Subprograms IV: The Substitution Model

CAS CS 320: Principles of Programming Languages

Thursday, April 18, 2024

Administrivia

- Project 2 is already posted and due on Monday, Apr 22.
- Final exam on Wednesday, May 8, 3:00-5:00 pm in STO 50.

What Is the *Substitution Model*?

We have already practiced it in preceding lectures without explicitly naming it ... It is a simple and straightforward method for *evaluating* programming languages.

What Is the *Substitution Model*?

We have already practiced it in preceding lectures without explicitly naming it ... It is a simple and straightforward method for *evaluating* programming languages.

Although there are hundreds of cases to deal with in even a semi-realistic language, everything is reduced to a set of well-defined reduction/evaluation rules. The basic idea is simple – stated approximately:

What Is the *Substitution Model*?

We have already practiced it in preceding lectures without explicitly naming it ... It is a simple and straightforward method for *evaluating* programming languages.

Although there are hundreds of cases to deal with in even a semi-realistic language, everything is reduced to a set of well-defined reduction/evaluation rules. The basic idea is simple – stated approximately:

- **evaluate subexpressions to values,**
- **when you have a function call, *substitute* argument value for the formal parameter in the function body,**
- **evaluate the resulting expression.**

What Is the *Substitution Model*?

But the substitution model is not without its *shortcomings*:

- **It is not straightforward to extend the model with support for side effects (e.g., reference assignments)**

- **It is not a very efficient model of how we really evaluate realistic programs.**

What Is the *Substitution Model*?

But the substitution model is not without its *shortcomings*:

- **It is not straightforward to extend the model with support for side effects (e.g., reference assignments)**
- **It is not a very efficient model of how we really evaluate realistic programs.**

And this is in contrast to what is called the *environment model* – a somewhat more realistic model, a little closer to how an interpreter/compiler actually operates, which we have also used (in limited ways) in preceding lectures without explicitly naming it ...

What Is the *Substitution Model*?

But the substitution model is not without its *shortcomings*:

- It is not straightforward to extend the model with support for side effects (e.g., reference assignments)

- It is not a very efficient model of how we really evaluate realistic programs.

And this is in contrast to what is called the *environment model* – a somewhat more realistic model, a little closer to how an interpreter/compiler actually operates, which we have also used (in limited ways) in preceding lectures without explicitly naming it ...

In this lecture we focus on the substitution model and then add a few comments on the environment model.

Preliminary Remarks

In the preceding 2-3 lectures we considered a *toy stack-oriented language*, which is close to the language used in the homework projects ...

A reminder is in the next few slides.

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num>
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

```
Example Program:
x = 1;
y = 2;
x = 3;
x;
y;
```

This is a simple language with assignment statements and a push operation for identifiers.

We will take a *configuration* to be:

environment

(S, E, P) or **ERROR**

stack program

Operational Semantics

$$\frac{}{\texttt{(S, E , x = n ; P)} \longrightarrow \texttt{(S, }\textbf{update}\texttt{(E, x, n), P)}}\ \texttt{(assign)}$$

$$\frac{\textbf{fetch}\texttt{(E, x) = n} \qquad \texttt{n is a number}}{\texttt{(S, E , x ; P)} \longrightarrow \texttt{(n :: S, E, P)}}\ \texttt{(fetch)}$$

$$\frac{\textbf{fetch}\texttt{(E, x) is not a number}}{\texttt{(S, E , x ; P)} \longrightarrow \textbf{ERROR}}\ \texttt{(fetchErr)}$$

Environments

$$(x \mapsto v \,,\, \overbrace{y \mapsto w}^{\texttt{binding}} \,,\, \underbrace{z}_{\texttt{variable}} \mapsto \underbrace{n}_{\texttt{value}})$$

An environment is a collection of **bindings.**

The exact way you implement an environment depends on the situation.

In the project we use an association list:

(ident * value) list

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num> | { <prog> }
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

Example Program:

```
x = 1;
f = {
  x = 2;
  x;
};
f;
x;
```

This is a simple language with assignment statements and a push operation for identifiers **and subroutines.**

