CS 4110 – Programming Languages and Logics
Lecture #18: More Definitional Translation and Continuations

In the last lecture we introduced a general framework for defining language features by translation. This lecture presents several additional example translations (for products, let-expressions, and laziness, and mutable references); discusses correctness; and introduces continuations.

0.1 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. (That is, we cannot further evaluate a product if both elements are values.) Given a product, we can access the first or second element using the operators $\#1$ and $\#2$ respectively. That is, $\#1\ (v_1, v_2) \to v_1$ and $\#2\ (v_1, v_2) \to v_2$. (Other common notation for projection includes π_1 and π_2, and fst and snd.)

The syntax of λ-calculus extended with products and let expressions is defined as follows.

$$\begin{aligned} e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \\ &\quad \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e \\ &\quad \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \\ v &::= \lambda x.\, e \mid (v_1, v_2) \end{aligned}$$

Note that values in this language are either functions or pairs of values.

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

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid \#1\ E \mid \#2\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ e_2$$

$$\frac{e \to e'}{E[e] \to E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \to e\{v/x\}}$$

$$\frac{}{\#1\ (v_1, v_2) \to v_1} \qquad \frac{}{\#2\ (v_1, v_2) \to v_2}$$

$$\frac{}{\mathsf{let}\ x = v\ \mathsf{in}\ e \to e\{v/x\}}$$

Next, we define an equivalent semantics by translation to the pure CBV λ-calculus.

$$
\begin{aligned}
\mathcal{T}[\![x]\!] &= x \\
\mathcal{T}[\![\lambda x.\, e]\!] &= \lambda x.\, \mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_1\ e_2]\!] &= \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![(e_1, e_2)]\!] &= (\lambda x.\, \lambda y.\, \lambda f.\, f\ x\ y)\ \mathcal{T}[\![e_1]\!]\ \mathcal{T}[\![e_2]\!] \\
\mathcal{T}[\![\#1\ e]\!] &= \mathcal{T}[\![e]\!]\ (\lambda x.\, \lambda y.\, x) \\
\mathcal{T}[\![\#2\ e]\!] &= \mathcal{T}[\![e]\!]\ (\lambda x.\, \lambda y.\, y) \\
\mathcal{T}[\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!] &= (\lambda x.\, \mathcal{T}[\![e_2]\!])\ \mathcal{T}[\![e_1]\!]
\end{aligned}
$$

Note that 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. Also note that the expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$ is equivalent to the application $(\lambda x.\, e_2)\ e_1$.

1 Laziness

In previous lectures we defined semantics for both the call-by-name λ-calculus and the call-by-value λ-calculus. It turns out that we can translate a call-by-name program into a call-by-value program. 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. 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.

$$
\begin{aligned}
\mathcal{T}[\![x]\!] &= x\ (\lambda y.\, y) \\
\mathcal{T}[\![\lambda x.\, e]\!] &= \lambda x.\, \mathcal{T}[\![e]\!] \\
\mathcal{T}[\![e_1\ e_2]\!] &= \mathcal{T}[\![e_1]\!]\ (\lambda z.\, \mathcal{T}[\![e_2]\!]) \qquad\qquad z \text{ is not a free variable of } e_2
\end{aligned}
$$

2 References

We can also 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. The syntax of this language is defined as follows.

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

Expression $\mathsf{ref}\ e$ creates a new memory location (like a malloc), and sets the initial contents of the location to (the result of) e. The expression $\mathsf{ref}\ e$ itself evaluates to a memory location ℓ.

Think of a location as being like a pointer to a memory address. 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. Locations ℓ are not intended to be used directly by a programmer: they 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.

