Lecture 17
Definitional Translation & Continuations
Definitional Translation

We know how to encode Booleans, conditionals, natural numbers, and recursion in $\lambda$-calculus.

Can we define a *real* programming language by translating everything in it into the $\lambda$-calculus?
Definitional Translation

We know how to encode Booleans, conditionals, natural numbers, and recursion in $\lambda$-calculus.

Can we define a real programming language by translating everything in it into the $\lambda$-calculus?

In definitional translation, we define a denotational semantics where the target is a simpler programming language instead of mathematical objects.
Multi-Argument $\lambda$-calculus

Let’s define a version of the $\lambda$-calculus that allows functions to take multiple arguments.

$$e ::= x \mid \lambda x_1, \ldots, x_n. e \mid e_0 e_1 \ldots e_n$$
Multi-Argument $\lambda$-calculus

We can define a CBV operational semantics:

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

$$e \rightarrow e'$$

$$\frac{\text{CONTEXT}}{E[e] \rightarrow E[e']}$$

$$\frac{(\lambda x_1, \ldots, x_n. \ e_0) \ v_1 \ldots v_n \rightarrow (e_0 \{v_1/x_1\} \{v_2/x_2\}) \ldots \{v_n/x_n\}}{\beta}$$

The evaluation contexts ensure that we evaluate multi-argument applications $e_0 \ e_1 \ldots \ e_n$ from left to right.
Definitional Translation

The multi-argument $\lambda$-calculus isn’t any more expressive than the pure $\lambda$-calculus.
Definitional Translation

The multi-argument λ-calculus isn’t any more expressive that the pure λ-calculus.

We can define a translation $\mathcal{T}[\cdot]$ that takes an expression in the multi-argument λ-calculus and returns an equivalent expression in the pure λ-calculus.
Definitional Translation

The multi-argument $\lambda$-calculus isn’t any more expressive than the pure $\lambda$-calculus.

We can define a translation $\mathcal{T}[[\cdot]]$ that takes an expression in the multi-argument $\lambda$-calculus and returns an equivalent expression in the pure $\lambda$-calculus.

$$
\begin{align*}
\mathcal{T}[[x]] & \triangleq x \\
\mathcal{T}[[\lambda x_1, \ldots, x_n. e]] & \triangleq \lambda x_1. \ldots \lambda x_n. \mathcal{T}[[e]] \\
\mathcal{T}[[e_0 e_1 e_2 \ldots e_n]] & \triangleq (\ldots ((\mathcal{T}[[e_0]] \mathcal{T}[[e_1]]) \mathcal{T}[[e_2]]) \ldots \mathcal{T}[[e_n]])
\end{align*}
$$

This translation curries the multi-argument $\lambda$-calculus.
Products (Pairs) and Let

Syntax

\[ e ::= x \]
\[ \ | \lambda x. e \]
\[ \ | e_1 e_2 \]
\[ \ | (e_1, e_2) \]
\[ \ | #1 e \]
\[ \ | #2 e \]
\[ \ | \text{let } x = e_1 \text{ in } e_2 \]

\[ v ::= \lambda x. e \]
\[ \ | (v_1, v_2) \]
Evaluation Contexts

\[ E ::= [\cdot] \]
\[ \mid E \, e \]
\[ \mid \nu \, E \]
\[ \mid (E, \, e) \]
\[ \mid (\nu, \, E) \]
\[ \mid \#1 \, E \]
\[ \mid \#2 \, E \]
\[ \mid \text{let } x = E \text{ in } e_2 \]
Products (Pairs) and Let

Semantics

\[
e \rightarrow e' \\
\frac{e \rightarrow e'}{E[e] \rightarrow E[e']}
\]

\[
\beta \\
(\lambda x. e) v \rightarrow e\{v/x\}
\]

\[
\#1 \ (v_1, v_2) \rightarrow v_1 \\
\#2 \ (v_1, v_2) \rightarrow v_2
\]

\[
\text{let } x = v \text{ in } e \rightarrow e\{v/x\}
\]
Products (Pairs) and Let

