e_2 \textbf{ ok}}{\Gamma \vdash e_1 \; e_2 \textbf{ ok}} \qquad \frac{\Gamma \vdash e_1 \textbf{ ok} \qquad \Gamma \vdash e_2 \textbf{ ok}}{e_2) \textbf{ ok}}$$



**Key Idea**



We keep a *context* $\Gamma$, a set of assumptions, on the left hand of $e$ **ok**, indicating what is required in order for $e$ to be *well-scoped*.

This could be read as hypothetical derivations for the judgement $e$ **ok** or as a binary judgement $\Gamma \vdash e$ **ok**; whichever you prefer.

## Scope-Checking



$$\frac{}{\Gamma \vdash (} \qquad \frac{\Gamma \vdash e_1 \textbf{ ok} \qquad \Gamma \vdash e_2 \textbf{ ok}}{\Gamma \vdash e_1 \textbf{ ok}} \qquad \frac{\Gamma \vdash e_1 \textbf{ ok} \qquad \Gamma \vdash e_2 \textbf{ ok}}{e_2) \textbf{ ok}}$$



**Key Idea**



We keep a *context* $\Gamma$, a set of assumptions, on the left hand of $e$ **ok**, indicating what is required in order for $e$ to be *well-scoped*.

This could be read as hypothetical derivations for the judgement $e$ **ok** or as a binary judgement $\Gamma \vdash e$ **ok**; whichever you prefer.

## Scope-Checking



$$\frac{}{\Gamma \vdash (} \qquad \frac{\Gamma \vdash e_1 \textbf{ ok} \qquad \Gamma \vdash e_2 \textbf{ ok}}{\Gamma \vdash e_1 \; e_2 \textbf{ ok}} \qquad \frac{\Gamma \vdash e_1 \textbf{ ok} \qquad \Gamma \vdash e_2 \textbf{ ok}}{e_2) \textbf{ ok}}$$



**Key Idea**



We keep a *context* $\Gamma$, a set of assumptions, on the left hand of $e$ **ok**, indicating what is required in order for $e$ to be *well-scoped*.

This could be read as hypothetical derivations for the judgement $e$ **ok** or as a binary judgement $\Gamma \vdash e$ **ok**; whichever you prefer.

## Scope-Checking



$$\frac{}{\Gamma \vdash ()\ \mathbf{ok}} \qquad \frac{\Gamma \vdash e_1\ \mathbf{ok} \qquad \Gamma \vdash e_2\ \mathbf{ok}}{\Gamma \vdash (e_1, e_2)\ \mathbf{ok}} \qquad \frac{\Gamma \vdash e_1\ \mathbf{ok} \qquad \Gamma \vdash e_2\ \mathbf{ok}}{\Gamma \vdash \mathbf{k}\ e_1\ e_2\ \mathbf{ok}}$$



**Key Idea**



We keep a *context* $\Gamma$, a set of assumptions, on the left hand of $e\ \mathbf{ok}$, indicating what is required in order for $e$ to be *well-scoped*.

This could be read as hypothetical derivations for the judgement $e\ \mathbf{ok}$ or as a binary judgement $\Gamma \vdash e\ \mathbf{ok}$; whichever you prefer.

## Scope-Checking Example



$$\frac{}{\vdash \texttt{(Let "x" (N 3) (Let "y" (N 4) (Plus (V "x") (V "y"))))}}$$

## Scope-Checking Example



$$\frac{\dfrac{}{\vdash (\texttt{N 3})}}{\vdash (\texttt{Let "x" (N 3) (Let "y" (N 4) (Plus (V "x") (V "y"))))}}$$

## Scope-Checking Example



$$\frac{}{\Gamma \vdash (} \qquad \frac{\Gamma \vdash e_1\ \textbf{ok} \qquad \Gamma \vdash e_2\ \textbf{ok}}{\Gamma \vdash e_1\ e_2)\ \textbf{ok}} \qquad \frac{\Gamma \vdash e_1\ \textbf{ok} \qquad \Gamma \vdash e_2\ \textbf{ok}}{\Gamma \vdash e_1\ e_2\ \textbf{ok}}$$





$$\frac{\dfrac{}{\vdash (\texttt{N}\ 3)} \qquad \dfrac{}{\texttt{"x"} \vdash (\texttt{Let "y" (N 4) (Plus (V "x") (V "y"))})}}{\vdash (\texttt{Let "x" (N 3) (Let "y" (N 4) (Plus (V "x") (V "y")))})}$$

