Definitional translation

Lecture 9 Tuesday, February 23, 2015

1 Definitional translation

We saw that the denotational semantics of IMP defined the meaning of IMP commands as mathematical functions from stores to stores. We described denotational semantics as being like compilation, from IMP to mathematics. 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 language being mathematics, it is a simpler programming language. Note that 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.

In this lecture, we will consider a number of language features, define an operational semantics for them, and then give an alternate semantics by translation to a simpler language. We first introduce *evaluation contexts* to help us present the new language features succinctly.

1.1 Evaluation contexts

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

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

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

Of the operational semantics rules, only the β-reduction rule told us how to "reduce" an expression; the other two rules were simply telling us the order to evaluate expressions in, i.e., first evaluate the left hand side of an application to a value, then evaluate the right hand side of an application to a value. 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 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. Evaluation contexts are defined using a BNF grammar that is similar to the grammar used to define the language. 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. The following are examples of evaluation contexts, and evaluation contexts with the hole filled in by an expression.

$$E_1 = [\cdot]\ (\lambda x.\, x) \qquad E_1[\lambda y.\, y\ y] = (\lambda y.\, y\ y)\ \lambda x.\, x$$
$$E_2 = (\lambda z.\, z\ z)\ [\cdot] \qquad E_2[\lambda x.\, \lambda y.\, x] = (\lambda z.\, z\ z)\ (\lambda x.\, \lambda y.\, x)$$
$$E_3 = ([\cdot]\ \lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda y.\, y)) \qquad E_3[\lambda f.\, \lambda g.\, f\ g] = ((\lambda f.\, \lambda g.\, f\ g)\ \lambda x.\, x\ x)\ ((\lambda y.\, y)\ (\lambda y.\, y))$$

Using evaluation contexts, we can define the evaluation semantics for the pure CBV lambda calculus with just two rules, one for evaluation contexts, and one for β-reduction.

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

Note that the evaluation contexts for the CBV lambda calculus 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 β-reduction.

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

$$E ::= [\cdot] \mid E\ e \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \beta\text{-REDUCTION}\ \frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1\{e_2/x\}}$$

We'll see the benefit of evaluation contexts as we see languages with more syntactic constructs.

1.2 Multi-argument functions and currying

Our syntax for functions restricted us to function that have 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 lambda calculus as follows.

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

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

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 lambda calculus isn't any more expressive that the pure lambda calculus. We can show this by showing how 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.

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.

1.3 Products and let

We introduce two useful language features to the lambda calculus: products and let expressions.

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, $\#1\ (v_1, v_2) \longrightarrow v_1$ and $\#2\ (v_1, v_2) \longrightarrow v_2$. (Other common notation for projection includes π_1 and π_2, and fst and snd.)

More formally, we define the syntax of lambda calculus with products and let expressions as follows. Values in this language are either functions or pairs of values.

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

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

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

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

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

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

We can give an equivalent semantics by translation to the pure CBV lambda calculus. 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.

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

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

1.4 CBN to CBV

We've seen semantics for both the call-by-name lambda calculus and the call-by-value lambda calculus. 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.

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

1.5 Adequacy of translation

We've presented several translations of languages. In each case, we had a semantics defined for both the source and target language. We would like the translation to be correct, that is, to preserve 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 "equal to" v.

What exactly it means for v' to be "equal to" v will depend on the translation. Sometimes, it will mean that v' is the translation of v; other times, it will mean that v' is somehow equivalent to the translation of v. In particular, we often need to define equivalence on functions. One possible solution is 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 $\longrightarrow_{\text{src}}$ and $\longrightarrow_{\text{trg}}$ are the evaluation relations for the source and target languages respectively.

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

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

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

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