WELLESLEY

Programming Languages

CS 251

Fall 2021

Carolyn Anderson

Recap

Our big step semantics for Racket

- **Values**: expressions that cannot be reduced any further

  Value rule: v ↓ v

- **Expressions**: bits of the language

  Addition rule:

$$\frac{e1 \downarrow v1 \quad e2 \downarrow v2}{(+\ e1\ e2) \downarrow v}$$

  where v1 and v2 are numbers and v is the sum of v1 and v2

Language components

- **Values**: expressions that cannot be reduced any further
- **Expressions**: bits of the language
- **Declarations**: bind variables to values

Function application

Syntax: (e_1 e_2)

function **argument**

Semantics: ????

What happens when a function is **applied**?

Function application

What happens when a function is **applied**?

First, there's a variable binding part:
When a function is applied to a value, the value gets **bound** to the function's **parameter** inside the **scope** of the function.

Binding as substitution

The Substitution Model of Variable Binding:
When a value *v* is bound to an expression ***e***, substitute the value *v* for every **unbound** occurrence of ***e*** in the scope of the binder.

Function application

Syntax: (e_1 e_2)

Semantics: ???

Let's consider this function application:

((lambda (x) x) 5)

Function application

Syntax: (e_1 e_2)

Semantics: ???

1. Bind the value to the function parameter within the function body, using the substitution model of variable binding.

((lambda (x) x) 5)   (~~lambda (x)~~ 5)

Substitution notation

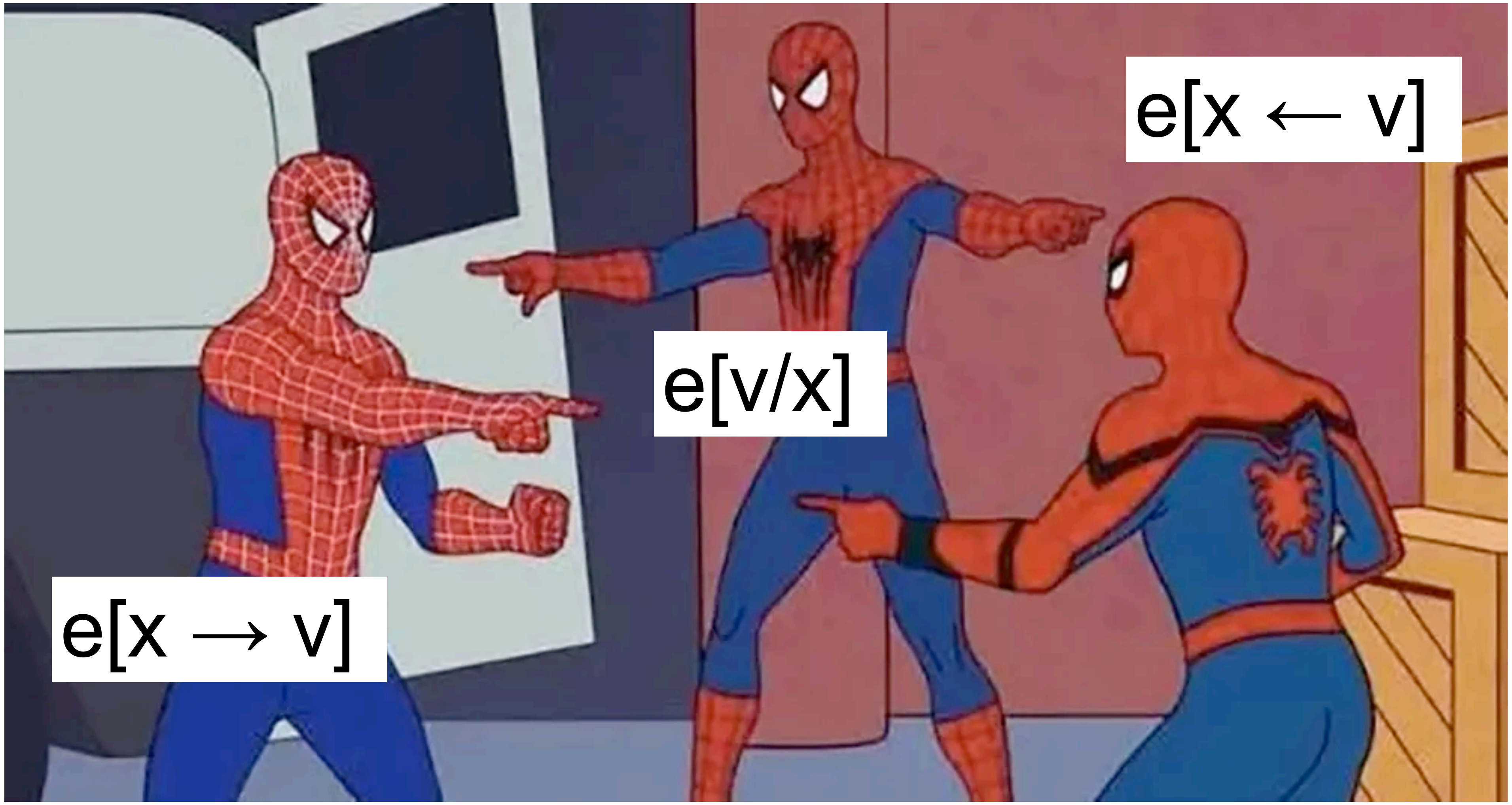
We need notation to represent substitution:

