Harvard School of Engineering and Applied Sciences — Computer Science 152

Control-flow constructs; Scope

Lecture 12 Thursday, March 4, 2010

1 Control flow constructs

CPS makes control flow explicit in the lambda calculus. This makes it easier to express complex control flow constructs. Here, we consider two non-local control-flow constructs: exceptions and co-routines.

1.1 Exceptions

Consider a CBV language with a simple exception-throwing mechanism. The syntax of the language is given by the following grammar.

$$\begin{aligned} e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathsf{try}\ e_1\ \mathsf{catch}\ x.\, e_2 \mid \mathsf{raise}\ e \\ v &::= \lambda x.\, e \end{aligned}$$

The construct $\mathsf{try}\ e_1\ \mathsf{catch}\ x.\, e_2$ tries to evaluate e_1. If e_1 does not raise an exception, then $\mathsf{try}\ e_1\ \mathsf{catch}\ x.\, e_2$ evaluates to whatever e_1 evaluates to. If e_2 raises an exception with value v, then we evaluate e_2 with x bound to v, and the whole expression $\mathsf{try}\ e_1\ \mathsf{catch}\ x.\, e_2$ evaluates to whatever $e_2\{v/x\}$ evaluates to.

The construct $\mathsf{raise}\ e$ evaluates e to a value v, and raises an exception with value v.

For example, the program $\mathsf{try}\ \mathsf{raise}\ 30 + 5\ \mathsf{catch}\ x.\ x + 7$ evaluates to 42, and the program $\mathsf{try}\ 30 + 5\ \mathsf{catch}\ x.\ x + 7$ evaluates 35. Finally, the program $\mathsf{try}\ (\mathsf{try}\ \mathsf{raise}\ 7\ \mathsf{catch}\ x.\ x + 1)\ \mathsf{catch}\ y.\ y + 7$ evaluates to 8.

1.1.1 Operational semantics

We will specify an operational semantics for this language that when it encounters a $\mathsf{raise}\ v$ expression, will propagate that expression upwards in the evaluation contexts until it encounters an enclosing try construct.

$$E ::= E\ e \mid v\ E \mid \mathsf{raise}\ E$$

Note that there is no evaluation context $\mathsf{try}\ E\ \mathsf{catch}\ x.\ e$. Instead, we will explicitly deal with the evaluation of a try construct, so that if it raises an exception we can evaluate the handler.

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

$$\text{TRY-BODY}\ \frac{e_1 \longrightarrow e_1'}{\mathsf{try}\ e_1\ \mathsf{catch}\ x.\ e_2 \longrightarrow \mathsf{try}\ e_1'\ \mathsf{catch}\ x.\ e_2} \qquad \text{TRY-NORMAL}\ \frac{}{\mathsf{try}\ v\ \mathsf{catch}\ x.\ e_2 \longrightarrow v}$$

$$\text{TRY-EXCEPTION}\ \frac{}{\mathsf{try}\ \mathsf{raise}\ v\ \mathsf{catch}\ x.\ e \longrightarrow e\{v/x\}} \qquad \text{RAISE}\ \frac{}{E[\mathsf{raise}\ v] \longrightarrow \mathsf{raise}\ v}$$

1.1.2 Translational semantics

We give a translational semantics for the language using continuation passing style. The translation $\mathcal{CPS}[\![e]\!]$ of an expression e will take two continuations as arguments: the continuation k will be the "normal continuation," to be used when e does not raise an exception; the continuation h will be the "exception-handler continuation," and should be used when an exception is raised.

$$\begin{aligned}
\mathcal{CPS}[\![x]\!]kh &= k\ x \\
\mathcal{CPS}[\![\lambda x.\, e]\!]kh &= k\ (\lambda x, k', h'.\, \mathcal{CPS}[\![e]\!]k'h') && k', h' \text{ are not free variables of } e \\
\mathcal{CPS}[\![e_1\ e_2]\!]kh &= \mathcal{CPS}[\![e_1]\!]\ (\lambda f.\, \mathcal{CPS}[\![e_2]\!]\ (\lambda v.\, f\ v\ k\ h)\ h)\ h && f \text{ is not a free variable of } e_2
\end{aligned}$$

$$\begin{aligned}
\mathcal{CPS}[\![\mathsf{try}\ e_1\ \mathsf{catch}\ x.\, e_2]\!]kh &= \mathcal{CPS}[\![e_1]\!]\ k\ (\lambda x.\, \mathcal{CPS}[\![e_2]\!]kh) \\
\mathcal{CPS}[\![\mathsf{raise}\ e]\!]kh &= \mathcal{CPS}[\![e]\!]\ h\ h
\end{aligned}$$

1.2 Coroutines

Coroutines are non-preemptive threads that do not execute at the same time, and can transfer control and data to each other. Coroutines have several uses, including implementing an actor model of computation, and implementing iterators, where one coroutine produces values from a collection that the other coroutine consumes.