We will take a *configuration* to be:

environment

(S, E, P) or **ERROR**

stack program

Operational Semantics

$$\frac{}{(\mathtt{S},\ \mathtt{E}\ ,\ \mathtt{x}\ =\ \mathtt{n}\ ;\ \mathtt{P}) \longrightarrow (\mathtt{S},\ \mathbf{update}(\mathtt{E},\ \mathtt{x},\ \mathtt{n}),\ \mathtt{P})}\ (\mathtt{assign})$$

$$\frac{\mathbf{fetch}(\mathtt{E},\ \mathtt{x}) = \mathtt{n} \qquad \mathtt{n\ is\ a\ number}}{(\mathtt{S},\ \mathtt{E}\ ,\ \mathtt{x}\ ;\ \mathtt{P}) \longrightarrow (\mathtt{n}\ ::\ \mathtt{S},\ \mathtt{E},\ \mathtt{P})}\ (\mathtt{fetchPush})$$

$$\frac{\mathbf{fetch}(\mathtt{E},\ \mathtt{f}) = \mathtt{Q} \qquad \mathtt{Q\ is\ a\ program}}{(\mathtt{S},\ \mathtt{E}\ ,\ \mathtt{f}\ ;\ \mathtt{P}) \longrightarrow (\mathtt{S},\ \mathtt{E},\ \mathtt{Q}\ \mathtt{P})}\ (\mathtt{fetchCall})$$

$$\frac{\mathbf{fetch}(\mathtt{E},\ \mathtt{x})\ \mathtt{is\ None}}{(\mathtt{S},\ \mathtt{E}\ ,\ \mathtt{x}\ ;\ \mathtt{P}) \longrightarrow \mathbf{ERROR}}\ (\mathtt{fetchErr})$$

Preliminary Remarks

In this lecture we will use two small languages:

The language of *arithmetical expressions* (once more) and a small fragment of OCaml (once more), here called *FunLang*.

In the course of our presentation we revisit operational semantics by making a distinction between two versions:

***small-step operational semantics* and**

***big-step operational semantics*.**

Small-Step Operational Semantics

- A reduction relation ($P \rightarrow Q$) relating program configurations P and Q is defined.
- A witness (proof) of the reduction relation can be constructed by applying a set of rules.
- If we can prove that $P \rightarrow Q$, then we say P reduces to Q in 1 step.
- The transitive reflexive closure $P \rightarrow^{*} Q$ of the single step reduction indicates that P reduces to Q in 0 or more steps.

Small-Step Operational Semantics

$$\frac{e_1 \rightarrow e_1'}{(\text{add } e_1 e_2) \rightarrow (\text{add } e_1' \, e_2)} \text{ add-left}$$

$$\frac{n \in \mathbb{Z} \qquad e \rightarrow e'}{(\text{add } n \; e) \rightarrow (\text{add } n \; e')} \text{ add-right}$$

$$\frac{n_1 \in \mathbb{B} \cup \{\text{ ERROR }\}}{(\text{add } n_1 n_2) \rightarrow \text{ERROR}} \text{ add-left-error}$$

$$\frac{n_1 \in \mathbb{Z} \qquad n_2 \in \mathbb{B} \cup \{\text{ ERROR }\}}{(\text{add } n_1 n_2) \rightarrow \text{ERROR}} \text{ add-right-error}$$

$$\frac{n_1 \in \mathbb{Z} \qquad n_2 \in \mathbb{Z}}{(\text{add } n_1 n_2) \rightarrow (n_1 + n_2)} \text{ add-ok}$$

$$\frac{}{\text{P} \rightarrow^* \text{P}} \text{ reflexivity}$$

$$\frac{\text{P} \rightarrow \text{Q} \qquad \text{Q} \rightarrow^* \text{R}}{\text{P} \rightarrow^* \text{R}} \text{ transitive}$$

Small-Step Operational Semantics

$$\textcolor{red}{(1)}\quad \cfrac{12 \in \mathbb{Z} \qquad \cfrac{10 \in \mathbb{Z} \qquad 5 \in \mathbb{Z}}{(\text{add } 10\ 5) \to 15}\ \text{add-ok}}{(\text{add } 12\ (\text{add } 10\ 5)) \to (\text{add } 12\ 15)}\ \text{add-right}$$

$$\frac{\textcolor{red}{(1)} \qquad\qquad \textcolor{red}{(2)}}{(\text{add } 12\ (\text{add } 10\ 5)) \to^{*} 27}\ \text{transitive}$$

$$\textcolor{red}{(2)}\quad \cfrac{\cfrac{12 \in \mathbb{Z} \qquad 15 \in \mathbb{Z}}{(\text{add } 12\ 15) \to 27}\ \text{add-ok} \qquad \cfrac{}{27 \to^{*} 27}\ \text{reflexivity}}{(\text{add } 12\ 15) \to^{*} 27}\ \text{transitive}$$

Big-Step Operational Semantics

- An evaluation relation $(P \Downarrow V)$ relating program P and value V is defined.
- A witness (proof) of the evaluation relation can be constructed by applying a set of rules.
- If we can prove that $P \Downarrow V$, then we say P evaluates to V.
- The evaluation relation states that P evaluates to V in a finite number of steps. No need to define transitive reflexive closure like in small-step semantics.

