This lecture concentrates on sematics by way of translation from a source language to a target language. It includes two examples of translation from Call By Name (CBN) semantics to Call By Value (CBV) semantics and from a simplified version of untype ML called uML into CBV lambda calculus. The notion of adequacy of translation is introduced. Big-step semantics for CBV lambda calculus are described.

- Big-step semantics for CBV λ-calculus
- Definition of translation
- Translation of CBN into CBV
- Adequacy of translation
- Translation of uML into CBV

1 Big-step semantics for CBV λ-calculus

$$\frac{e\{v/x\} \Downarrow v'}{(\lambda x.\, e)\ v \Downarrow v'} \qquad \frac{e \Downarrow (\lambda x.\, e) \quad e' \Downarrow v' \quad e\{v'/x\} \Downarrow v''}{e\ e' \Downarrow v''}$$

2 Translation

Translation from a well formed source language to a well understood target language is useful not only for reasoning about a program, but also for writing compilers.

- If the target language is a programming language then the translation is called a *definitional translation*. Note that a definitional translation does not necessarily produce clean or efficient code.
- If the target language is mathematical objects then the translation is called *denotational semantics*.

2.1 Notation

$[\![\cdot]\!]$ is a function that translates an expression in the source language to something meaningful. The things that have meaning are in the target languages. $\mathcal{C}[\![e]\!]$ or $[\![e]\!]_{CBV}$ notation can be used to name translation functions.

2.2 CBN into CBV

Recall CBN and CBV sematincs with evaluation contexts:

- CBN

$$E ::= E\ e \ \mid\ [\cdot]$$

$$E[(\lambda x.\, e)\ e'] \to E[e\{e'/x\}]$$

- CBV

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

$$E[(\lambda x.\, e)\ v] \to E[e\{v/x\}]$$

Can we translate CBN into CBV? In CBV an application $e\ e'$ is not evaluated unless e' is a value. To delay evaluation of e', we can place it inside a value. This trick effectively makes a thunk.

- CBV: $e\ e'$
- CBN: $e\ (\lambda x.\, e')$

Using this trick, we can define a translation function from CBN into CBV as follows:

$$\begin{aligned} [\![e\ e']\!] &= [\![e]\!](\lambda z.\, [\![e']\!]) \\ [\![(\lambda x.\, e')]\!] &= \lambda x.\, [\![e]\!] \\ [\![x]\!] &= x\ ID \end{aligned}$$

Notice that the translation from CBN to CBV describes a CBN compiler. This translation would have to be performed if CBN were to be implemented in hardware.

2.3 Short-circuiting **if**-statements in CBV

Recall these definitions:

$$\begin{aligned} \mathbf{true} &\triangleq \lambda x.\lambda y.x \\ \mathbf{false} &\triangleq \lambda x.\lambda y.y \\ \mathbf{if} &\triangleq \lambda b.\lambda x.\lambda y.bxy \end{aligned}$$

Using CBV semantics to evaluate the experssion **if** $b\ e_1\ e_2$ both e_1 and e_2 are evaluated before a choice is made. As we have defined it so far, **if** is *strict* in e_1 and e_2; that is, if either e_1 or e_2 fail to terminate, **if** won't either. Ordinarily, we would expect **if** to be strict only in b, and lazy in e_1 and e_2.
Example: **false** $\Omega\ ID$

- CBV semantics evaluate to Ω
- CBN semantics evaluate to ID

We can use CBN translation to achieve this effect. First attempt:

$$[\![\lambda x.\lambda y.x]\!] = \lambda x.[\![\lambda y.x]\!] = \lambda x.\lambda y.(x\ ID)$$

$$[\![\lambda x.\lambda y.x]\!] = \lambda x.[\![\lambda y.y]\!] = \lambda x.\lambda y.(y\ ID)$$

$$[\![\lambda b.\lambda x.\lambda y.byx]\!] = \lambda b.\lambda x.\lambda y.[\![bxy]\!] = \lambda b.\lambda x.\lambda y.[\![bx]\!](\lambda z.[\![y]\!]) = \lambda b.\lambda x.\lambda y.(b\ ID)(\lambda z.[\![x]\!])(\lambda z.[\![y]\!])$$

