Continuations are great for talking about semantics, but they are also useful more concretely, for programming and for compilation. We look at some of these uses here.

1 First-class continuations

Some languages expose continuations as first-class values. Examples of such languages include Scheme and SML/NJ. In the latter, there is a module that a continuation type α **cont**, representing a continuation expecting a value of type α. There are two functions for manipulating continuations:

callcc: $(\alpha$ **cont**$\rightarrow\alpha)\rightarrow\alpha$

(**callcc** f) passes the current continuation to the function f.

throw: α **cont**$\rightarrow\alpha\rightarrow\beta$

(**throw** k v) sends the value v to the continuation k.

Because callcc passes the current continuation, corresponding to the evaluation context of the callcc itself, invocation of the passed continuation makes the callcc expression itself seem to return again. It's up to the evaluation context of the callcc to decide whether it's seeing the original return from f or a later invocation of the passed continuation.

1.1 Semantics of first-class continuations

Using the translation approach we introduced earlier, we can easily describe these mechanisms. Suppose we represent a continuation value for the continuation k by tagging it with the integer 7. Then we can translate **callcc** and **throw** as follows:

$$\begin{aligned} [\![\textbf{callcc}\ e]\!]\rho k &= [\![e]\!]\rho(\mathit{CHECK\text{-}FUN}(\lambda f.\ f\ (7,k)\ k)) \\ [\![\textbf{throw}\ e_1\ e_2]\!]\rho k &= [\![e_1]\!]\rho(\mathit{CHECK\text{-}CONT}(\lambda k'.\ [\![e_2]\!]\rho k')) \end{aligned}$$

The key to the added power is the non-linear use of k in the **callcc** rule. This allows k to be reused any number of times.

1.2 Implementing threads with continuations

Once we have first-class continuations, we can use them to implement all the different control structures we might want. We can even use them to implement (non-preemptive) threads, as in the following SML/NJ-like code that explains how Concurrent ML (CML) is implemented:

```
type thread = unit cont
ready: thread queue = new_queue (* a mutable FIFO queue *)
enqueue(t) = insert ready t
dispatch() = throw (dequeue ready) ()
spawn(f: unit→unit): unit =
  callcc( fn(k) ⇒ (enqueue k; f(); dispatch()))
yield(): unit = callcc (fn(k) ⇒ enqueue k; dispatch())
```

The interface to threads is the functions **spawn** and **yield**. The **spawn** function expects a function **f** containing the work to be done in the newly spawned thread. The **yield** function causes the current thread to relinquish control to the next thread on the ready queue. Control also transfers to a new thread when one thread finishes evaluating. To complete the implementation of this thread package, we just need a queue implementation. CML has preemptive threads, in which threads implicitly yield automatically after a certain amount of time; this requires just a little help from the operating system.

2 Compiling with continuations

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 a stripped-down version of uML to a language similar to assembly.

The result of doing such a translation is that we will have a fairly complete recipe for compiling any of the languages we have talked about into the class down to the hardware.

3 Source language

Our source language looks like the lambda calculus with tuples and numbers, with the obvious (call-by-value) semantics:

$$e \quad ::= \quad n \;\mid\; x \;\mid\; \lambda x.e \;\mid\; e_0\ e_1 \;\mid\; (e_0, e_1, \ldots, e_n) \;\mid\; (\#n\ e) \;\mid\; e_0 + e_1$$

The target language looks more like assembly language:

$$\begin{array}{rcl} p & ::= & bb_1; bb_2; \ldots; bb_n \\ bb & ::= & lb : c_1; c_2; \ldots; c_n; \textbf{jump}\ x \\ c & ::= & \textbf{mov}\ x_1, x_2 \\ & \mid & \textbf{mov}\ x, n \\ & \mid & \textbf{mov}\ x, lb \\ & \mid & \textbf{add}\ x_1, x_2, x_3 \\ & \mid & \textbf{load}\ x_1, x_2[n] \\ & \mid & \textbf{cons}\ x_0, x_1, \ldots, x_n \end{array}$$

A program p consists of a series of *basic blocks* bb, each with a distinct label lb. Each basic block contains a sequence of commands c, and ends with a jump instruction. Commands correspond to assembly language instructions and are largely self-evident; the only one that is high-level is the **cons** instruction, which allocates n words of space, places the address of the space into x_0, and puts the contents of variables (registers) $x_1 \ldots x_n$ into those words.

