References and Continuations

CS 1520 (Spring 2025)

Harvard University

Thursday, February 27, 2025

Today, we will learn about

- References
- Continuations
- CPS translation

References

- We introduce constructs for creating, reading, and updating memory locations, also called *references*.
- The resulting language is still a functional language (since functions are first-class values), but expressions can have side-effects, that is, they can modify state.

References: syntax

$$e ::= x \mid \lambda x.\, e \mid e_0\ e_1 \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid \ell$$

$$v ::= \lambda x.\, e \mid \ell$$

References: syntax

$$e ::= x \mid \lambda x.\, e \mid e_0\ e_1 \mid \mathsf{ref}\ e \mid\ !e \mid e_1 := e_2 \mid \ell$$
$$v ::= \lambda x.\, e \mid \ell$$

- ref e creates a new memory location (like a malloc), and sets the initial contents of the location to (the result of) e.
- The expression ref e itself evaluates to a memory location ℓ.

References: syntax

$$
\begin{aligned}
e &::= x \mid \lambda x.\, e \mid e_0\ e_1 \mid \mathsf{ref}\ e \mid !e \mid e_1 := e_2 \mid \ell \\
v &::= \lambda x.\, e \mid \ell
\end{aligned}
$$

- The expression $!e$ assumes that e evaluates to a memory location, and $!e$ evaluates to the current contents of the memory location.
- Expression $e_1 := e_2$ assumes that e_1 evaluates to a memory location ℓ, and updates the contents of ℓ with (the result of) e_2.

References

- Locations ℓ are not part of the *surface syntax* of the language, the syntax that a programmer would write.
- They are introduced only by the operational semantics.

References: small-step CBV operational semantics.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid \text{ref}\ E \mid !E \mid E := e \mid v := E$$

$$\frac{< e, \sigma > \longrightarrow < e', \sigma' >}{< E[e], \sigma > \longrightarrow < E[e'], \sigma' >}$$

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

References: small-step CBV operational semantics.

$$\textsc{Alloc}\ \frac{}{<\mathsf{ref}\ v, \sigma > \longrightarrow < \ell, \sigma[\ell \mapsto v] >}\ \ell \notin \mathsf{dom}(\sigma)$$

$$\textsc{Deref}\ \frac{}{< !\ell, \sigma > \longrightarrow < v, \sigma >}\ \sigma(\ell) = v$$

$$\textsc{Assign}\ \frac{}{< \ell := v, \sigma > \longrightarrow < v, \sigma[\ell \mapsto v] >}$$

References do not add any expressive power to the lambda calculus

References do not add any expressive power to the lambda calculus

It is possible to translate lambda calculus with references to the pure lambda calculus.

Continuations

So far we have seen a number of language features that extend lambda calculus, and have translated many of these into the pure lambda calculus:

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

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

Continuations

- This style of translation works well when the source language is similar to the target language.
- However, when the control structures of the source and target languages differ considerably, it doesn't work as well.

Continuations

Continuations are a programming technique that may be used directly by a programmer, or used in program transformations by a compiler.

Continuations

Intuitively, a continuation represents "the rest of the program."

if foo $<$ 10 then 32 $+$ 6 else 7 $+$ bar

Consider the evaluation of the expression foo $<$ 10.

if foo < 10 then $32 + 6$ else $7 +$ bar

When we finish evaluating foo < 10, we will evaluate the if statement, and then evaluate the appropriate branch.

if foo $<$ 10 then 32 + 6 else 7 + bar

The *continuation* of the subexpression foo $<$ 10 is the rest of the computation that will occur after we evaluate the subexpression.

if foo $<$ 10 then 32 $+$ 6 else 7 $+$ bar

We can write this continuation as a function that takes the result of the subexpression:

$$(\lambda y.\, \text{if } y \text{ then } 32 + 6 \text{ else } 7 + \text{bar})\ (\text{foo} < 10)$$

if foo $<$ 10 then 32 $+$ 6 else 7 $+$ bar

$$(\lambda y.\, \text{if } y \text{ then } 32 + 6 \text{ else } 7 + \text{bar})\ (\text{foo} < 10)$$

The evaluation order and result remain the same, we just extracted the continuation of the subexpression in to a function.

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

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

We start by defining a continuation for the outermost evaluation context, which takes a value, and applies the identity function to it.

$$k_0 = \lambda v.\, (\lambda x.\, x)\ v$$

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

The evaluation context that is evaluated next-to-last takes a value, adds 4 to it, and then passes the result to k_0.

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

Likewise, for the next evaluation contexts.

$$k_2 = \lambda b.\, k_1\ (b+3)$$
$$k_3 = \lambda c.\, k_2\ (c+2)$$

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

$$\begin{aligned}
k_0 &= \lambda v.\, (\lambda x.\, x)\ v \\
k_1 &= \lambda a.\, k_0\ (a + 4) \\
k_2 &= \lambda b.\, k_1\ (b + 3) \\
k_3 &= \lambda c.\, k_2\ (c + 2)
\end{aligned}$$

The program itself is now equivalent to $k_3\ 1$. We can rewrite the above as

$$\begin{aligned}
&\text{let } c = 1 \text{ in} \\
&\text{let } b = c + 2 \text{ in} \\
&\text{let } a = b + 3 \text{ in} \\
&\text{let } v = a + 4 \text{ in} \\
&(\lambda x.\, x)\ v
\end{aligned}$$

This is fairly close to some machine instructions of the form:

$$
\begin{array}{l}
\text{set } c, 1 \\
\text{add } b, c, 2 \\
\text{add } a, b, 3 \\
\text{add } v, a, 4 \\
\text{call id}, v
\end{array}
$$

Using continuations, functions can be transformed into “functions that don’t return”—functions that take, besides the usual arguments, an additional argument representing a continuation.