Big-Step Operational Semantics

$$\frac{v \in \mathbb{Z}}{v \Downarrow v}\ \text{int-ok}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1, v_2 \in \mathbb{Z}}{(\text{add}\ e_1 e_2) \Downarrow (v_1 + v_2)}\ \text{add-ok}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad v_1 \notin \mathbb{Z} \lor v_2 \notin \mathbb{Z}}{(\text{add}\ e_1 e_2) \Downarrow \text{ERROR}}\ \text{add-error}$$

Big-step rules are comparatively easy to define. Every $\mathrm{P} \Downarrow V$ could encompass multiple small-step reductions.

The trade-off is that big-step rules lack the precision of small-step rules. Notice that the big-step rules for addition no long specifies the evaluation order of its arguments.

Big-Step Operational Semantics

What can we formally say about the following reduction relation without any knowledge of the right hand side?

$$(\text{add } 12 \ (\text{add } 10 \ 5)) \Downarrow ??$$

Big-Step Operational Semantics

Two proof obligations.

$$\frac{\boxed{\dfrac{??}{12 \Downarrow X}} \qquad \boxed{\dfrac{??}{(\text{add } 10\ 5) \Downarrow Y}} \qquad X, Y \in \mathbb{Z}}{\big(\text{add } 12\ (\text{add } 10\ 5)\big) \Downarrow (X + Y)}\ \text{add-ok}$$

Big-Step Operational Semantics

$$\dfrac{\dfrac{12 \in \mathbb{Z}}{12 \Downarrow 12}\,\text{int-ok} \qquad \boxed{\dfrac{??}{(\text{add } 10\ 5) \Downarrow Y}} \qquad 12, Y \in \mathbb{Z}}{\left(\text{add } 12\ (\text{add } 10\ 5)\right) \Downarrow (12 + Y)}\,\text{add-ok}$$

Big-Step Operational Semantics

$$
\dfrac{
\dfrac{12 \in \mathbb{Z}}{12 \Downarrow 12}\,\text{int-ok}
\qquad
\dfrac{
\dfrac{??}{10 \Downarrow X}\,\text{int-ok}
\quad
\dfrac{??}{5 \Downarrow Y}\,\text{int-ok}
\quad
X, Y \in \mathbb{Z}
}{(\text{add } 10\ 5) \Downarrow (X + Y)}\,\text{add-ok}
\qquad
12, (X + Y) \in \mathbb{Z}
}{\big(\text{add } 12\ (\text{add } 10\ 5)\big) \Downarrow (12 + (X + Y))}\,\text{add-ok}
$$

Big-Step Operational Semantics

$$\dfrac{\dfrac{12 \in \mathbb{Z}}{12 \Downarrow 12}\,\text{int-ok} \quad \dfrac{\dfrac{10 \in \mathbb{Z}}{10 \Downarrow 10}\,\text{int-ok} \quad \dfrac{5 \in \mathbb{Z}}{5 \Downarrow 5}\,\text{int-ok} \quad 10, 5 \in \mathbb{Z}}{(\text{add } 10\ 5) \Downarrow (10 + 5)}\,\text{add-ok} \quad 12, (10 + 5) \in \mathbb{Z}}{\big(\text{add } 12\ (\text{add } 10\ 5)\big) \Downarrow (12 + (10 + 5))}\,\text{add-ok}$$

Big-Step Operational Semantics

$$\dfrac{\dfrac{12 \in \mathbb{Z}}{12 \Downarrow 12}\,\text{int-ok} \qquad \dfrac{\dfrac{10 \in \mathbb{Z}}{10 \Downarrow 10}\,\text{int-ok} \qquad \dfrac{5 \in \mathbb{Z}}{5 \Downarrow 5}\,\text{int-ok} \qquad 10, 5 \in \mathbb{Z}}{(\text{add } 10\ 5) \Downarrow 15}\,\text{add-ok} \qquad 12, 15 \in \mathbb{Z}}{\big(\text{add } 12\ (\text{add } 10\ 5)\big) \Downarrow 27}\,\text{add-ok}$$

Operational Semantics of FunLang

```
<expr> ::= <var> | <int> | <expr> + <expr>
         | fun <var> -> <expr>
         | let <var> = <expr> in <expr>
         | <expr> <expr>
<var>  ::= variables
<int>  ::= integers
```

FunLang is a tiny functional language with first class functions. To define its operational semantics, we need a notion of variable substitution.

We use the notation $m[n/x]$ to mean substitute expression n for variable x in expression m.

Operational Semantics of FunLang

Examples of substitution:

- x[1/x] = 1
- (x + x)[1/x] = 1 + 1
- (fun x -> y)[fun z -> z/y] = fun x -> fun z -> z
- (let x = y + z in z + x)[9/z] = let x = y + 9 in 9 + x
- (fun y -> x + x)[let z = 1 in z + z/x] = fun y -> (let z = 1 in z + z) + (let z = 1 in z + z)

Operational Semantics of FunLang

When dealing with substitution, there are some conditions to be careful about.

