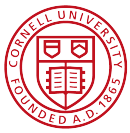
CS 4110 – Programming Languages and Logics
Lecture #18: Definitional Translation



In this lecture we develop the idea of definitional translation (for multi-argument functions, products, let-expressions, and laziness, and mutable references) and discusses correctness.

1 Definitional translation

We have seen how to encode a number of high-level language constructs—booleans, conditionals, natural numbers, and recursion—in λ-calculus. We now consider definitional translation, where we define the meaning of language constructs by translation to another language. This is a form of denotational semantics, but instead of the target being mathematical objects, it is a simpler programming language (such as λ-calculus). Note that definitional translation does not necessarily produce clean or efficient code; rather, it defines the meaning of the source language constructs in terms of the target language.

For each language construct, we will define an operational semantics directly, and then give an alternate semantics by translation to a simpler language.

2 Multi-argument functions and currying

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

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

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

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

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

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

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

Now, the multi-argument λ-calculus isn't any more expressive that the pure λ-calculus. We can show this by showing how any multi-argument λ-calculus program can be translated into an equivalent pure λ-calculus program. We define a translation function $\mathcal{T}[\![\cdot]\!]$ that takes an expression in the multi-argument λ-calculus and returns an equivalent expression in the pure

λ-calculus. That is, if e is a multi-argument lambda calculus expression, $\mathcal{T}[\![e]\!]$ is a pure λ-calculus expression.

We define the translation as follows.

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

This process of rewriting a function that takes multiple arguments as a chain of functions that each take a single argument is called *currying*. Consider a mathematical function that takes two arguments, the first from domain A and the second from domain B, and returns a result from domain C. We could describe this function, using mathematical notation for domains of functions, as being an element of $A \times B \to C$. Currying this function produces a function that is an element of $A \to (B \to C)$. That is, 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.

3 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 $(\lambda x.\, e_2)\ e_1$.

4 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.

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

5 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 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. 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.

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

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

$$\textsc{Deref}\ \frac{}{\langle \sigma,\ !\ell \rangle \to \langle \sigma, v \rangle}\ \sigma(\ell) = v \qquad \textsc{Assign}\ \frac{}{\langle \sigma, \ell := v \rangle \to \langle \sigma[\ell \mapsto v], v \rangle}$$

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.

6 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:

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

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

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