CPS

When the function finishes, it invokes the continuation on its result, instead of returning the result to its caller. Writing functions in this way is usually referred to as *Continuation-Passing Style*.

CPS version of factorial

$$\begin{aligned} FACT_{cps} = Y\ \lambda f.\, \lambda n, k.\ & \\ & \text{if } n = 0 \text{ then } k\ 1 \text{ else } f\ (n-1)\ (\lambda v.\, k\ (n * v)) \end{aligned}$$

CPS translation

- We can translate lambda calculus programs into continuation-passing style.
- We define a translation function $\mathcal{CPS}[\![\cdot]\!]$
- It takes a CBV lambda calculus expression, and translates the expression to a CBV lambda calculus expression in continuation-passing style.

From lambda calculus with pairs to CPS

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e$$

From lambda calculus with pairs to CPS

The translation $\mathcal{CPS}[\![e]\!]$ will produce a function whose argument is the continuation to which to pass the result.

That is, for all expressions e, the translation is of the form $\mathcal{CPS}[\![e]\!] = \lambda k. \ldots$, where k is a continuation.

From lambda calculus with pairs to CPS

We will both assume and guarantee that for any expression *e*, the translation $\mathcal{CPS}[\![e]\!] = \lambda k.\ \ldots$ will apply *k* to the result of evaluating *e*.

From lambda calculus with pairs to CPS

$$\mathcal{CPS}[\![n]\!]k = k\ n$$

$$\mathcal{CPS}[\![e_1 + e_2]\!]k = \mathcal{CPS}[\![e_1]\!]\ (\lambda n. \mathcal{CPS}[\![e_2]\!]\ (\lambda m.\, k\ (n + m)))$$

n is not a free variable of e_2

$$\mathcal{CPS}[\![(e_1, e_2)]\!]k = \mathcal{CPS}[\![e_1]\!]\ (\lambda v. \mathcal{CPS}[\![e_2]\!]\ (\lambda w.\, k\ (v, w)))$$

v is not a free variable of e_2

From lambda calculus with pairs to CPS

$$\mathcal{CPS}[\![\#1\ e]\!]k = \mathcal{CPS}[\![e]\!]\ (\lambda v.\, k\ (\#1\ v))$$

$$\mathcal{CPS}[\![\#2\ e]\!]k = \mathcal{CPS}[\![e]\!]\ (\lambda v.\, k\ (\#2\ v))$$

$$\mathcal{CPS}[\![x]\!]k = k\ x$$

$$\mathcal{CPS}[\![\lambda x.\, e]\!]k = k\ (\lambda x, k'.\, \mathcal{CPS}[\![e]\!]k')$$

k' is not a free variable of e

$$\mathcal{CPS}[\![e_1\ e_2]\!]k = \mathcal{CPS}[\![e_1]\!]\ (\lambda f.\, \mathcal{CPS}[\![e_2]\!]\ (\lambda v.\, f\ v\ k))$$

f is not a free variable of e_2

Example: $\mathcal{CPS}[\![(\lambda a.\, a + 6)\ 7]\!] ID$

$$= \mathcal{CPS}[\![(\lambda a.\, a + 6)]\!]\ \ (\lambda f.\, \mathcal{CPS}[\![7]\!]\ (\lambda v.\, f\ \ v\ \ ID))$$

$$= (\lambda f.\, \mathcal{CPS}[\![7]\!]\ (\lambda v.\, f\ \ v\ \ ID))\ (\lambda a, k'.\, \mathcal{CPS}[\![a + 6]\!] k')$$

$$= (\lambda f.\, (\lambda v.\, f\ \ v\ \ ID)\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a + 6]\!] k')$$

$$= (\lambda f.\, (\lambda v.\, f\ \ v\ \ ID)\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a]\!] \\ (\lambda n.\, \mathcal{CPS}[\![6]\!]\ (\lambda m.\, k'\ \ (m + n))))$$

Example: $\mathcal{CPS}[\![(\lambda a.\, a + 6)\ 7]\!]\mathit{ID}$

$$= (\lambda f.\, (\lambda v.\, f\ \ v\ \ \mathit{ID})\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a]\!] \\ (\lambda n.\, \mathcal{CPS}[\![6]\!]\ (\lambda m.\, k'\ (m + n))))$$

$$= (\lambda f.\, (\lambda v.\, f\ \ v\ \ \mathit{ID})\ 7)\ (\lambda a, k'.\, \mathcal{CPS}[\![a]\!]\ (\lambda n.\, (\lambda m.\, k'\ (m + n))\ 6))$$

$$= (\lambda f.\, (\lambda v.\, f\ \ v\ \ \mathit{ID})\ 7)\ (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m + n))\ 6)\ a)$$

Example: $\mathcal{CPS}[\![(\lambda a.\, a + 6)\ 7]\!]\mathit{ID}$

$$
\begin{aligned}
& (\lambda f.\, (\lambda v.\, f\ \ v\ \ \mathit{ID})\ 7)\ (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m + n))\ 6)\ a) \\
& \longrightarrow (\lambda v.\, (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m + n))\ 6)\ a)\ v\ \ \mathit{ID})\ 7 \\
& \longrightarrow (\lambda a, k'.\, (\lambda n.\, (\lambda m.\, k'\ (m + n))\ 6)\ a)\ 7\ \mathit{ID} \\
& \longrightarrow (\lambda n.\, (\lambda m.\, \mathit{ID}\ (m + n))\ 6)\ 7 \\
& \longrightarrow (\lambda m.\, \mathit{ID}\ (m + 7))\ 6 \\
& \longrightarrow \mathit{ID}\ (6 + 7) \\
& \longrightarrow \mathit{ID}\ 13 \\
& \longrightarrow 13
\end{aligned}
$$