Bound variables do not get substituted.

- (fun x -> x)[y/x] = fun x -> x
- (let x = y in y + x)[z + z/x] = let x = y in y + x

Bound variables must be renamed to avoid "capturing" when substituting in an expression containing said bound variable.

- (fun x -> y)[x + x/y] = (fun z -> y)[x + x/y] = fun z -> x + x
- (let x = y in y + x)[x + x/y] = (let z = y in y + z)[x + x/y] = let z = x + x in x + x + z

CBN Operational Semantics of FunLang

$$\frac{e_1 \to e_1{}'}{e_1 + e_2 \to e_1' + e_2}\ \text{add-left} \qquad \frac{n\ \in\ \mathbb{Z} \qquad e \to e'}{n + e \to n +\ e'}\ \text{add-right} \qquad \frac{n_1 \in\ \mathbb{Z} \qquad n_2 \in\ \mathbb{Z}}{n_1 + n_2 \to\ (n_1 + n_2)}\ \text{add-ok}$$

$$\frac{e_1 \to e_1{}'}{e_1\ e_2 \to e_1'\ e_2}\ \text{app-left} \qquad \frac{}{(\text{fun}\ x\ \to e_1)\ e_2 \to e_1[e_2/x]}\ \text{cbn-beta}$$

$$\frac{}{\text{let}\ x = e_1 \text{in}\ e_2 \to e_2[e_1/x]}\ \text{cbn-zeta}$$

CBN Operational Semantics of FunLang

$$\frac{e_1 \rightarrow {e_1}'}{e_1 + e_2 \rightarrow e_1' + e_2}\ \text{add-left} \qquad \frac{n \in \mathbb{Z} \qquad e \rightarrow e'}{n + e \rightarrow n + e'}\ \text{add-right} \qquad \frac{n_1 \in \mathbb{Z} \qquad n_2 \in \mathbb{Z}}{n_1 + n_2 \rightarrow (n_1 + n_2)}\ \text{add-ok}$$

$$\frac{e_1 \rightarrow {e_1}'}{e_1\ e_2 \rightarrow e_1'\ e_2}\ \text{app-left} \qquad \boxed{\frac{}{(\text{fun}\ x\ \rightarrow e_1)\ e_2 \rightarrow e_1[e_2/x]}\ \text{cbn-beta}}$$

$$\boxed{\frac{}{\text{let}\ x = e_1 \text{in}\ e_2 \rightarrow e_2[e_1/x]}\ \text{cbn-zeta}}$$

Call-by-name semantics does not impose any restrictions on the form of substituted arguments.

CBN Operational Semantics of FunLang

Consider the following example:

```
let add = fun x -> fun y -> x + y in
let addx = add (1 + 2) in
addx 2
```

This should be familiar to us as an example of partial application of functions.

How can we prove its behavior formally?

CBN Operational Semantics of FunLang

$$\frac{}{\begin{array}{l}\texttt{let add = fun x -> fun y -> x + y in} \\ \texttt{let addx = add (1 + 2) in} \\ \texttt{addx 2}\end{array} \rightarrow \begin{array}{l}\texttt{(let addx = add (1 + 2) in} \\ \texttt{ addx 2)[fun x -> fun y -> x + y /add]}\end{array}}\ \text{cbn-zeta}$$

(1)

CBN Operational Semantics of FunLang

$$\frac{}{\texttt{let add = fun x -> fun y -> x + y in let addx = add (1 + 2) in addx 2} \rightarrow \texttt{(let addx = add (1 + 2) in addx 2)[fun x -> fun y -> x + y /add]}}\ \text{cbn-zeta}$$

(1)
```
let add = fun x -> fun y -> x + y in      →   (let addx = add (1 + 2) in
let addx = add (1 + 2) in                      addx 2)[fun x -> fun y -> x + y /add]
addx 2
```

(2)
```
let addx = (fun x -> fun y -> x + y) (1 + 2) in   →   (addx 2)[(fun x -> fun y -> x + y) (1 + 2) /addx]
addx 2
```

cbn-zeta

CBN Operational Semantics of FunLang

(1)

$$\frac{}{\begin{array}{l}\texttt{let add = fun x -> fun y -> x + y in}\\ \texttt{let addx = add (1 + 2) in}\\ \texttt{addx 2}\end{array} \rightarrow \begin{array}{l}\texttt{(let addx = add (1 + 2) in}\\ \texttt{ addx 2)[fun x -> fun y -> x + y /add]}\end{array}}\ \text{cbn-zeta}$$

