

# Definitional Translation

CS 1520 (Spring 2025)

Harvard University

Tuesday, February 25, 2025

# Today, we will learn to

- ▶ translate from one language to another using a definitional translation
- ▶ simplify writing small-step evaluations using evaluation contexts
- ▶ reason about adequacy of translations

# Definitional Translation

- ▶ Denotational semantics: define the meaning of IMP commands as mathematical functions from stores to stores.
- ▶ Definitional translation: define the meaning of language constructs by translation to another language.
- ▶ A form of denotational semantics, but instead of the target language being mathematics, it is a simpler programming language.

- ▶ Definitional translation does not necessarily produce clean or efficient code
- ▶ Rather, it defines the meaning of the source language in terms of the target language.

# Evaluation Contexts

Recall the syntax and CBV operational semantics for the lambda calculus.

$$e ::= x \mid \lambda x. e \mid e_1 e_2$$
$$v ::= \lambda x. e$$

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$\frac{e \longrightarrow e'}{v e \longrightarrow v e'}$$

$$\beta\text{-REDUCTION} \frac{}{(\lambda x. e) v \longrightarrow e\{v/x\}}$$

# Evaluation Contexts

- ▶ Of the operational semantics rules, only the  $\beta$ -reduction rule told us how to “reduce” an expression
- ▶ The other two rules were simply telling us the order to evaluate expressions in

# Evaluation Contexts

The operational semantics of many of the languages we will consider have this feature: there are two kinds of rules, one kind specifying evaluation order, and the other kind specifying the “interesting” reductions.

# Evaluation Contexts

*Evaluation contexts* provide us with a mechanism to separate out these two kinds of rules.



An evaluation context  $E$  (sometimes written  $E[\cdot]$ ) is an expression with a “hole” in it, that is with a single occurrence of the special symbol  $[\cdot]$  (called the “hole”) in place of a subexpression.

The following grammar defines evaluation contexts for the pure CBV lambda calculus.

$$E ::= [\cdot] \mid E e \mid v E$$

We write  $E[e]$  to mean the evaluation context  $E$  where the hole has been replaced with the expression  $e$ .

$$E_1 = [\cdot] (\lambda x. x)$$

$$E_2 = (\lambda z. z z) [\cdot]$$

$$E_3 = ([\cdot] \lambda x. x x) ((\lambda y. y) (\lambda y. y))$$

$$E_1[\lambda y. y y] = (\lambda y. y y) \lambda x. x$$

$$E_2[\lambda x. \lambda y. x] = (\lambda z. z z) (\lambda x. \lambda y. x)$$

$$E_3[\lambda f. \lambda g. f g] = ((\lambda f. \lambda g. f g) \lambda x. x x) ((\lambda y. y) (\lambda y. y))$$

# Evaluation semantics for the pure CBV lambda calculus

$$\text{CTXT} \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

$$\beta\text{-REDUCTION} \frac{}{(\lambda x. e) v \longrightarrow e\{v/x\}}$$

# Evaluation semantics for the pure CBV lambda calculus

$$C_{\text{TXT}} \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

$$\beta\text{-REDUCTION} \frac{}{(\lambda x. e) v \longrightarrow e\{v/x\}}$$

Note that these ensure that we evaluate the left hand side of an application to a value, and then evaluate the right hand side of an application to a value before applying  $\beta$ -reduction.

$$E = [\cdot] (\lambda x. x)$$

- ▶ Here  $E[y]$  is a valid  $\lambda$ -calculus term, namely  $y (\lambda x. x)$ .
- ▶ But the evaluation gets stuck as neither,  $C_{TXT}$  nor  $\beta$ -REDUCTION can be applied.
- ▶ This is same as our previous definition of operational semantics.

$$e_0 = ((\lambda x. x + 30) (5 + 2)) + 5$$

In this example assume that we have integers and addition, and that the evaluation contexts are given by

$$E ::= [\cdot] \mid E e \mid v E \mid E + e \mid v + E.$$

$$e_0 = ((\lambda x. x + 30) (5 + 2)) + 5$$

$$C_{\text{TXT}} \frac{\overline{5 + 2 \rightarrow 7}}{((\lambda x. x + 30) (5 + 2)) + 5 \rightarrow ((\lambda x. x + 30) 7) + 5}$$



$$e_0 = ((\lambda x. x + 30) (5 + 2)) + 5$$

$$\text{C}_{\text{TXT}} \frac{\beta\text{-REDUCTION} \frac{(\lambda x. x + 30) 7 \longrightarrow 7 + 30}{((\lambda x. x + 30) 7) + 5 \longrightarrow (7 + 30) + 5}}{((\lambda x. x + 30) 7) + 5 \longrightarrow (7 + 30) + 5}$$

$$e_0 = ((\lambda x. x + 30) (5 + 2)) + 5$$

$$C_{\text{TXT}} \frac{\overline{7 + 30 \longrightarrow 37}}{(7 + 30) + 5 \longrightarrow 37 + 5}$$

$$e_0 = ((\lambda x. x + 30) (5 + 2)) + 5$$

---

$$37 + 5 \longrightarrow 42$$

We can also specify the operational semantics of CBN lambda calculus using evaluation contexts:

$$E ::= [\cdot] \mid E e$$

$$\text{C}_{\text{TXT}} \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

$$\beta\text{-REDUCTION} \frac{}{(\lambda x. e_1) e_2 \longrightarrow e_1 \{e_2/x\}}$$