$e[x \rightarrow v]$ represents the result of substituting all unbound occurrences of x in e with v.

Substitution notation

We need notation to represent substitution:

$e[x \rightarrow v]$ represents the result of substituting all unbound occurrences of x in e with v.



Semantics of application

Syntax: (e_1 e_2)

Semantics: ???

1. Bind the value to the function parameter within the function body, using the substitution model of variable binding.

Semantics of application

Syntax: (e_1 e_2)

Semantics: ???

0. Evaluate e_1 to a value v_1. If v_1 is a function:

1. Bind the value to the function parameter within the function body, using the substitution model of variable binding.

Semantics of application

Syntax: $(e_1\ e_2)$

Semantics: ???

0. Evaluate e_1 to a value v_1. If v_1 is a function with parameter x and body e_b:

1. Bind the value to the function parameter within the function body, using the substitution model of variable binding.

Semantics of application

Syntax: (e_1 e_2)

Semantics: ???

0. Evaluate e_1 to a value v_1. If v_1 is a function with parameter x and body e_b:

1. Evaluate e_2 to a value v_2.

2. Bind v_2 to x within e_b, using the substitution model of variable binding.

Semantics of application

Syntax: $(e_1\ e_2)$

Semantics:

$$\frac{e_1 \downarrow (\text{lambda } (x)\ e_b) \qquad e_2 \downarrow v_2}{(e_1\ e_2) \downarrow e_b[x \rightarrow v_2]}$$

0. Evaluate e_1 to a value v_1. If v_1 is a function with parameter x and body e_b:

1. Evaluate e_2 to a value v_2.

2. Bind v_2 to x within e_b, using the substitution model of variable binding.

Semantics of application

Let's test our semantics!

Practice:

Write down the big step semantics for **let**.

Syntax: (let ((x e_1)) e_2)

Semantics:

Semantics of local binding

Syntax: (let ((x e_1)) e_2)

Semantics:

$$\frac{e_1 \downarrow v}{(\text{let } ((x\ e_1))\ e_2) \downarrow e_2[x \rightarrow v]}$$

Context

One property of our big step semantics is that it doesn't model context.

Our rules stipulate the same behavior for a given expression regardless of where it occurs.

Side effects revisited

- **Side effect**: any observable effect other than producing a value
- More formally:

  An expression has a side effect if it changes its own **context**.

- Mutation is a side effect because it changes a variable's value within the current scope (unlike **let**). This makes the variable's behavior **context-dependent**: you have to know whether you are referencing it before or after the mutation.

Side effects revisited

- An expression has a side effect if it changes its own **context**.
- Errors are a kind of side effect. Why?

Side effects revisited

- An expression has a side effect if it changes its **context**.
- Errors are a kind of side effect. They halt evaluation, making the evaluation of later expressions context-dependent.
- Example program: `e1 e2`

  If `e1` results in an error, `e2` will not be evaluated!
- Although functional programming languages are often described as "side effect free", they give rise to errors just like any other language!

What do we do about errors?

In our big step semantics, we can describe situations where errors arise. But we won't track errors, since that requires representing the program context (hard 😢).

When we hit an error, we'll just abandon the derivation.