Translation

\[
\begin{align*}
\mathcal{T}[x] &= x \\
\mathcal{T}[\lambda x. e] &= \lambda x. \mathcal{T}[e] \\
\mathcal{T}[e_1 e_2] &= \mathcal{T}[e_1] \mathcal{T}[e_2] \\
\mathcal{T}[(e_1, e_2)] &= (\lambda x. \lambda y. \lambda f. f x y) \mathcal{T}[e_1] \mathcal{T}[e_2] \\
\mathcal{T}[\#1 e] &= \mathcal{T}[e] (\lambda x. \lambda y. x) \\
\mathcal{T}[\#2 e] &= \mathcal{T}[e] (\lambda x. \lambda y. y) \\
\mathcal{T}[\text{let } x = e_1 \text{ in } e_2] &= (\lambda x. \mathcal{T}[e_2]) \mathcal{T}[e_1]
\end{align*}
\]
Laziness

Consider the call-by-name \( \lambda \)-calculus...

Syntax

\[
e ::= x \\
| e_1 e_2 \\
| \lambda x. e
\]

\[
\nu ::= \lambda x. e
\]

Semantics

\[
\frac{e_1 \rightarrow e'_1}{e_1 e_2 \rightarrow e'_1 e_2}
\]

\[
(\lambda x. e_1) e_2 \rightarrow e_1\{e_2/x\} \quad \beta
\]
Laziness

Translation

\[ T[x] = x (\lambda y. y) \]
\[ T[\lambda x. e] = \lambda x. T[e] \]
\[ T[e_1 e_2] = T[e_1] (\lambda z. T[e_2]) \quad \text{z is not a free variable of } e_2 \]
Syntax

\[
e ::= x \\
\quad | \lambda x. e \\
\quad | e_0 \ e_1
\]

\[
v ::= \lambda x. e
\]
Syntax

\[ e ::= x \]
\[ \quad \mid \lambda x. e \]
\[ \quad \mid e_0 e_1 \]
\[ \quad \mid \text{ref } e \]

\[ \nu ::= \lambda x. e \]
Syntax

\[ e ::= x \]
\[ \quad | \ \lambda x. \ e \]
\[ \quad | \ e_0 \ e_1 \]
\[ \quad | \ \text{ref} \ e \]
\[ \quad | \ !e \]

\[ \nu ::= \lambda x. \ e \]
References

Syntax

\[ e ::= x \]
\[ \quad | \lambda x. e \]
\[ \quad | e_0 e_1 \]
\[ \quad | \text{ref } e \]
\[ \quad | !e \]
\[ \quad | e_1 ::= e_2 \]

\[ \nu ::= \lambda x. e \]
\[ e ::= x \]
\[ \quad | \lambda x. e \]
\[ \quad | e_0 e_1 \]
\[ \quad | \text{ref } e \]
\[ \quad | !e \]
\[ \quad | e_1 ::= e_2 \]
\[ \quad | \ell \]

\[ \nu ::= \lambda x. e \]
Syntax

e ::= x
    | λx. e
    | e_0 e_1
    | ref e
    | !e
    | e_1 := e_2
    | ℓ

ν ::= λx. e
    | ℓ
References

Evaluation Contexts

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

Evaluation Contexts

\[ E ::= [\cdot] \]
\[ \mid E \, e \]
\[ \mid \nu \, E \]
\[ \mid \text{ref} \, E \]
References