## Scope-Checking Example



$$\dfrac{\dfrac{}{\Gamma \vdash (} \quad \dfrac{\Gamma \vdash e_1 \textbf{ ok} \qquad \Gamma \vdash e_2 \textbf{ ok}}{\Gamma \vdash (} \quad \dfrac{\Gamma \vdash e_1 \textbf{ ok} \qquad \Gamma \vdash e_2 \textbf{ ok}}{\Gamma \vdash e_2) \textbf{ ok}}}{\Gamma \vdash \textbf{ok}}$$

$$\dfrac{\dfrac{}{\vdash \texttt{(N 3)}} \quad \dfrac{\dfrac{}{\texttt{"x"} \vdash \texttt{(N 4)}} \qquad \dfrac{}{\texttt{"y"} , \texttt{"x"}}}{\texttt{"x"} \vdash \texttt{(Let "y" (N 4) (Plus (V "x") (V "y")))}}}{\vdash \texttt{(Let "x" (N 3) (Let "y" (N 4) (Plus (V "x") (V "y"))))}}$$

## Scope-Checking Example



$$\frac{\dfrac{}{\Gamma \vdash (\texttt{N}\ 3)\ \textbf{ok}} \qquad \dfrac{\dfrac{}{\Gamma \vdash (\texttt{N}\ 4)\ \textbf{ok}} \qquad \dfrac{\dfrac{}{\Gamma, \texttt{"y"}, \texttt{"x"} \vdash (\texttt{V}\ \texttt{"x"})\ \textbf{ok}} \qquad \dfrac{}{\Gamma, \texttt{"y"}, \texttt{"x"} \vdash (\texttt{V}\ \texttt{"y"})\ \textbf{ok}}}{\Gamma, \texttt{"y"}, \texttt{"x"} \vdash (\texttt{Plus}\ (\texttt{V}\ \texttt{"x"})\ (\texttt{V}\ \texttt{"y"}))\ \textbf{ok}}}{\Gamma, \texttt{"x"} \vdash (\texttt{Let}\ \texttt{"y"}\ (\texttt{N}\ 4)\ (\texttt{Plus}\ (\texttt{V}\ \texttt{"x"})\ (\texttt{V}\ \texttt{"y"})))\ \textbf{ok}}}{\Gamma \vdash (\texttt{Let}\ \texttt{"x"}\ (\texttt{N}\ 3)\ (\texttt{Let}\ \texttt{"y"}\ (\texttt{N}\ 4)\ (\texttt{Plus}\ (\texttt{V}\ \texttt{"x"})\ (\texttt{V}\ \texttt{"y"}))))\ \textbf{ok}}$$

# Dynamic Semantics

Dynamic Semantics can be specified in many ways:

1. *Denotational Semantics* is the *compositional* construction of a *mathematical object* for each form of *syntax*. COMP6752 (briefly)

# Dynamic Semantics

Dynamic Semantics can be specified in many ways:

1. *Denotational Semantics* is the *compositional* construction of a *mathematical object* for each form of *syntax*. COMP6752 (briefly)
2. *Axiomatic* ow correctness of a program to

# Dynamic Semantics

Dynamic Semantics can be specified in many ways:



1. *Denotational Semantics* is the *compositional* construction of a *mathematical object* for each form of *syntax*. COMP6752 (briefly)
2. *Axiomatic* ow correctness of a program to
3. *Operation* *state machine* or *transition system*.

# Dynamic Semantics



Dynamic Semantics can be specified in many ways:

1. *Denotational Semantics* is the *compositional* construction of a *mathematical object* for each form of *syntax*. COMP6752 (briefly)
2. *Axiomatic* ow correctness of a program to
3. *Operation* *state machine* or *transition system*.





**In this course**

We focus mostly on operational semantics. We will use axiomatic semantics (Hoare Logic) on Thursday in the imperative programming topic. Denotational semantics are mostly an extension topic, except for the very next slide.

## Denotational Semantics



$$
\begin{aligned}
[\![\texttt{Var}\ x]\!] &= \\
[\![\texttt{Plus}\ e_1\ e_2]\!] & \\
[\![\texttt{Times}\ e_1\ e_2]\!] & \\
[\![\texttt{Let}\ x\ e_1\ e_2]\!] &
\end{aligned}
$$

## Denotational Semantics



$$
\begin{aligned}
[\![\texttt{Var}\ x]\!] &= \\
[\![\texttt{Plus}\ e_1\ e_2]\!] &= \\
[\![\texttt{Times}\ e_1\ e_2]\!] &= \\
[\![\texttt{Let}\ x\ e_1\ e_2]\!] &= \\
\end{aligned}
$$