We define a small-step CBV operational semantics. We use configurations $\langle \sigma, e \rangle$, where e is an expression, and σ is a map from locations to values.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid \mathsf{ref}\ E \mid !E \mid E := e \mid v := E \qquad \frac{\langle \sigma, e \rangle \to \langle \sigma', e' \rangle}{\langle \sigma, E[e] \rangle \to \langle \sigma', E[e'] \rangle}$$

$$\beta\text{-REDUCTION}\ \frac{}{\langle \sigma, (\lambda x.\, e)\ v \rangle \to \langle \sigma, e\{v/x\} \rangle} \qquad \text{ALLOC}\ \frac{}{\langle \sigma, \mathsf{ref}\ v \rangle \to \langle \sigma[\ell \mapsto v], \ell \rangle}\ \ell \notin dom(\sigma)$$

$$\text{DEREF}\ \frac{}{\langle \sigma,\ !\ell \rangle \to \langle \sigma, v \rangle}\ \sigma(\ell) = v \qquad \text{ASSIGN}\ \frac{}{\langle \sigma, \ell := v \rangle \to \langle \sigma[\ell \mapsto v], v \rangle}$$

References do not add any expressive power to the λ-calculus: it is possible to translate λ-calculus with references to the pure λ-calculus. Intuitively, this is achieved by explicitly representing the store, and threading the store through the evaluation of the program. The details are left as an exercise.

3 Adequacy of translation

In each of the previous translations, we defined a semantics for the source language (using evaluation contexts and small-step rules) and the target language (by translation). We would like to be able to show that the translation is correct—that is, that it preserves the meaning of source programs.

More precisely, 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 "equivalent to" v. What exactly it means for v' to be "equivalent to" v will depend on the translation. Sometimes, it will mean that v' is literally the translation of v; other times, it will mean that v' is merely related to the translation of v by some equivalence.

One tricky issue is that in general, there can be many ways to define equivalences on functions. One way is to say that two functions are equivalent if they agree on the result when applied to any value of a base type (e.g., integers or booleans). The idea is that if two functions disagree when passed a more complex value (say, a function), then we could write a program that uses these functions to produce functions that disagree on values of base types.

There are two criteria for a translation to be *adequate*: soundness and completeness. For clarity, let's suppose that $\mathbf{Exp}_{\text{src}}$ is the set of source language expressions, and that $\to_{\text{src}}$ and $\to_{\text{trg}}$ are the evaluation relations for the source and target languages respectively. A translation is sound if every target evaluation represents a source evaluation:

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

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

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

4 Continuations

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

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

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 considerable, it doesn't work as well.

Continuations are a programming technique that may be used directly by a programmer, or used in program transformations by a compiler. Because they make the control flow of the program explicit, they can be used to overcome discrepancies between source and target languages in definitional translation. They can also be used to define the semantics of control-flow constructs such as exceptions.

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

$$\mathsf{if}\ \mathsf{foo} < 10\ \mathsf{then}\ 32 + 6\ \mathsf{else}\ 7 + \mathsf{bar}$$

and consider the evaluation of the expression $\mathsf{foo} < 10$. When we finish evaluating this subexpression, we will evaluate the if statement, and then evaluate the appropriate branch. The *continuation* of the subexpression $\mathsf{foo} < 10$ is the rest of the computation that will occur after we evaluate the subexpression. We can write this continuation as a function that takes the result of the subexpression:

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

The evaluation order and result of this program will be the same as the original expression; the difference is that we extracted the continuation of the subexpression in to a function.

The nice thing about continuations is that it makes the control explicit, and this is especially useful in the case of functional programs, where control is not explicit otherwise. In fact, we can rewrite a program to make continuations more explicit. Let's consider another program, and convert it so that continuations are explicit

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

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

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

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.

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

The program itself is now equivalent to $k_3\ 1$. Since $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ is just syntactic sugar for $(\lambda x.\, e')\ e$, we can actually rewrite the above as

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

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

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

Using continuations, functions can be transformed into "functions that don't return"—i.e., functions that take, besides the usual arguments, an additional argument representing a continuation. 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, or CPS for short. For instance, the CPS version of factorial looks like the following:

$$\mathsf{FACT}_{cps} = \mathsf{Y}\ \lambda f.\, \lambda n, k.\, \mathsf{if}\ n = 0\ \mathsf{then}\ k\ 1\ \mathsf{else}\ f\ (n - 1)\ (\lambda v.\, k\ (n * v))$$

Note that the last thing that code in FACT_{cps} does is call a function (either k or f), and does not do anything with the result.

Continuation-passing style is an important concept in the compilation of functional languages and is used as an intermediate compiler representation (it has been used in compilers for Scheme, ML, etc). The main advantage is that CPS makes the control flow explicit and makes it easier to translate functional code to machine code where control is explicit (in the form of sequences of machine instructions and jumps). For instance, a CPS call can be easily translated into a jump to the invoked method, since the invoked function does not return the control.