Evaluation Contexts

\[ E ::= [\cdot] \]

\[ E e \]

\[ \nu E \]

\[ \text{ref } E \]

\[ !E \]
Evaluation Contexts

\[ E ::= [\cdot] \\
| E \ e \\
| \nu \ E \\
| \text{ref} \ E \\
| !E \\
| E ::= e \]
Evaluation Contexts

\[ E ::= [\cdot] \]
\[ \mid E \ e \]
\[ \mid \nu E \]
\[ \mid \text{ref} \ E \]
\[ \mid \! E \]
\[ \mid E ::= e \]
\[ \mid \nu ::= E \]
Semantics

\[ \sigma : \text{Loc} \rightarrow \text{Val} \]

\[
\begin{align*}
\langle \sigma, e \rangle & \rightarrow \langle \sigma', e' \rangle \\
\langle \sigma, E[e] \rangle & \rightarrow \langle \sigma', E[e'] \rangle
\end{align*}
\]

\[
\langle \sigma, (\lambda x. e) \, v \rangle \rightarrow \langle \sigma, e\{v/x\} \rangle
\]

\[
\frac{\ell \not\in \text{dom}(\sigma)}{\langle \sigma, \text{ref} \, v \rangle \rightarrow \langle \sigma[\ell \mapsto v], \ell \rangle}
\]

\[
\frac{\sigma(\ell) = v}{\langle \sigma, !\ell \rangle \rightarrow \langle \sigma, v \rangle}
\]

\[
\langle \sigma, \ell := v \rangle \rightarrow \langle \sigma[\ell \mapsto v], v \rangle
\]
References

Translation

...left as an exercise to the reader. ;-)
Adequacy

How do we know if a translation is correct?
Adequacy

How do we know if a translation is correct?

Every target evaluation should represent a source evaluation...

**Definition (Soundness)**

\[ \forall e \in \text{Exp}_{\text{src}} \text{. if } T[e] \rightarrow_{\text{trg}}^* \nu' \text{ then } \exists \nu. e \rightarrow_{\text{src}}^* \nu \]

and \( \nu' \) equivalent to \( \nu \)
Adequacy

How do we know if a translation is correct?

Every target evaluation should represent a source evaluation...

**Definition (Soundness)**

\[ \forall e \in \text{Exp}_{\text{src}}. \text{if } \mathcal{T}[e] \rightarrow^{*}_{\text{trg}} v' \text{ then } \exists v. e \rightarrow^{*}_{\text{src}} v \]

and \( v' \) equivalent to \( v \)

...and every source evaluation should have a target evaluation:

**Definition (Completeness)**

\[ \forall e \in \text{Exp}_{\text{src}}. \text{if } e \rightarrow^{*}_{\text{src}} v \text{ then } \exists v'. \mathcal{T}[e] \rightarrow^{*}_{\text{trg}} v' \]

and \( v' \) equivalent to \( v \)
Continuations

In the preceding translations, the control structure of the source language was translated directly into the corresponding control structure in the target language.

For example:

$$\mathcal{T}[\lambda x. e] = \lambda x. \mathcal{T}[e]$$
$$\mathcal{T}[e_1 e_2] = \mathcal{T}[e_1] \mathcal{T}[e_2]$$

What can go wrong with this approach?
Continuations

- A snippet of code that represents “the rest of the program”
- Can be used directly by programmers...
- ...or in program transformations by a compiler
- Make the control flow of the program explicit
- Also useful for defining the meaning of features like exceptions
Example

Consider the following expression:

$$(\lambda x. x) \ ((1 + 2) + 3) + 4$$
Example

Consider the following expression:

\[(\lambda x. x) ((1 + 2) + 3) + 4\]

If we make all of the continuations explicit, we obtain:

\[k_0 = \lambda v. (\lambda x. x) v\]
Example

Consider the following expression:

$$(\lambda x. x) \left( (1 + 2) + 3 \right) + 4$$

If we make all of the continuations explicit, we obtain:

$$k_0 = \lambda v. (\lambda x. x) \, v$$
$$k_1 = \lambda a. k_0 \, (a + 4)$$
Example