(2)

$$\frac{}{\begin{array}{l}\texttt{let addx = (fun x -> fun y -> x + y) (1 + 2) in}\\ \texttt{addx 2}\end{array} \rightarrow \texttt{(addx 2)[(fun x -> fun y -> x + y) (1 + 2) /addx]}}\ \text{cbn-zeta}$$

(3)

$$\frac{\dfrac{}{\texttt{(fun x -> fun y -> x + y) (1 + 2)} \rightarrow \texttt{(fun y -> x + y)[(1 + 2)/x]}}\ \text{cbn-beta}}{\texttt{(fun x -> fun y -> x + y) (1 + 2) 2} \rightarrow \texttt{(fun y -> (1 + 2) + y) 2}}\ \text{app-left}$$

CBN Operational Semantics of FunLang

$$(1)\quad \dfrac{}{\begin{array}{l}\texttt{let add = fun x -> fun y -> x + y in}\\ \texttt{let addx = add (1 + 2) in}\\ \texttt{addx 2}\end{array} \;\rightarrow\; \begin{array}{l}\texttt{(let addx = add (1 + 2) in}\\ \texttt{ addx 2)[fun x -> fun y -> x + y /add]}\end{array}}\ \text{cbn-zeta}$$

$$(2)\quad \dfrac{}{\begin{array}{l}\texttt{let addx = (fun x -> fun y -> x + y) (1 + 2) in}\\ \texttt{addx 2}\end{array} \;\rightarrow\; \texttt{(addx 2)[(fun x -> fun y -> x + y) (1 + 2) /addx]}}\ \text{cbn-zeta}$$

$$(3)\quad \dfrac{\dfrac{}{\texttt{(fun x -> fun y -> x + y) (1 + 2)} \;\rightarrow\; \texttt{(fun y -> x + y)[(1 + 2)/x]}}\ \text{cbn-beta}}{\texttt{(fun x -> fun y -> x + y) (1 + 2) 2} \;\rightarrow\; \texttt{(fun y -> (1 + 2) + y) 2}}\ \text{app-left}$$

$$(4)\quad \dfrac{}{\texttt{(fun y -> (1 + 2) + y) 2} \;\rightarrow\; \texttt{((1 + 2) + y)[2/y]}}\ \text{cbn-beta}$$

CBN Operational Semantics of FunLang

$$\frac{}{\begin{array}{l}\text{let add = fun x -> fun y -> x + y in}\\ \text{let addx = add (1 + 2) in}\\ \text{addx 2}\end{array} \rightarrow \begin{array}{l}\text{(let addx = add (1 + 2) in}\\ \text{addx 2)[fun x -> fun y -> x + y /add]}\end{array}}\ \text{cbn-zeta} \quad (1)$$

$$\frac{}{\begin{array}{l}\text{let addx = (fun x -> fun y -> x + y) (1 + 2) in}\\ \text{addx 2}\end{array} \rightarrow \text{(addx 2)[(fun x -> fun y -> x + y) (1 + 2) /addx]}}\ \text{cbn-zeta} \quad (2)$$

$$\frac{\dfrac{}{\text{(fun x -> fun y -> x + y) (1 + 2)} \rightarrow \text{(fun y -> x + y)[(1 + 2)/x]}}\ \text{cbn-beta}}{\text{(fun x -> fun y -> x + y) (1 + 2) 2} \rightarrow \text{(fun y -> (1 + 2) + y) 2}}\ \text{app-left} \quad (3)$$

$$\frac{}{\text{(fun y -> (1 + 2) + y) 2} \rightarrow \text{((1 + 2) + y)[2/y]}}\ \text{cbn-beta} \quad (4)$$

$$\frac{\dots}{(1 + 2) + 2 \rightarrow 3 + 2}\ \text{add-left} \quad (5)$$

CBN Operational Semantics of FunLang

(1)
$$\frac{}{\begin{array}{l}\text{let add = fun x -> fun y -> x + y in}\\ \text{let addx = add (1 + 2) in}\\ \text{addx 2}\end{array} \;\to\; \begin{array}{l}\text{(let addx = add (1 + 2) in}\\ \text{ addx 2)[fun x -> fun y -> x + y /add]}\end{array}}\ \text{cbn-zeta}$$

(2)
$$\frac{}{\begin{array}{l}\text{let addx = (fun x -> fun y -> x + y) (1 + 2) in}\\ \text{addx 2}\end{array} \;\to\; \text{(addx 2)[(fun x -> fun y -> x + y) (1 + 2) /addx]}}\ \text{cbn-zeta}$$

(3)
$$\frac{\dfrac{}{\text{(fun x -> fun y -> x + y) (1 + 2)} \;\to\; \text{(fun y -> x + y)[(1 + 2)/x]}}\ \text{cbn-beta}}{\text{(fun x -> fun y -> x + y) (1 + 2) 2} \;\to\; \text{(fun y -> (1 + 2) + y) 2}}\ \text{app-left}$$