4.1 CPS translation

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

Let's consider a translation from λ-calculus with pairs and integers. The syntax of the source language is as follows.

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

The translation $\mathcal{CPS}[\![e]\!]$ will produce a function that 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. 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.

For convenience, instead of writing $\mathcal{CPS}[\![e]\!] = \lambda k. \ldots$ we write $\mathcal{CPS}[\![e]\!]\ k = \ldots$.

$$
\begin{aligned}
\mathcal{CPS}[\![n]\!]\ k &= k\ n \\
\mathcal{CPS}[\![e_1 + e_2]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda n. \mathcal{CPS}[\![e_2]\!]\ (\lambda m. k\ (n+m))) && n \text{ is not a free variable of } e_2
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{CPS}[\![(e_1, e_2)]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda v. \mathcal{CPS}[\![e_2]\!]\ (\lambda w. k\ (v, w))) && v \text{ is not a free variable of } e_2 \\
\mathcal{CPS}[\![\#1\ e]\!]\ k &= \mathcal{CPS}[\![e]\!]\ (\lambda v. k\ (\#1\ v)) \\
\mathcal{CPS}[\![\#2\ e]\!]\ k &= \mathcal{CPS}[\![e]\!]\ (\lambda v. k\ (\#2\ v))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{CPS}[\![x]\!]\ k &= k\ x \\
\mathcal{CPS}[\![\lambda x. e]\!]\ k &= k\ (\lambda x. \lambda k'. \mathcal{CPS}[\![e]\!]\ k') && k' \text{ is not a free variable of } e \\
\mathcal{CPS}[\![e_1\ e_2]\!]\ k &= \mathcal{CPS}[\![e_1]\!]\ (\lambda f. \mathcal{CPS}[\![e_2]\!]\ (\lambda v. f\ v\ k)) && f \text{ is not a free variable of } e_2
\end{aligned}
$$

We translate a function $\lambda x. e$ to a function that takes an additional argument k', which is the continuation after the function application. That is, k' is the continuation to which we hand the result of evaluating the function body. In function application, we see that in addition to the actual argument, we also give the continuation as the additional argument.

Let's see an example translation and execution...

$$
\begin{aligned}
\mathcal{CPS}[\![(\lambda a. a + 6)\ 7]\!]\ \mathsf{ID} &= \mathcal{CPS}[\![(\lambda a. a + 6)]\!]\ \ (\lambda f. \mathcal{CPS}[\![7]\!]\ (\lambda v. f\ v\ \mathsf{ID})) \\
&= (\lambda f. \mathcal{CPS}[\![7]\!]\ (\lambda v. f\ v\ \mathsf{ID}))\ (\lambda a, k'. \mathcal{CPS}[\![a + 6]\!] k') \\
&= (\lambda f. (\lambda v. f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'. \mathcal{CPS}[\![a + 6]\!] k') \\
&= (\lambda f. (\lambda v. f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'. \mathcal{CPS}[\![a]\!]\ (\lambda n. \mathcal{CPS}[\![6]\!]\ (\lambda m. k'\ (m+n)))) \\
&= (\lambda f. (\lambda v. f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'. \mathcal{CPS}[\![a]\!]\ (\lambda n. (\lambda m. k'\ (m+n))\ 6)) \\
&= (\lambda f. (\lambda v. f\ v\ \mathsf{ID})\ 7)\ (\lambda a, k'. (\lambda n. (\lambda m. k'\ (m+n))\ 6)\ a) \\
&\to (\lambda v. (\lambda a, k'. (\lambda n. (\lambda m. k'\ (m+n))\ 6)\ a)\ v\ \mathsf{ID})\ 7 \\
&\to (\lambda a, k'. (\lambda n. (\lambda m. k'\ (m+n))\ 6)\ a)\ 7\ \mathsf{ID} \\
&\to (\lambda n. (\lambda m. \mathsf{ID}\ (m+n))\ 6)\ 7 \\
&\to (\lambda m. \mathsf{ID}\ (m+7))\ 6 \\
&\to \mathsf{ID}\ (6+7) \\
&\to \mathsf{ID}\ 13 \\
&\to 13
\end{aligned}
$$