We add two new language constructs: $\mathsf{cobegin}\ e_1 \mid\mid x.\ e_2$ and $\mathsf{yield}\ e$. The syntax of the new language is given by the following grammar.

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid \mathsf{cobegin}\ e_1 \mid\mid x.\ e_2 \mid \mathsf{yield}\ e$$

A cobegin expression $\mathsf{cobegin}\ e_1 \mid\mid x.\ e_2$ starts a pair of coroutines that evaluate e_1 and e_2 in parallel. When either one of the expressions finishes evaluating, the whole cobegin expression terminates with the value of that expression. Control is initially given to the first expression e_1. Each coroutine evaluates uninterrupted until it evaluates a $\mathsf{yield}\ v$ expression, at which point, control is transferred to the other coroutine, with the value v being used as the result of the coroutine last yield. For the first time that control is given to e_2, the variable x takes on the value of the yielded expression.

Before giving a translational semantics, let's consider an example.

$$\begin{array}{l}
\mathsf{cobegin} \\
\quad \mathsf{let}\ i = \mathsf{yield}\ 1\ \mathsf{in}\ \mathsf{let}\ j = \mathsf{yield}\ (3+i)\ \mathsf{in}\ \mathsf{yield}\ (7+j) \\
\mid\mid \\
\quad x.\ \mathsf{yield}\ (x+1)
\end{array}$$

Control starts in the first coroutine, which yields the value 1; this value replaces variable x in the second coroutine, which then yields $1 + 1 = 2$ back to the first coroutine. That value is bound to i, and the first coroutine yields $3 + i = 5$, which is the result of the yield expression of the second coroutine. The second coroutine then terminates with the value 5.

Here's another example. It evaluates to 42.

$$\begin{array}{l}
\mathsf{cobegin} \\
\quad \mathsf{yield}\ (33 + (\mathsf{yield}\ 1)) \\
\mid\mid \\
\quad x.\ 1 + \mathsf{yield}\ (\mathsf{cobegin}\ \mathsf{yield}\ 7 \mid\mid y.\ y + x)
\end{array}$$

We define a CPS translational semantics for coroutines. The target language is the pure CBV lambda calculus. We slightly change the notion of continuation for the purposes of this translation. Previously, a continuation took a result, and continued with the rest of the computation; now, a continuation takes a result and a *yield continuation* and continues with the rest of the computation. A yield continuation is the continuation to yield a value to in the rest of the computation. We need to pass a yield translation to the continuation since the continuation to use for yield expressions changes as the program executes, and is not determined by the lexical scope (as was the case with exception handling above).

The translation $\mathcal{CPS}[\![e]\!]$ of an expression e will take two continuations as arguments: the continuation k will be the normal continuation, and continuation y will be the yield continuation: the continuation to pass yield results to.

The translation of abstractions and applications is similar to their translation in the language with exceptions, given above.

$$\begin{aligned}
\mathcal{T}[\![x]\!]ky &= k\ x\ y \\
\mathcal{T}[\![\lambda x.\, e]\!]ky &= k\ (\lambda x, k', y'.\, \mathcal{T}[\![e]\!]k'y')\ y && k', y' \text{ are not free variables of } e \\
\mathcal{T}[\![e_1\ e_2]\!]ky &= \mathcal{T}[\![e_1]\!]\ (\lambda f, y'.\, \mathcal{T}[\![e_2]\!]\ (\lambda v, y''.\, f\ v\ k\ y'')\ y')\ y && f, y' \text{ are not free variables of } e_2
\end{aligned}$$

$$\begin{aligned}
\mathcal{T}[\![\mathsf{yield}\ e]\!]ky &= \mathcal{T}[\![e]\!]\ (\lambda v, y'.\, y'\ v\ k)\ y \\
\mathcal{T}[\![\mathsf{cobegin}\ e_1 \mid\mid x.\ e_2]\!]ky &= \mathcal{T}[\![e_1]\!]\ (\lambda v, y'.\, k\ v\ y)\ (\lambda x, y''.\, \mathcal{T}[\![e_2]\!]\ (\lambda v, y'''.\, k\ v\ y)\ y'') && y'' \text{ is not a free variable of } e_2
\end{aligned}$$