Consider the following expression:

$$(\lambda x. x) \left( (1 + 2) + 3 \right) + 4$$

If we make all of the continuations explicit, we obtain:

$$k_0 = \lambda v. (\lambda x. x) \, v$$
$$k_1 = \lambda a. k_0 \, (a + 4)$$
$$k_2 = \lambda b. k_1 \, (b + 3)$$
Example

Consider the following expression:

$$(\lambda x . x) ((1 + 2) + 3) + 4$$

If we make all of the continuations explicit, we obtain:

$$k_0 = \lambda v . (\lambda x . x) \, v$$
$$k_1 = \lambda a . k_0 \, (a + 4)$$
$$k_2 = \lambda b . k_1 \, (b + 3)$$
$$k_3 = \lambda c . k_2 \, (c + 2)$$
Example

Consider the following expression:

\[(\lambda x. x) ((1 + 2) + 3) + 4\]

If we make all of the continuations explicit, we obtain:

\[k_0 = \lambda v. (\lambda x. x) v\]
\[k_1 = \lambda a. k_0 (a + 4)\]
\[k_2 = \lambda b. k_1 (b + 3)\]
\[k_3 = \lambda c. k_2 (c + 2)\]

The original expression is equivalent to \(k_3 1\), or:

\[(\lambda c. (\lambda b. (\lambda a. (\lambda v. (\lambda x. x) v) (a + 4)) (b + 3)) (c + 2)) 1\]
Example (Continued)

Recall that let $x = e$ in $e'$ is syntactic sugar for $(\lambda x. e') e$.

Hence, we can rewrite the expression with continuations more succinctly as

\[
\begin{align*}
\text{let } c &= 1 \text{ in } \\
\text{let } b &= c + 2 \text{ in } \\
\text{let } a &= b + 3 \text{ in } \\
\text{let } \nu &= a + 4 \text{ in } \\
(\lambda x. x) \nu
\end{align*}
\]
CPS Transformation

We write $CPS[e] \ k = \ldots$ instead of $CPS[e] = \lambda k. \ldots$

We assume that the new variables introduced are “fresh.”
CPS Transformation

\[ CPS[n] k = k n \]

We write \( CPS[e] k = \ldots \) instead of \( CPS[e] = \lambda k. \ldots \)

We assume that the new variables introduced are “fresh.”
CPS Transformation

\[ CPS[n] k = kn \]
\[ CPS[e_1 + e_2] k = CPS[e_1] (\lambda n. CPS[e_2] (\lambda m. k (n + m))) \]

We write \( CPS[e] k = \ldots \) instead of \( CPS[e] = \lambda k. \ldots \)

We assume that the new variables introduced are “fresh.”
CPS Transformation

\[
\text{CPS}[n] \, k = k \, n \\
\text{CPS}[e_1 + e_2] \, k = \text{CPS}[e_1] \, (\lambda n. \text{CPS}[e_2] \, (\lambda m. k \, (n + m))) \\
\text{CPS}[(e_1, e_2)] \, k = \text{CPS}[e_1] \, (\lambda v. \text{CPS}[e_2] \, (\lambda w. k \, (v, w)))
\]

We write \(\text{CPS}[e] \, k = \ldots\) instead of \(\text{CPS}[e] = \lambda k. \ldots\)

We assume that the new variables introduced are “fresh.”
CPS Transformation

\[ \text{CPS}[n] k = kn \]
\[ \text{CPS}[e_1 + e_2] k = \text{CPS}[e_1] (\lambda n. \text{CPS}[e_2] (\lambda m. k (n + m))) \]
\[ \text{CPS}[(e_1, e_2)] k = \text{CPS}[e_1] (\lambda v. \text{CPS}[e_2] (\lambda w. k (v, w))) \]
\[ \text{CPS}[\#1 e] k = \text{CPS}[e] (\lambda v. k (#1 v)) \]

We write \( \text{CPS}[e] k = \ldots \) instead of \( \text{CPS}[e] = \lambda k. \ldots \)