Notice that this translation is trying to be lazy in b, which is not necessary. Second attempt:

$$[\![\mathbf{if}\ e_b\ e_1\ e_2]\!]_{if} = [\![e_b]\!]_{if}(\lambda z.[\![e_1]\!]_{if})(\lambda z.[\![e_2]\!]_{if})$$

2.4 Adequacy of translation

What does it mean to say that a translation is correct? Intuitively, we think that if an expression e in the source language steps in zero or more steps to a values v and the translation of e step in zero or more steps to a value v_t in the target language, then $v_t \approx [\![v]\!]$.

source	$e \rightarrow^* v$
target	$[\![e]\!] \rightarrow^* v_t\ \approx\ [\![v]\!]$

There are two criteria for a translation to be considered *adequate*: *soundness* and *completeness*.

- Soundness:

  $$[\![e]\!] \rightarrow^* v_t\ \Rightarrow\ \exists v\ \mid\ (e \rightarrow^* v) \wedge ([\![v]\!] \approx v_t))$$

  Every target evaluation represents a source evaluation.

- Completeness:

$$e \rightarrow^* v \;\Rightarrow\; \exists v_t \;\mid\; ([\![e]\!] \rightarrow_t^* v_t) \wedge (v_t \approx [\![v]\!]))$$

  Every source evaluation represents a target evaluation.

There is one caveat: it is difficult to define equivalence on functions. One possible solution is to show that function agree on base types (integers, booleans, etc.). The idea is that if the source and target language disagree on some functions, we could use write another program that used the disagreement on functions to force a disagreement on base types, by applying those functions to values at which the function disagreed.

2.5 uML

untyped ML syntax:

$$\begin{array}{rcl} e ::= n & \mid & x \mid \textbf{true} \mid \textbf{false} \mid \lambda x_1 \ldots x_n.e \mid e_0\ e_1\ \ldots\ e_n \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\ & \mid & (e_1, e_2) \mid \#1\ e \mid \#2\ e \mid \textbf{if}\ e_0\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \end{array}$$

Let's define a translation function from uML to CBV λ-calculus

$$[\![\cdot]\!] : uML \rightarrow \lambda_{CBV}$$

$$\begin{array}{rcl} [\![n]\!] & = & church\ numeral\ for\ n \\ [\![x]\!] & = & x \\ [\![\textbf{true}]\!] & = & \lambda x.\lambda y.x \\ [\![\textbf{false}]\!] & = & \lambda x.\lambda y.y \\ [\![\textbf{if}\ b\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!] & = & [[\![b]\!](\lambda z.[\![e_1]\!])(\lambda z.[\![e_2]\!])]ID \\ [\![\lambda x_1 x_2 \ldots x_n.e]\!] & = & \lambda x_1.[\![\lambda x_2 ... \lambda x_n.e]\!] \\ [\![e_0\ e_1 \ldots e_n]\!] & = & [\![e_0 \ldots e_{n-1}]\!][\![e_n]\!] \\ [\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!] & = & (\lambda x.[\![e_2]\!])[\![e_1]\!] \\ [\![(e_1, e_2)]\!] & = & CONS[\![e_1]\!][\![e_2]\!] \\ [\![\#1\ e]\!] & = & FIRST[\![e]\!] \\ [\![\#2\ e]\!] & = & SECOND[\![e]\!] \end{array}$$

Unfortunately, this translation is not sound, because it allows target-language evaluations that correspond to no source- language evaluation. For example, the the source code expression **true true** has no legal next step; it gets *stuck*, because there is no operational semantics rule in uML of the form **true true** $\rightarrow_{uML} e$. However, the *translation* of **true true** is a valid lambda-calculus expression that evaluates to another lambda-calculus expression. What happens in this evaluation depends on the implementation of **true** and cannot be understood from the equational specification of booleans.

A possible solution to the soundness difficulty is to introduce many rules that transition stuck expressions to a special "error" value. For example, we might have a rule **true** $e \rightarrow error$ to cover the case just given. However, we will still have to fix our translation to add run-time type checking, in order to catch these errors, and we'll have to add a representation of *error*. Another solution is type checking and throwing a compile-time type error!