# Multi-argument functions and currying

Our syntax for functions restricted us to functions that have a single argument:  $\lambda x. e$ . We could define a language that allows functions to have multiple arguments.

# Multi-argument functions and currying

$$e ::= x \mid \lambda x_1, \dots, x_n. e \mid e_0 e_1 \dots e_n$$

Here, a function  $\lambda x_1, \dots, x_n. e$  takes  $n$  arguments, with names  $x_1$  through  $x_n$ . In a multi argument application  $e_0 e_1 \dots e_n$ , we expect  $e_0$  to evaluate to an  $n$ -argument function, and  $e_1, \dots, e_n$  are the arguments that we will give the function.

We can define a CBV operational semantics for the multi-argument lambda calculus as follows.

$$E ::= [\cdot] \mid v_0 \dots v_{i-1} E e_{i+1} \dots e_n$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

$\beta$ -reduction

---


$$(\lambda x_1, \dots, x_n. e_0) v_1 \dots v_n \longrightarrow e_0 \{v_1/x_1\} \{v_2/x_2\} \dots \{v_n/x_n\}$$

The evaluation contexts ensure that we evaluate a multi-argument application  $e_0 e_1 \dots e_n$  by evaluating each expression from left to right down to a value.



- ▶ The multi-argument lambda calculus isn't any more expressive than the pure lambda calculus.
- ▶ Any multi-argument lambda calculus program can be translated into an equivalent pure lambda calculus program.

- ▶ We define a translation function  $\mathcal{T}[\cdot]$  that takes an expression in the multi-argument lambda calculus and returns an equivalent expression in the pure lambda calculus.
- ▶ That is, if  $e$  is a multi-argument lambda calculus expression,  $\mathcal{T}[e]$  is a pure lambda calculus expression.

# The translation function

$$\mathcal{T}[\llbracket x \rrbracket] = x$$

$$\mathcal{T}[\llbracket \lambda x_1, \dots, x_n. e \rrbracket] = \lambda x_1. \dots \lambda x_n. \mathcal{T}[\llbracket e \rrbracket]$$

$$\mathcal{T}[\llbracket e_0 e_1 e_2 \dots e_n \rrbracket] = (\dots ((\mathcal{T}[\llbracket e_0 \rrbracket] \mathcal{T}[\llbracket e_1 \rrbracket]) \mathcal{T}[\llbracket e_2 \rrbracket]) \dots \mathcal{T}[\llbracket e_n \rrbracket])$$

# Currying

The process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*.

# Currying

- ▶ Consider a function in  $A \times B \rightarrow C$ .
- ▶ Currying this function produces an element of  $A \rightarrow (B \rightarrow C)$ .
- ▶ The curried version of the function takes an argument from domain  $A$ , and returns a function that takes an argument from domain  $B$  and produces a result of domain  $C$ .

# Products and let

- ▶ A product is a pair of expressions  $(e_1, e_2)$ .
- ▶ If  $e_1$  and  $e_2$  are both values, then we regard the product as also being a value.
- ▶ Given a product, we can access the first or second element using the operators  $\#1$  and  $\#2$  respectively.

# Products and let

$$\#1 (v_1, v_2) \longrightarrow v_1$$

$$\#2 (v_1, v_2) \longrightarrow v_2$$

# Lambda calculus with products and let expressions

$$e ::= x \mid \lambda x. e \mid e_1 e_2 \\ \mid (e_1, e_2) \mid \#1 e \mid \#2 e \\ \mid \text{let } x = e_1 \text{ in } e_2$$
$$v ::= \lambda x. e \mid (v_1, v_2)$$



# Lambda calculus with products and let expressions

$$\begin{aligned} e ::= & x \mid \lambda x. e \mid e_1 e_2 \\ & \mid (e_1, e_2) \mid \#1 e \mid \#2 e \\ & \mid \text{let } x = e_1 \text{ in } e_2 \end{aligned}$$

$$v ::= \lambda x. e \mid (v_1, v_2)$$

In this language, values are either functions or pairs of values.