## Denotational Semantics



$$[\![\cdot]\!] : \textbf{AST} \to (\textbf{Var} \rightharpoonup \mathbb{Z}) \to \mathbb{Z}$$

Our denotation fo ents (mapping from variables to v



$$\begin{aligned}
[\![\texttt{Num}\ n]\!] &= \lambda E.\ n \\
[\![\texttt{Var}\ x]\!] &= \\
[\![\texttt{Plus}\ e_1\ e_2]\!] &= \\
[\![\texttt{Times}\ e_1\ e_2]\!] &= \\
[\![\texttt{Let}\ x\ e_1\ e_2]\!] &= \lambda E.
\end{aligned}$$

## Denotational Semantics



$$[\![\cdot]\!] : \mathbf{AST} \to (\mathbf{Var} \rightharpoonup \mathbb{Z}) \to \mathbb{Z}$$

Our denotation fo *ents* (mapping from variables to v



$$\begin{aligned}
[\![\texttt{Num}\ n]\!] &= \lambda E.\ n \\
[\![\texttt{Var}\ x]\!] &= \lambda E. \\
[\![\texttt{Plus}\ e_1\ e_2]\!] &= \lambda E. \\
[\![\texttt{Times}\ e_1\ e_2]\!] &= \lambda E. \\
[\![\texttt{Let}\ x\ e_1\ e_2]\!] &= \lambda E.
\end{aligned}$$

## Denotational Semantics



$$[\![\cdot]\!] : \mathbf{AST} \to (\mathbf{Var} \rightharpoonup \mathbb{Z}) \to \mathbb{Z}$$

Our denotation fo *ents* (mapping from variables to t



$$\begin{aligned}
[\![\texttt{Num}\ n]\!] &= \lambda E.\ n \\
[\![\texttt{Var}\ x]\!] &= \lambda E. \\
[\![\texttt{Plus}\ e_1\ e_2]\!] &= \lambda E. \\
[\![\texttt{Times}\ e_1\ e_2]\!] &= \lambda E. \\
[\![\texttt{Let}\ x\ e_1\ e_2]\!] &= \lambda E.
\end{aligned}$$

## Denotational Semantics



$$[\![\cdot]\!] : \mathbf{AST} \to (\mathbf{Var} \nrightarrow \mathbb{Z}) \to \mathbb{Z}$$

Our denotation fo *ents* (mapping from variables to t



$$\begin{aligned}
[\![\texttt{Var } x]\!] &= \lambda E.\ E( \\
[\![\texttt{Plus } e_1\ e_2]\!] &= \lambda E.\ [\![e_1 \\
[\![\texttt{Times } e_1\ e_2]\!] &= \lambda E.\ [\![e_1 \\
[\![\texttt{Let } x\ e_1\ e_2]\!] &= \lambda E.\ [\![e_2]\!]\ (E[x := [\![e_1]\!]E])
\end{aligned}$$



Where $E[x := n]$ is a new environment just like $E$ except the variable $x$ now maps to $n$.

## Operational Semantics

There are two main kinds of operational semantics.

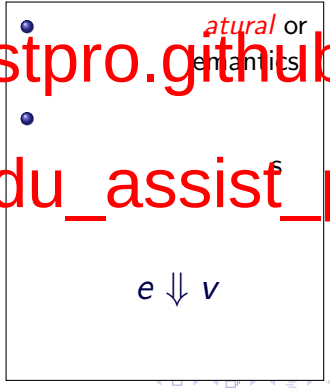
# Operational Semantics

There are two main kinds of operational semantics.



Small Step

Big Step



# Operational Semantics

There are two main kinds of operational semantics.



## Small Step

- Also cal
  *operat*
  *seman*
- A judgement that specifies transitions between *states*:

$$e \mapsto e'$$

## Big Step

- *atural* or
  emantics
- s

$$e \Downarrow v$$

## Big-Step Semantics

We need:

- A set of evaluable expressions $E$
- A set of values $V$
- A relation



**Example (Arith**

$E$ is the set of all clos $\{ \mid \}$ $\mathbb{Z}$.

$$\frac{}{(\texttt{Num}\ n) \Downarrow n}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (v_1 + v_2)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Times}\ e_1\ e_2) \Downarrow (v_1 \times v_2)}$$

**To Code** Let's do it in Haskell!

## Big-Step Semantics

We need:

- A set of evaluable expressions $E$
- A set of values $V$
- A relation



**Example (Arith**

$E$ is the set of all clos $\{ \mid \}$ $\mathbb{Z}$.

