Lecture notes for CS 6110 (Spring'09) taught by Andrew Myers at Cornell; edited by Amal Ahmed, Fall'09.

1 The limitations of direct translation

So far we have been building translations that preserve control flow. That is, the control flow in the target language corresponds directly to control flow in the source. This makes it difficult to describe language features that involve *non-local control flow*, such as errors, exceptions, goto, break, continue, and tail recursion. All of these features violate the simple stack-like control flow of the lambda calculus, which comes from the fact that functions impose a call/return discipline. To do a better job of capturing non-local control flow, we go beyond functions to *continuations*.

We will see that because continuations expose control explicitly, they make a good intermediate language for compilation, because control is exposed explicitly in machine language as well. (We can show this by writing a translation from uML to a language similar to assembly. Doing such a translation would give us a fairly complete recipe for compiling any of the languages we have talked about in class down to the hardware.)

2 Continuations

Consider the statement **if** $x \leq 0$ **then** x **else** $x + 1$. Using evaluation contexts, we can separate the redex from the rest of the expression:

$$(\textbf{if } [\cdot] \textbf{ then } x \textbf{ else } x + 1)[x \leq 0]$$

Of course, we have another way of performing subsitution: β-reduction. We can also write this as:

$$(\lambda y.\, \textbf{if } y \textbf{ then } x \textbf{ else } x + 1)\ (x \leq 0)$$

To evaluate this, we would first evaluate the argument $x \leq 0$ to obtain a boolean value, then apply the function $\lambda y.\, \textbf{if } y \textbf{ then } x \textbf{ else } x + 1$ to this value. This function captures an evaluation context as an explicit, separate term. The function $\lambda y.\, \textbf{if } y \textbf{ then } x \textbf{ else } x + 1$ is called a *continuation*, because it specifies what is to be done with the result of the current computation in order to continue the computation.

We can perform this kind of transformation on every evaluation context in the program. The result will be a term in continuation-passing style (CPS). Because it makes transfers of control explicit, CPS is a better target language when we want to capture more complex ways to transfer control.

3 Continuation-passing style

We can define a version of the lambda calculus that syntactically enforces continuation-passing style.

Recall that our grammar for the λ-calculus was:

$$e \ ::= \ x \ \mid \ \lambda x.\, e \ \mid \ e_0\ e_1$$

Our grammar for the CPS λ-calculus will be:

$$v \;:=\; x \;\mid\; \lambda x_1 \ldots x_n.\, e \qquad\qquad e \;::=\; v_0\ v_1\ \cdots\ v_n$$

This is a highly constrained syntax. Barring reductions inside the scope of a λ-abstraction operator, the expressions v are all irreducible. The only reducible expression is $v_0\ v_1\ \cdots\ v_n$. In fact, we will only use $n = 1$ and $n = 2$ to get the full expressive power of the lambda calculus. When $n = 1$, we have a continuation. We will use $n = 2$ to model source-language functions.

The small-step semantics is particularly simple. It has a single rule for β reduction:

$$(\lambda x_1 \ldots x_n.\, e)\ v_1\ \ldots\ v_n \quad\longrightarrow\quad e\{v_i/x_i\}^{(i\in 1..n)}$$

Notice that we do not need any evaluation contexts! The syntax ensures that if there is a β redex, it is at the top level.

The big-step semantics is also simple, with only a single rule:

$$\frac{e\{v_i/x_i\}^{(i\in 1..n)} \Downarrow v'}{(\lambda x_1 \ldots x_n.\, e)\ v_1\ \ldots\ v_n \Downarrow v'.}$$

The resulting proof tree will not be tree-like. The rule has one premise, so a proof will be a stack of rule instances, each one corresponding exactly to a step in the small-step semantics. In other words, the big- and small-step semantics correspond exactly.

Both semantics lead us to build an interpreter that runs a loop performing reductions, and does not need to make recursive calls. The fact that we can build a simpler interpreter for the language is a strong hint that this language is lower-level than the lambda calculus. Because it is lower-level (and actually closer to assembly code), CPS is typically used in functional language compilers as an intermediate representation. It also is a good code representation if one is building an interpreter,

6 CPS conversion

Despite restricting the syntax in CPS, we haven't lost expressive power. Given a λ-calculus expression e, it is possible to define a translation $[\![e]\!]$ that translates it into CPS. This translation is known as *CPS conversion*. It was developed as part of Steele's Rabbit Scheme compiler, but was described earlier by Reynolds.

The translation takes an arbitrary λ-term e and produces a CPS term $[\![e]\!]$ which is a function taking a continuation as argument. Intuitively, $[\![e]\!]k$ applies k to the result of e. We can think of a continuation as a drop box that expects a result and sends it on its way through the rest of the computation—but never returns.

We want our translation to satisfy $e \xrightarrow[CBV]{*} v \iff [\![e]\!]k \xrightarrow[CPS]{*} [\![v]\!]k$ for primitive values v and any variable $k \notin FV(e)$, and $e \Uparrow_{CBV} \Leftrightarrow [\![e]\!]k \Uparrow_{CPS}$.

One challenge is how to translate functions into CPS. Since all control transfers must be explicit applications of continuations, we must make the return from a function call explicit as a distinct continuation. Therefore, we represent a function $\lambda x.\ \cdots$ in CPS form as a function that takes an explicit continuation argument: $\lambda k x.\ \cdot$. The argument k corresponds to the return address to which the function will return.

Using this insight, the translation is as follows:

$$\begin{aligned}
[\![x]\!]k &= k\ x \\
[\![\lambda x.\, e]\!]k &= k\ (\lambda k' x.\, [\![e]\!]k') \\
[\![e_0\ e_1]\!]k &= [\![e_0]\!](\lambda f.\, [\![e_1]\!](\lambda v.\, f\ k\ v))
\end{aligned}$$

(Here we have defined $[\![e]\!]$ in the style of ML; thus $[\![e]\!]k = e'$ really means $[\![e]\!] = \lambda k.\, e'$.)

6.1 An example

In λ_{CBV}, we have

$$(\lambda xy.\, x)\ 1 \quad \rightarrow \quad \lambda y.\, 1$$

Let's evaluate the CPS-translation of the left-hand side using the CPS evaluation rules. Note that we translate integers and other simple values just like we do variables:

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

Converting to CPS, we get:

$$\begin{array}{rcl}
[\![(\lambda xy.\, x)\ 1]\!]k & = & [\![\lambda x.\, \lambda y.\, x]\!](\lambda f.\, [\![1]\!](\lambda v.\, f\ k\ v)) \\
& = & (\lambda f.\, [\![1]\!](\lambda v.\, f\ k\ v))(\lambda k' x.\, [\![\lambda y.\, x]\!]k') \\
& = & (\lambda f.\, (\lambda v.\, f\ k\ v)\ 1)(\lambda k' x.\, [\![\lambda y.\, x]\!]k') \\
& = & (\lambda f.\, (\lambda v.\, f\ k\ v)\ 1)(\lambda k' x.\, k'\ (\lambda k'' y.\, k''\ x)) \\
& \longrightarrow & (\lambda v.\, (\lambda k' x.\, k'\ (\lambda k'' y.\, k''\ x))\ k\ v)\ 1 \\
& \longrightarrow & (\lambda k' x.\, k'\ (\lambda k'' y.\, k''\ x))\ k\ 1 \\
& \longrightarrow & k\ (\lambda k'' y.\, k''\ 1) \\
& = & [\![\lambda y.\, 1]\!]k \\
& = & k\ \mathcal{V}[\![\lambda y.\, 1]\!]
\end{array}$$