(4)
$$\frac{}{\text{(fun y -> (1 + 2) + y) 2} \;\to\; \text{((1 + 2) + y)[2/y]}}\ \text{cbn-beta}$$

(5)
$$\frac{\dots}{(1 + 2) + 2 \;\to\; 3 + 2}\ \text{add-left}$$

(6)
$$\frac{\dots}{3 + 2 \;\to\; 5}\ \text{add-ok}$$

CBN Operational Semantics of FunLang

cbn-zeta

(1) let add = fun x -> fun y -> x + y in
let addx = add (1 + 2) in
addx 2
→ (let addx = add (1 + 2) in
addx 2)[fun x -> fun y -> x + y /add]

The applied argument does not reduce until later. CBN is also know as lazy.

cbn-zeta

(2) let addx = (fun x -> fun y -> x + y) (1 + 2) in
addx 2
→ (addx 2)[(fun x -> fun y -> x + y) (1 + 2) /addx]

cbn-beta

(fun x -> fun y -> x + y) (1 + 2) → (fun y -> x + y)[(1 + 2)/x]

app-left

(3) (fun x -> fun y -> x + y) (1 + 2) 2 → (fun y -> (1 + 2) + y) 2

cbn-beta

(4) (fun y -> (1 + 2) + y) 2 → ((1 + 2) + y)[2/y]

…

add-left

(5) (1 + 2) + 2 → 3 + 2

…

add-ok

(6) 3 + 2 → 5

CBN Operational Semantics of FunLang

———————— reflexive

(6) 5 $\rightarrow^*$ 5

———————— transitive

(5) 3 + 2 $\rightarrow^*$ 5

———————— transitive

(4) (1 + 2) + 2 $\rightarrow^*$ 5

———————— transitive

(3) (fun y -> (1 + 2) + y) 2 $\rightarrow^*$ 5

———————— transitive

(2) (fun x -> fun y -> x + y) (1 + 2) 2 $\rightarrow^*$ 5

———————— transitive

let addx = (fun x -> fun y -> x + y) (1 + 2) in $\rightarrow^*$ 5
addx 2

(1)

———————— transitive

let add = fun x -> fun y -> x + y in $\rightarrow^*$ 5
let addx = add (1 + 2) in
addx 2

CBV Operational Semantics of FunLang

In order to describe call-by-value semantics, we first need a notion of values. The following judgment specifies values of FunLang.

$$\frac{n \in \mathbb{Z}}{n\ \mathrm{value}}\ \text{int-val} \qquad\qquad \frac{}{\mathrm{fun}\ x\ \to m\ \mathrm{value}}\ \text{fun-val}$$

Whenever we can derive the judgment $(v\ \mathrm{value})$ for some expression v, we say that v is a value.

CBV Operational Semantics of FunLang

$$\frac{e_1 \to e_1{}'}{e_1 + e_2 \to e_1' + e_2}\ \text{add-left}$$

$$\frac{n \in \mathbb{Z} \qquad e \to e'}{n + e \to n + e'}\ \text{add-right}$$

$$\frac{n_1 \in \mathbb{Z} \qquad n_2 \in \mathbb{Z}}{n_1 + n_2 \to (n_1 + n_2)}\ \text{add-ok}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \text{app-left}$$

$$\frac{e_2 \to e_2'}{(\text{fun}\ x \to e_1)\ e_2 \to (\text{fun}\ x \to e_1)\ e_2{}'}\ \text{app-right}$$

$$\frac{v\ \text{value}}{(\text{fun}\ x \to e)\ v \to e[v/x]}\ \text{cbv-beta}$$

$$\frac{e_1 \to e_1'}{\text{let}\ x = e_1 \text{in}\ e_2 \to \text{let}\ x = e_1'\ \text{in}\ e_2}\ \text{let}$$

$$\frac{v\ \text{value}}{\text{let}\ x = v\ \text{in}\ e \to e[v/x]}\ \text{cbv-zeta}$$

CBV Operational Semantics of FunLang

$$\frac{e_1 \to e_1{}'}{e_1 + e_2 \to e_1' + e_2}\ \text{add-left}$$

$$\frac{n \in \mathbb{Z} \qquad e \to e'}{n + e \to n + e'}\ \text{add-right}$$

$$\frac{n_1 \in \mathbb{Z} \qquad n_2 \in \mathbb{Z}}{n_1 + n_2 \to (n_1 + n_2)}\ \text{add-ok}$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}\ \text{app-left}$$

$$\frac{e_2 \to e_2'}{(\text{fun}\ x \to e_1)\ e_2 \to (\text{fun}\ x \to e_1)\ e_2{}'}\ \text{app-right}$$

$$\boxed{\frac{v\ \text{value}}{(\text{fun}\ x \to e)\ v \to e[v/x]}\ \text{cbv-beta}}$$

$$\frac{e_1 \to e_1'}{\text{let}\ x = e_1 \text{in}\ e_2 \to \text{let}\ x = e_1'\ \text{in}\ e_2}\ \text{let}$$

$$\boxed{\frac{v\ \text{value}}{\text{let}\ x = v\ \text{in}\ e \to e[v/x]}\ \text{cbv-zeta}}$$

Call-by-value restricts the form of substituted arguments to be values.