$$\frac{}{(\texttt{Num}\ n) \Downarrow n}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (v_1 + v_2)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Times}\ e_1\ e_2) \Downarrow (v_1 \times v_2)}$$

**To Code** Let's do it in Haskell!

# Big-Step Semantics

We need:

- A set of evaluable expressions $E$
- A set of values $V$
- A relation



**Example (Arith**

$E$ is the set of all clos $\{ \mid \}$ $\mathbb{Z}$.

$$\frac{}{(\texttt{Num}\ n) \Downarrow n} \qquad (\texttt{Let}\ e_1$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (v_1 + v_2)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Times}\ e_1\ e_2) \Downarrow (v_1 \times v_2)}$$

**To Code** Let's do it in Haskell!

# Big-Step Semantics

We need:

- A set of evaluable expressions $E$
- A set of values $V$
- A relation



**Example (Arith**

$E$ is the set of all clos $\{ \mid \}$ $\mathbb{Z}$.

$$\frac{}{(\texttt{Num}\ n) \Downarrow n} \qquad \frac{e_1 \Downarrow v_1}{(\texttt{Let}\ e_1$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (v_1 + v_2)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Times}\ e_1\ e_2) \Downarrow (v_1 \times v_2)}$$

**To Code** Let's do it in Haskell!

# Big-Step Semantics

We need:

- A set of evaluable expressions $E$
- A set of values $V$
- A relation



**Example (Arith**

$E$ is the set of all clos { | } $\mathbb{Z}$.

$$\frac{e_1 \Downarrow v_1 \qquad e_2[}{(\texttt{Let}\ e_1}$$

$$\frac{}{(\texttt{Num}\ n) \Downarrow n} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (v_1 + v_2)} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(\texttt{Times}\ e_1\ e_2) \Downarrow (v_1 \times v_2)}$$

**To Code** Let's do it in Haskell!

## Evaluation Strategies

$$\frac{e_1 \Downarrow v_1 \qquad e_2[x := (\texttt{Num}\ v_1)] \Downarrow v_2}{(\texttt{Let}\ e_1\ (x)\ e_2) \Downarrow v_2}$$



Any other ways to e

## Evaluation Strategies

$$\frac{e_1 \Downarrow v_1 \qquad e_2[x := (\texttt{Num}\ v_1)] \Downarrow v_2}{(\texttt{Let}\ e_1\ (x.\ e_2)) \Downarrow v_2}$$



Any other ways to e

The above is called *call-by-name*:



$$\frac{}{(\texttt{Let}\ e_1\ (x.\ e_2)}$$

# Evaluation Strategies

$$\frac{e_1 \Downarrow v_1 \qquad e_2[x := (\texttt{Num}\ v_1)] \Downarrow v_2}{(\texttt{Let}\ e_1\ (x.\ e_2)) \Downarrow v_2}$$



Any other ways to e

The above is called *call-by-name*:



$$\frac{}{(\texttt{Let}\ e_1\ (x.\ e_2))}$$

This can be computationally very expensive, for example:



$\texttt{let}\ x = \langle \textit{very expensive computatio}$

In confluent languages like this or $\lambda$-calculus, this only matters for performance. In other languages, this is not so. Why?

# Evaluation Strategies



$$\frac{e_1 \Downarrow v_1 \qquad e_2[x := (\texttt{Num}\ v_1)] \Downarrow v_2}{(\texttt{Let}\ e_1\ (x.\ e_2)) \Downarrow v_2}$$

Any other ways to e

The above is called *call-by-name*:



$$\frac{}{(\texttt{Let}\ e_1\ (x.\ e_2)}$$



This can be computationally very expensive, for example:

$\texttt{let}\ x = \langle \textit{very expensive computatio}$

In confluent languages like this or $\lambda$-calculus, this only matters for performance. In other languages, this is not so. Why?

Haskell uses *call-by-need* or lazy evaluation, which optimises cases like this.

## Small Step Semantics



For small step semantics, we need:

- A set of states
- A set of initial s
- A set of final st
- A relation $\mapsto \subseteq \Sigma \times \Sigma$, which specifies only "one st





An *execution* or trace $\sigma_1 \mapsto \sigma_2 \mapsto \sigma_3 \mapsto \cdots$ is $\sigma$ e exists no $\sigma_{n+1}$ such that $\sigma_n \mapsto \sigma_{n+1}$; and is called *comple* $_n \in F$.