We assume that the new variables introduced are “fresh.”
CPS Transformation

\[ CPS[n] k = kn \]
\[ CPS[e_1 + e_2] k = CPS[e_1] (\lambda n. CPS[e_2] (\lambda m. k (n + m))) \]
\[ CPS[(e_1, e_2)] k = CPS[e_1] (\lambda v. CPS[e_2] (\lambda w. k (v, w))) \]
\[ CPS[\#1 e] k = CPS[e] (\lambda v. k (\#1 v)) \]
\[ CPS[\#2 e] k = CPS[e] (\lambda v. k (\#2 v)) \]

We write \( CPS[e] k = \ldots \) instead of \( CPS[e] = \lambda k. \ldots \)

We assume that the new variables introduced are “fresh.”
CPS Transformation

\[ \text{CPS}[n] k = k \times n \]
\[ \text{CPS}[e_1 + e_2] k = \text{CPS}[e_1] (\lambda n. \text{CPS}[e_2] (\lambda m. k (n + m))) \]
\[ \text{CPS}[(e_1, e_2)] k = \text{CPS}[e_1] (\lambda v. \text{CPS}[e_2] (\lambda w. k (v, w))) \]
\[ \text{CPS}[\#1 e] k = \text{CPS}[e] (\lambda v. k (\#1 v)) \]
\[ \text{CPS}[\#2 e] k = \text{CPS}[e] (\lambda v. k (\#2 v)) \]
\[ \text{CPS}[x] k = k \times x \]

We write \( \text{CPS}[e] k = \ldots \) instead of \( \text{CPS}[e] = \lambda k. \ldots \)

We assume that the new variables introduced are “fresh.”
CPS Transformation

\[
\begin{align*}
\text{CPS}[n] k &= k n \\
\text{CPS}[e_1 + e_2] k &= \text{CPS}[e_1] (\lambda n. \text{CPS}[e_2] (\lambda m. k (n + m))) \\
\text{CPS}[(e_1, e_2)] k &= \text{CPS}[e_1] (\lambda v. \text{CPS}[e_2] (\lambda w. k (v, w))) \\
\text{CPS}[#1/e] k &= \text{CPS}[e] (\lambda v. k (#1 v)) \\
\text{CPS}[#2/e] k &= \text{CPS}[e] (\lambda v. k (#2 v)) \\
\text{CPS}[x] k &= k x \\
\text{CPS}[(\lambda x. e)] k &= k (\lambda x. \lambda k'. \text{CPS}[e] k')
\end{align*}
\]

We write \( \text{CPS}[e] k = \ldots \) instead of \( \text{CPS}[e] = \lambda k. \ldots \)

We assume that the new variables introduced are “fresh.”
CPS Transformation

\[
\text{CPS}[n] k = kn
\]
\[
\text{CPS}[e_1 + e_2] k = \text{CPS}[e_1] (\lambda n. \text{CPS}[e_2] (\lambda m. k (n + m)))
\]
\[
\text{CPS}[(e_1, e_2)] k = \text{CPS}[e_1] (\lambda v. \text{CPS}[e_2] (\lambda w. k (v, w)))
\]
\[
\text{CPS}[\#1 e] k = \text{CPS}[e] (\lambda v. k (\#1 v))
\]
\[
\text{CPS}[\#2 e] k = \text{CPS}[e] (\lambda v. k (\#2 v))
\]
\[
\text{CPS}[x] k = kx
\]
\[
\text{CPS}[\lambda x. e] k = k (\lambda x. \lambda k'. \text{CPS}[e] k')
\]
\[
\text{CPS}[e_1 e_2] k = \text{CPS}[e_1] (\lambda f. \text{CPS}[e_2] (\lambda v. f v k))
\]

We write \(\text{CPS}[e] k = \ldots\) instead of \(\text{CPS}[e] = \lambda k. \ldots\)

We assume that the new variables introduced are “fresh.”