CBV Operational Semantics of FunLang

(1)

$$\frac{\dfrac{}{\text{fun x -> fun y -> x + y value}}\ \text{fun-val}}{\begin{array}{l}\text{let add = fun x -> fun y -> x + y in}\\ \text{let addx = add (1 + 2) in}\\ \text{addx 2}\end{array} \rightarrow \begin{array}{l}\text{(let addx = add (1 + 2) in}\\ \text{ addx 2)[fun x -> fun y -> x + y /add]}\end{array}}\ \text{cbv-zeta}$$

(2)

$$\frac{\dfrac{\dfrac{\dots}{1 + 2 \rightarrow 3}\ \text{add-ok}}{\text{(fun x -> fun y -> x + y) (1 + 2)} \rightarrow \text{(fun x -> fun y -> x + y) 3}}\ \text{app-right}}{\begin{array}{l}\text{let addx = (fun x -> fun y -> x + y) (1 + 2) in}\\ \text{addx 2}\end{array} \rightarrow \begin{array}{l}\text{let addx = (fun x -> fun y -> x + y) 3 in}\\ \text{addx 2}\end{array}}\ \text{let}$$

CBV Operational Semantics of FunLang

$$\dfrac{\dfrac{}{\text{fun x -> fun y -> x + y value}}\ \text{fun-val}}{\begin{array}{l}\text{let add = fun x -> fun y -> x + y in}\\ \text{let addx = add (1 + 2) in}\\ \text{addx 2}\end{array} \to \begin{array}{l}\text{(let addx = add (1 + 2) in}\\ \text{ addx 2)[fun x -> fun y -> x + y /add]}\end{array}}\ \text{cbv-zeta} \quad (1)$$

$$\dfrac{\dfrac{\dfrac{\dots}{1 + 2 \to 3}\ \text{add-ok}}{\text{(fun x -> fun y -> x + y) (1 + 2)} \to \text{(fun x -> fun y -> x + y) 3}}\ \text{app-right}}{\begin{array}{l}\text{let addx = (fun x -> fun y -> x + y) (1 + 2) in}\\ \text{addx 2}\end{array} \to \begin{array}{l}\text{let addx = (fun x -> fun y -> x + y) 3 in}\\ \text{addx 2}\end{array}}\ \text{let} \quad (2)$$

The argument to the function is reduced immediately. CBV is also known as eager.

CBV Operational Semantics of FunLang

(3)
$$\dfrac{\dfrac{\dfrac{}{\text{3 value}}\ \text{int-val}}{\text{(fun x -> fun y -> x + y) 3} \;\to\; \text{(fun y -> x + y)[3/x]}}\ \text{cbv-beta}}{\begin{array}{l}\text{let addx = (fun x -> fun y -> x + y) 3 in}\\ \text{addx 2}\end{array} \;\to\; \begin{array}{l}\text{let addx = (fun y -> x + y)[3/x] in}\\ \text{addx 2}\end{array}}\ \text{let}$$

(4)
$$\dfrac{\dfrac{}{\text{(fun y -> 3 + y) value}}\ \text{fun-val}}{\begin{array}{l}\text{let addx = (fun y -> 3 + y) in}\\ \text{addx 2}\end{array} \;\to\; \text{(addx 2)[(fun y -> 3 + y)/addx]}}\ \text{cbv-zeta}$$

(5)
$$\dfrac{\dfrac{}{\text{2 value}}\ \text{int-val}}{\text{(fun y -> 3 + y) 2} \;\to\; \text{(3 + y)[2/y]}}\ \text{cbv-beta}$$

(6)
$$\dfrac{}{3 + 2 \;\to\; 5}\ \text{add-ok}$$

CBV Operational Semantics of FunLang

$$\dfrac{}{5 \rightarrow^{*} 5}\ \text{reflexive}$$

(6)

$$\dfrac{3 + 2 \rightarrow^{*} 5}{}\ \text{transitive}$$

(5)

$$\dfrac{\texttt{(fun y -> 3 + y) 2} \rightarrow^{*} 5}{}\ \text{transitive}$$

(4)

$$\dfrac{}{\texttt{let addx = fun y -> 3 + y in addx 2} \rightarrow^{*} 5}\ \text{transitive}$$

