Scribe: Greg Bronevetsky and Elliot Anshelevich Lecturer: Andrew Myers

1 Introduction

The semantics we've used so far are called *Direct Semantics* because source language constructs are translated directly into meta-language constructs. For example, a function call is translated into a functions application. Direct semantics are less convenient for modeling non-local control transfers such as exceptions, gotos, etc.

To define the semantics of such a language we can use *Continuation Passing Style Semantics*. The idea is this: a continuation is the part of the program that has yet to be evaluated. Instead of rewriting a given expression in terms of other expressions, we take things one step at a time. We evaluate the current expression as much as possible and pass the result to the rest of the program. So a continuation takes in the results of the program so far and finishes the job of execution.

For example, this program:

$$\textbf{if } x < 0 \textbf{ then } x \textbf{ else } f(x-1)$$

can be broken up into the expression

$$x < 0$$

and the continuation (expressed here as an evaluation context)

$$\textbf{if}[\cdot] \textbf{ then } x \textbf{ else } f(x-1)$$

The expression is evaluated and then passed into the continuation, which takes it the rest of the way. Writing the continuation as a function, we can transform this program into:

$$(\lambda v.\ \textbf{if } v \textbf{ then } x \textbf{ else } \ f(x-1))(x < 0)$$

Applying this transformation to every part of the program , we produce a program in continuation style (CPS). CPS is a lot like lambda calculus except that where functions in lambda calculus return values, CPS functions do not. (or they always return Unit, which is effectively the same thing) We can think of CPS as a restricted form of λ calculus that can be described by the following grammar:

$$e ::= x \ \mid \ \lambda x.\ s \ \mid \ \mathsf{halt}$$

The λ expression is a continuation. It waits for the value x and then goes on with the rest of the program. halt is a special continuation that accepts an argument, halts the program, and returns the value of that argument as the result of the program. For example, a program that evaluates to halt 2 is a program whose result it 2.

We still need to define a statement s:

$$s ::= e_1\ e_2$$

Since statements do not return values, neither do functions. Thus, CPS is a subset of lambda calculus that never uses function results.

2 Small Step Semantics ($s \rightarrow s'$)

$$v := \lambda x.\ s \ \mid \ \mathsf{halt}$$

Final Configuration: $\mathsf{halt}\ e$

$$(\lambda x.\ s)\ e \rightarrow s\{e/x\}$$

We don't say anything about the evaluation contexts because we just do everything step-by-step and the context follows this evaluation.

3 Large Step Semantics ($s \Downarrow v$)

$$v := \lambda x.\ s \ \mid \ \mathsf{halt}$$

$$\frac{}{\mathsf{halt}\ v \Downarrow v} \qquad \frac{s\{e/x\} \Downarrow v}{(\lambda x.\ s)\ e \Downarrow v}$$

The proof tree for a CPS program is just a stack as code fragments lower in the stack pass their subresults up the stack. Accordingly, designing a recursive interpreter for such a program wouldn't make sense since control never passes down the stack, only up. And thus, such a program should be evaluated in a loop. Here's an example:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{halt}\ (\lambda y.\ y) \Downarrow (\lambda y.\ y)}}{\mathsf{halt}\ (y\{(\lambda y.\ y)/y\} \Downarrow)(\lambda y.\ y)}}{\mathsf{halt}\ ((\lambda y.\ y)\ (\lambda y.\ y)) \Downarrow (\lambda y.\ y)}}{\mathsf{halt}\ ((x\ x)\ \{(\lambda y.\ y)/x\}) \Downarrow (\lambda y.\ y)}}{\mathsf{halt}\ ((\lambda x.\ x\ x)\ (\lambda y.\ y)) \Downarrow (\lambda y.\ y)}$$

4 Regaining the power of Lambda calculus

CPS is a subset of lambda calculus but is unfortunately not as powerful. However, the full power of lambda calculus can be regained by allowing for the use of 2 argument functions. So let's change the previous CPS grammar to:

$$\begin{aligned} e &::= x \ \mid \ \lambda x.\ s \ \mid \ \lambda\ x_1\ x_2.s \\ s &::= e_1\ e_2 \ \mid \ e_1\ e_2\ e_3 \end{aligned}$$

This is a 2-argument function application in Small-Step semantics:

$$(\lambda\ x_1\ x_2.s)\ e_1\ e_2 \rightarrow s\{e_1/x_1,\ e_2/x_2\}$$

Note that we're not using currying here but are instead just passing two arguments to this function. This is because currying depends on functions returning other functions. Unfortunately, in CPS functions don't return values. Ooops! So we just pass the two arguments directly.

The two arguments in the function have different roles. The expression $\lambda x_1 x_2.s$ will correspond to an ordinary λ calculus function.

The x_1 is the normal argument that the function would have normally received. The function does some processing using x_1 in s and then passes this information to x_2. The argument x_2 is a continuation that takes the intermediate results produced by the function and passes them on to the rest of the program.