The translation of $\mathsf{yield}\ e$ evaluates e, and the normal continuation given to $\mathcal{T}[\![\mathsf{yield}\ e]\!]$ takes the result v, a yield continuation y', and passes v to the yield continuation y'. The continuation y' also expects a yield continuation, and we give it the normal continuation k; that is, if the coroutine yields a value back to us, we should continue with the computation k. The yield continuation given to $\mathcal{T}[\![\mathsf{yield}\ e]\!]$ is y; this is because if e itself performs a yield, then the currently inactive coroutine should receive that value.

The translation of $\mathsf{cobegin}\ e_1 \mid\mid x.\ e_2$ evaluates e_1 with a yield continuation that starts evaluation e_2. Note that if e_1 yields a value to the coroutine e_2, then it also gives a yield continuation, which will be given the first value (if any) that e_2 yields.

2 Scope

So far in the operational semantics for the lambda calculus, we have used substitution when applying functions to expressions: replacing free occurrences of variables in the function body with the actual arguments. An alternative semantics is to maintain an *environment* that maps variables to values. When we want the value of a variable we look it up in the environment; when we apply a function, we extend the environment with the new value for the variable.

We can define a translational semantics that makes the environment explicit. Note that we write "x" for the name of the variable x; this could be a string representation of the variable, an integer (where each variable has a distinct integer).

$$\begin{aligned}
\mathcal{D}[\![x]\!]\rho &= \rho\ \text{“}x\text{”} \\
\mathcal{D}[\![\lambda x.\, e]\!]\rho &= \lambda y.\, \mathcal{D}[\![e]\!](\mathsf{extend}\ \rho\ \text{“}x\text{”}\ y) \\
\mathcal{D}[\![e_1\ e_2]\!]\rho &= (\mathcal{D}[\![e_1]\!]\rho)\ (\mathcal{D}[\![e_2]\!]\rho) \\
\mathsf{extend} &= \lambda \rho.\, \lambda s.\, \lambda v.\, \lambda x.\, \mathsf{if}\ s = x\ \mathsf{then}\ v\ \mathsf{else}\ \rho\ x
\end{aligned}$$

Note that when we want the value of a variable, we look it up in the environment. For a function, when the function is applied to an actual argument, we extend the environment, mapping the function's variable to the actual argument. Note that the environment that is extended is the environment in use *when the function was defined*. This is called *lexical scoping* (and also *static scoping* and *block-structured scoping*: the value of a free variable in a function body is determined by where the function was defined. That is, the binding of variables is determined statically, based on the lexical structure of the program.

By making the environment explicit, we can consider another kind of scoping: *dynamic scoping*. Under dynamic scoping, the value of a free variable in a function body is read from the environment is use *when the function is applied*. The binding of variables is determined by the runtime behavior of the program.

The following program demonstrates the difference between lexical and dynamic scoping.

$$\begin{aligned}&\text{let } x = 10 \text{ in}\\&\text{let } f = \lambda y.\, x + 1 \text{ in}\\&\text{let } a = f\ 0 \text{ in}\\&\text{let } x = 20 \text{ in}\\&\text{let } b = f\ 0 \text{ in } b\end{aligned}$$

The function contains a free variable x. Under lexical scoping, the environment used to determine the value of x is the environment in which the function was defined; thus, under lexical scoping, applying f will always produce the same result, and both a and b will have the value 11. By contrast, under dynamic scoping, the environment used to determine the value of x will be the environment in use when f is applied. Under dynamic scoping, a will equal 11, but b will equal 21.

We define a translation for dynamic scoping as follows. Note that the translation of functions now takes two arguments: the actual argument, and the dynamic environment, the environment in use when the function is applied. It is this dynamic environment that is extended with the new value for the function's variable.

$$\begin{aligned}\mathcal{D}[\![x]\!]\rho &= \rho\ \text{“}x\text{”}\\\mathcal{D}[\![\lambda x.\, e]\!]\rho_{lex} &= \lambda y, \rho_{dyn}.\, \mathcal{D}[\![e]\!](\text{extend } \rho_{dyn}\ \text{“}x\text{”}\ y)\\\mathcal{D}[\![e_1\ e_2]\!]\rho &= (\mathcal{D}[\![e_1]\!]\rho)\ (\mathcal{D}[\![e_2]\!]\rho)\ \rho\end{aligned}$$