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e \text{ ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Plu}}$$



**To Code** Let's do it in Haskell!

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e \text{ ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Plu}}$$



**To Code** Let's do it in Haskell!

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e \textbf{ ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Num}\ n)} \qquad \frac{}{(\texttt{Num}\ m)}$$

$$\frac{}{(\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m))}$$



**To Code** Let's do it in Haskell!

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e \textbf{ ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Num}\ n) \Downarrow (\texttt{Num}\ n)} \qquad \frac{e_1 \Downarrow (\texttt{Num}\ n) \qquad e_2 \Downarrow (\texttt{Num}\ m)}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m))}$$

**To Code** Let's do it in Haskell!

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e \textbf{ ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Num}\ n) \Downarrow (\texttt{Num}\ n)} \qquad \frac{e_1 \Downarrow (\texttt{Num}\ n) \quad e_2 \Downarrow (\texttt{Num}\ m)}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m))}$$

**To Code** Let's do it in Haskell!

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e\ \mathbf{ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Num}\ n) \Downarrow (\texttt{Num}\ n)} \qquad \frac{e_1 \Downarrow (\texttt{Num}\ n) \quad e_2 \Downarrow (\texttt{Num}\ m)}{(\texttt{Plus}\ e_1\ e_2) \Downarrow (\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m))}$$

(Similarly for T

**To Code** Let's do it in Haskell!

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e \text{ ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m)) \mapsto (\texttt{Num}\ n+m)}$$

(Similarly for T

$$\frac{}{(\texttt{Let}\ e_1\ (x.\ e_2)) \mapsto}$$

**To Code** Let's do it in Haskell!

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e \text{ ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m)) \mapsto (\texttt{Num}\ n+m)}$$

$$\frac{e_1 \mapsto e_1'}{(\texttt{Let}\ e_1\ (x.\ e_2)) \mapsto (\texttt{Let}\ e_1'\ (x.\ e_2))}$$

(Similarly for T

**To Code** Let's do it in Haskell!

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e\ \mathbf{ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m)) \mapsto (\texttt{Num}\ n+m)} \qquad \frac{e_1 \mapsto e_1'}{(\texttt{Plus}\ e_1\ e_2) \mapsto (\texttt{Plus}\ e_1'\ e_2)}$$

(Similarly for T

$$\frac{e_1 \mapsto e_1'}{(\texttt{Let}\ e_1\ (x.\ e_2)) \mapsto (\texttt{Let}\ e_1'\ (x.\ e_2))}$$

$$\frac{}{(\texttt{Let}\ (\texttt{Num}\ n)\ (x.\ e_2)) \mapsto}$$

**To Code** Let's do it in Haskell!

# Example

**Example (Arithmetic Expressions)**

$\Sigma$ and $V$ are the set of all closed expressions $\{e \mid e\ \mathbf{ok}\}$. $F$ is the set of evaluated expressions $\{(\texttt{Num}\ n) \mid n \in \mathbb{Z}\}$.

$$\frac{}{(\texttt{Plus}\ (\texttt{Num}\ n)\ (\texttt{Num}\ m)) \mapsto (\texttt{Num}\ n+m)} \qquad \frac{e_1 \mapsto e_1'}{(\texttt{Plus}\ e_1\ e_2) \mapsto (\texttt{Plus}\ e_1'\ e_2)}$$

(Similarly for T

$$\frac{e_1 \mapsto e_1'}{(\texttt{Let}\ e_1\ (x.\ e_2)) \mapsto (\texttt{Let}\ e_1'\ (x.\ e_2))}$$

$$\frac{}{(\texttt{Let}\ (\texttt{Num}\ n)\ (x.\ e_2)) \mapsto e_2[x := \texttt{Num}\ n]}$$

**To Code** Let's do it in Haskell!

# Equivalence



**Comparing small step and big step**

Small step seman
step semantics gi



Having specified t
show they are equivalent, that is:



*If there exists a trace* $e \mapsto \cdots \mapsto (\mathtt{Num}$

We will need to define some notation to remove those blasted magic dots.

## Notation



Let $\overset{\star}{\mapsto}$ be the *reflexive, transitive closure* of $\mapsto$.



We can now state our property formally as:



$$e \overset{\star}{\mapsto} (\mathtt{Num}\ n) \iff$$

# Doing the Proof

The proof will be done on the iPad with typeset versions being uploaded as usual. The big-step to small-step direction can be proven by reasonably straightforward rule induction:





$$\frac{}{v \Downarrow v}$$

The other direction requires the lemma:



$$\frac{e \mapsto e' \quad e' \Downarrow n}{e \Downarrow n}$$

The abridged proof is presented in this lecture, with all cases left for the course website.