5 CPS Conversion

We can convert normal Lambda calculus expressions into equivalent CPS statements. For this purpose we'll use the functions $\mathcal{T}$ and $\mathcal{D}$ where $\mathcal{T}$ is conversion function that takes in an expression e and converts it into a CPS statement s.

$$\mathcal{T}[\![e]\!] \;=\; s$$

Our goal here is to make sure that if a λ expression evaluates to v, then so does its corresponding CPS statement.

$$e \rightarrow^* v \iff s \rightarrow^* \mathsf{halt}\ v$$

Unfortunately, this won't quite hold because the form of v will be different depending on whether it's expressed as a λ expression or in CPS so we rephrase it as:

$$e \rightarrow^* v \iff s \rightarrow^* \mathsf{halt}\ \mathcal{T}[\![v]\!]$$

Even this equivalence doesn't hold because there are other functions equivalent to $\mathcal{T}[\![v]\!]$ that might be produced by evaluation. Now let's actually define the CPS Conversion. First of all we have to make sure that the resulting CPS program ends with a halt.

$$\mathcal{T}[\![e]\!] \quad = \quad \mathcal{D}[\![e]\!]\ \mathsf{halt}$$

Having taken care of that, let's define $\mathcal{D}$:

$$\mathcal{D}[\![e]\!]k \quad = \quad s$$

Here s needs to evaluate e and then pass the result to k. e is the lambda calculus term and k is a one argument function: the continuation that expects whatever s will give to it so that it can go on processing.

$$\begin{aligned} \mathcal{D}[\![x]\!]k &= k\ x \\ \mathcal{D}[\![\lambda\ x.e]\!]k &= k(\lambda\ k',\ x.\mathcal{D}[\![e]\!]k') \quad \textit{where } k \notin FV[e] \end{aligned}$$

Here k is the continuation that receives the function as a whole. The function itself has been modified to accept the continuation k' and pass the result of e to k'.

$$\mathcal{D}[\![e_1\ e_2]\!]k = \mathcal{D}[\![e_1]\!](\lambda f.\mathcal{D}[\![e_2]\!](\lambda\ v.f\ v\ k)) \quad \textit{where } f,\ v \notin FV[k] \textit{ and } f \notin FV[e_1]$$

Here $\mathcal{D}[\![e_1]\!]$ gets the rest of the expression as its continuation. When $\mathcal{D}[\![e_1]\!]$ evaluates, its result gets passed to the λ expression as f. Now $\mathcal{D}[\![e_1]\!]$ is the object of focus and it gets the second λ expression as its continuation. When $\mathcal{D}[\![e_2]\!]$ evaluates, its result is passed to its continuation, which now has everything it needs to do its job. It has f, which is the function represented by e_1. It has v, which is the value represented by e_2. Finally, it has k, the original continuation which will get the result of $\mathcal{D}[\![e_1\ e_2]\!]$. The expression ($f$ v k) neatly implements the application by passing the argument and the continuation into the function.

We now have a complete conversion function from Lambda calculus to CPS. Note also that we can translate λ calculus into standard λ calculus in a CPS style by defining halt as a λ expression. This is simple: halt is just the identity. Note that halt winds up being the last function left when the rest of the program has been evaluated and thus, it is the only function to return a value. So let's rewrite $\mathcal{T}$ as:

$$\mathcal{T}[\![e]\!] = \mathcal{D}[\![e]\!](\lambda x.x)$$

Now let's try translating a sample program from plain lambda calculus to CPS. The program is: $(((\lambda\, x.\lambda\, y.y)1)2)$

$$\begin{array}{lcll}
= & \mathcal{D}[\![(((\lambda\ x.\lambda\ y.y)1)2)]\!]\ \mathsf{halt} & \\
= & \mathcal{D}[\![((\lambda\ x.\lambda\ y.y)1)]\!]\ (\lambda\ f_1.\mathcal{D}[\![2]\!](\lambda v_1.f_1\ v_1\ \mathsf{halt})) & \{\lambda\ f_1.\mathcal{D}[\![2]\!](\lambda v_1.f_1\ v_1\ \mathsf{halt}) = \lambda\ f_1.f_1\ 2\ \mathsf{halt}\} \\
= & \mathcal{D}[\![(\lambda\ x.\lambda\ y.y)]\!]\ (\lambda\ f_2.\mathcal{D}[\![1]\!](\lambda v_2.f_2\ v_2\ (\lambda\ f_1.f_1\ 2\ \mathsf{halt}\})) & \{\lambda\ f_2.\mathcal{D}[\![1]\!](\lambda v_2.f_2\ v_2\ (\lambda\ f_1.f_1\ 2\ \mathsf{halt}\}) = \\
 & & \qquad = \lambda\ f_2.f_2\ 1\ (\lambda\ f_1.f_1\ 2\ \mathsf{halt})\} \\
= & (\lambda\ f_2.f_2\ 1\ (\lambda\ f_1.f_1\ 2\ \mathsf{halt}))\ (\lambda\ x,k.\mathcal{D}[\![\lambda\ y.y]\!]k) & \{\mathcal{D}[\![\lambda\ y.y]\!]k' = k(\lambda\ y,k'.k'(y))\} \\
= & (\lambda\ f_2.f_2\ 1\ (\lambda\ f_1.f_1\ 2\ \mathsf{halt}))\ (\lambda\ x,k.k(\lambda\ y,k'.k'(y))) &
\end{array}$$

This expression evaluates as follows, assuming halt is the (non-CPS) term $(\lambda\ x.x)$.

$$\begin{array}{ll}
\rightarrow & (\lambda\ x,k.k(\lambda\ y,k'.k'(y)))\ 1\ (\lambda\ f_1.f_1\ 2\ \mathsf{halt})) \\
\rightarrow & (\lambda\ f_1.f_1\ 2\ \mathsf{halt})\ (\lambda\ y,k'.k'(y)) \\
\rightarrow & (\lambda\ y,k'.k'(y))\ 2\ \mathsf{halt} \\
\rightarrow & \mathsf{halt}(2) \\
\rightarrow & (\lambda\ x.x)\ 2 \\
\rightarrow & 2
\end{array}$$