(3)

$$\dfrac{}{\texttt{let addx = (fun x -> fun y -> x + y) 3 in addx 2} \rightarrow^{*} 5}\ \text{transitive}$$

(2)

$$\dfrac{}{\texttt{let addx = (fun x -> fun y -> x + y) (1 + 2) in addx 2} \rightarrow^{*} 5}\ \text{transitive}$$

(1)

$$\dfrac{}{\texttt{let add = fun x -> fun y -> x + y in let addx = add (1 + 2) in addx 2} \rightarrow^{*} 5}\ \text{transitive}$$

Environment Based Operational Semantics

```
<expr>  ::= <var> | <value> | <expr> + <expr>
          | fun <var> -> <expr>
          | let <var> = <expr> in <expr>
          | <expr> <expr>
<value> ::= <int> | clo <var> -> (<expr>, <env>)
<env>   ::= (<var> ↦ <value>) :: <env> | []
<var>   ::= variables
<int>   ::= integers
```

Substitution is not efficient in practice. We shall consider an alternative environment based operational semantics that's more efficient when implemented.

Environment Based Operational Semantics

Closures (clo x -> (m, E)) are data structures representing functions at runtime. Expression m is the body of the function which may contain variables other than x. The environment E is used to resolve the values bound to all variables (besides x) in m.

For example:

clo x -> ((x + z + y), [z ↦ 1, y ↦ 2])

Notice that for expression (x + z + y), besides variable x which is the argument of the closure, variables y and z have binding in [z ↦ 1, y ↦ 2].

Environment Based Operational Semantics

We will define the big-step evaluation relation $\mathrm{E}\,; e \Downarrow v$ to indicate the expression e evaluates to value v under environment E.

Our notion of values is slightly different from before.

$$\frac{n \in \mathbb{Z}}{n \text{ value}}\ \text{int-val} \qquad\qquad \frac{}{\text{clo } x \ \to (m, E) \text{ value}}\ \text{clo-val}$$

Environment Based Operational Semantics

$$\frac{v \in \mathbb{Z}}{E; v \Downarrow v}\ \text{int-ok}$$

$$\frac{E; e_1 \Downarrow v_1 \quad E; e_2 \Downarrow v_2 \quad v_1, v_2 \in \mathbb{Z}}{E; e_1 + e_2 \Downarrow (v_1 + v_2)}\ \text{add-ok}$$

$$\frac{}{E[x \mapsto v]; x \Downarrow v}\ \text{var-ok}$$

$$\frac{}{E; \text{fun}\ x\ \rightarrow m \Downarrow \text{clo}\ x\ \rightarrow (m, E)}\ \text{fun-clo}$$

$$\frac{E; e_1 \Downarrow \text{clo}\ x\ \rightarrow (m, E') \quad E; e_2 \Downarrow u \quad E'[x \mapsto u]; m \Downarrow v}{E; e_1\ e_2 \Downarrow v}\ \text{app-ok}$$

$$\frac{E; e_1 \Downarrow u \quad E[x \mapsto u]; e_2 \Downarrow v}{E; \text{let}\ x = e_1 \text{in}\ e_2 \Downarrow v}\ \text{let-ok}$$

Environment Based Operational Semantics

$$\dfrac{\dfrac{}{[];(\text{fun } x \to \text{fun } y \to x+y) \Downarrow \text{clo } x \to (\text{fun } y \to x+y, [])}\,\text{fun-clo} \qquad \dfrac{}{[];1 \Downarrow 1}\,\text{int-ok} \qquad \dfrac{}{[x \mapsto 1];\ \text{fun } y \to x+y \Downarrow \text{clo } y \to (x+y, [x \mapsto 1])}\,\text{fun-clo}}{(1) \qquad [];(\text{fun } x \to \text{fun } y \to x+y)\ 1 \Downarrow \text{clo } y \to (x+y, [x \mapsto 1])}\,\text{app-ok}$$

$$\dfrac{(1) \qquad [];2 \Downarrow 2 \qquad \dfrac{\dfrac{}{[x \mapsto 1, y \mapsto 2];x \Downarrow 1}\,\text{var-ok} \quad \dfrac{}{[x \mapsto 1, y \mapsto 2];y \Downarrow 2}\,\text{var-ok} \quad 1,2 \in \mathbb{Z}}{[x \mapsto 1, y \mapsto 2];x+y \Downarrow 3}}{[];\big((\text{fun } x \to \text{fun } y \to x+y)\ 1\big)\ 2 \Downarrow 3}\,\text{app-ok}$$

(THIS PAGE INTENTIONALLY LEFT BLANK)