We define a small-step CBV operational semantics for the language using evaluation contexts.

$$E ::= [\cdot] \mid E e \mid v E \mid (E, e) \mid (v, E)$$

$$\mid \#1 E$$

$$\mid \#2 E$$

$$\mid \text{let } x = E \text{ in } e_2$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

$$\beta\text{-REDUCTION} \frac{}{(\lambda x. e) v \longrightarrow e\{v/x\}}$$

$$\frac{}{\#1 (v_1, v_2) \longrightarrow v_1}$$

$$\frac{}{\#2 (v_1, v_2) \longrightarrow v_2}$$

$$\frac{}{\text{let } x = v \text{ in } e \longrightarrow e\{v/x\}}$$

# Lambda calculus with products and let to the pure lambda calculus

$$\mathcal{T}[[x]] = x$$

$$\mathcal{T}[[\lambda x. e]] = \lambda x. \mathcal{T}[[e]]$$

$$\mathcal{T}[[e_1 e_2]] = \mathcal{T}[[e_1]] \mathcal{T}[[e_2]]$$

$$\mathcal{T}[[e_1, e_2]] = (\lambda x. \lambda y. \lambda f. f x y) \mathcal{T}[[e_1]] \mathcal{T}[[e_2]]$$

$$\mathcal{T}[[\#1 e]] = \mathcal{T}[[e]] (\lambda x. \lambda y. x)$$

$$\mathcal{T}[[\#2 e]] = \mathcal{T}[[e]] (\lambda x. \lambda y. y)$$

$$\mathcal{T}[[\text{let } x = e_1 \text{ in } e_2]] = (\lambda x. \mathcal{T}[[e_2]]) \mathcal{T}[[e_1]]$$

- ▶ We can give an equivalent semantics by translation to the pure CBV lambda calculus.
- ▶ We encode a pair  $(e_1, e_2)$  as a value that takes a function  $f$ , and applies  $f$  to  $v_1$  and  $v_2$ , where  $v_1$  and  $v_2$  are the result of evaluating  $e_1$  and  $e_2$  respectively.

- ▶ The projection operators pass a function to the encoding of pairs that selects either the first or second element as appropriate.
- ▶ The expression  $\text{let } x = e_1 \text{ in } e_2$  is equivalent to the application  $(\lambda x. e_2) e_1$ .

# CBN to CBV

We can translate a call-by-name program into a call-by-value program.

# CBN to CBV

- ▶ In CBV, arguments to functions are evaluated before the function is applied
- ▶ In CBN, functions are applied as soon as possible.
- ▶ In the translation, we delay the evaluation of arguments by wrapping them in a function.



## CBN to CBV

- ▶ This is called a *thunk*: wrapping a computation in a function to delay its evaluation.
- ▶ Since arguments to functions are turned into thunks, when we want to use an argument in a function body, we need to evaluate the thunk.
- ▶ We do so by applying the thunk (which is simply a function)
- ▶ It doesn't matter what we apply the thunk to, since the thunk's argument is never used.

# CBN to CBV

$$\mathcal{T}[[x]] = x (\lambda y. y)$$

$$\mathcal{T}[[\lambda x. e]] = \lambda x. \mathcal{T}[[e]]$$

$$\mathcal{T}[[e_1 e_2]] = \mathcal{T}[[e_1]] (\lambda z. \mathcal{T}[[e_2]])$$

where  $z$  is not a free variable of  $e_2$

# CBV to CBN

It may be worth thinking about translation in the opposite direction i.e. CBV to CBN. One approach is to use *continuations* which will be introduced in the next lecture.

# Adequacy of translation

We would like the translation to be correct, that is, to preserve the meaning of source programs.

# Adequacy of translation

We would like an expression  $e$  in the source language to evaluate to a value  $v$  if and only if the translation of  $e$  evaluates to a value  $v'$  such that  $v'$  is “equal to”  $v$ .

# Adequacy of translation

There are two criteria for a translation to be *adequate*: soundness and completeness.

Suppose  $\mathbf{Exp}_{\text{src}}$  is the set of source language expressions, and that  $\longrightarrow_{\text{src}}$  and  $\longrightarrow_{\text{trg}}$  are the evaluation relations for the source and target languages respectively.

# Soundness

A translation is *sound* if every target evaluation represents a source evaluation:

$$\forall e \in \mathbf{Exp}_{\text{src}}. \text{ if } \mathcal{T}[[e]] \longrightarrow_{\text{trg}}^* v' \text{ then } \exists v. e \longrightarrow_{\text{src}}^* v \\ \text{ and } v' \text{ equivalent to } v$$



# Completeness

A translation is *complete* if every source evaluation has a target evaluation.

$\forall e \in \mathbf{Exp}_{\text{src}}.$  if  $e \longrightarrow_{\text{src}}^* v$  then  $\exists v'. \mathcal{T}[[e]] \longrightarrow_{\text{trg}}^* v'$   
and  $v'$  equivalent to  $v$