We don't need a more general **store** instruction because the language is functional. And we aren't worrying about memory management. The **jump** instruction is an indirect jump. It makes the program counter take the value of the argument register.

4 Intermediate language 1

The first intermediate language is in continuation-passing style:

$$\begin{array}{rcl} v & ::= & n \;\mid\; x \;\mid\; \lambda k x.c \;\mid\; \textbf{halt} \\ & \mid & \overline{\lambda} x.c \\ e & ::= & v \;\mid\; v_0 + v_1 \;\mid\; (v_1, v_2, \ldots, v_n) \;\mid\; (\#n\ v) \\ c & ::= & \textbf{let}\ x = e\ \textbf{in}\ c \\ & \mid & v_0\ v_1\ v_2 \\ & \mid & v_0\ v_1 \end{array}$$

Some things to note about the intermediate language:

- Lambda abstractions corresponding to continuations are marked with a overline. These are considered *administrative* lambdas that we will eliminate at compile time, either by reducing them or by converting them to real lambdas.

- There are no subexpressions in the language (e does not occur in its own definition).
- Commands c look a lot like basic blocks:

$$\begin{array}{l}\textbf{let } x_1 = e_1 \textbf{ in} \\ \quad \textbf{let } x_2 = e_2 \textbf{ in} \\ \qquad \ddots \\ \qquad\quad \textbf{let } x_n = e_n \textbf{ in} \\ \qquad\qquad v_0\ v_1\ v_2\end{array}$$

- Lambdas are not closed and can occur inside other lambdas.

The contract of the translation is that $[\![e]\!]k$ will evaluate e and pass its result to the continuation k. To translate an entire program, we use $k = \textbf{halt}$, where **halt** is the continuation to send the result of the entire program to. Here is the translation from the source to the first intermediate language:

$$\begin{array}{rcl}
[\![x]\!]k & = & k\ x \\
[\![n]\!]k & = & k\ n \\
[\![\lambda x.\, e]\!]k & = & k\ (\lambda x k'.([\![e]\!]\ k')) \\
[\![e_0\ e_1]\!]k & = & [\![e_0]\!]\Big(\overline{\lambda} f.[\![e_1]\!](\overline{\lambda} v.(f\ v\ k))\Big) \\
[\![(e_1, e_2, \ldots, e_n)]\!]k & = & [\![e_1]\!]\Big(\overline{\lambda} x_1. \ldots [\![e_n]\!](\overline{\lambda} x_n.\ \textbf{let } t = (x_1, x_2, \ldots, x_n) \textbf{ in } (k\ t))\Big) \\
[\![\#n\ e]\!]k & = & [\![e]\!](\overline{\lambda} t.\ \textbf{let } y = \#n\ t \textbf{ in } (k\ y)) \\
[\![(e_1 + e_2)]\!]k & = & [\![e_1]\!](\overline{\lambda} x_1.[\![e_2]\!](\overline{\lambda} x_2.\ \textbf{let } z = x_1 + x_2 \textbf{ in } (k\ z)))
\end{array}$$

Let's see an example. We translate the expression $[\![(\lambda a.(\#1\ a))\ (3,4)]\!]k$, using $k = \textbf{halt}$.

$$\begin{array}{rcl}
 & & [\![(\lambda a.(\#1\ a))\ (3,4)]\!]\ k \\
= & & [\![\lambda a.(\#1\ a)]\!]\ (\overline{\lambda} f.[\![(3,4)]\!](\overline{\lambda} v.(f\ v\ k))) \\
= & & (\overline{\lambda} f.[\![(3,4)]\!](\overline{\lambda} v.(f\ v\ k)))\ (\lambda a k'.[\![\#1\ a]\!]\ k') \\
= & & (\overline{\lambda} f.[\![3]\!]\Big(\overline{\lambda} x_1.[\![4]\!](\overline{\lambda} x_2.\ \textbf{let } b = (x_1, x_2) \textbf{ in } (\overline{\lambda} v.(f\ v\ k))\ b)\Big))\ (\lambda a k'.[\![\#1\ a]\!]\ k') \\
= & & (\overline{\lambda} f.\Big(\overline{\lambda} x_1.(\overline{\lambda} x_2.\ \textbf{let } b = (x_1, x_2) \textbf{ in } (\overline{\lambda} v.(f\ v\ k))\ b)\ 4\Big)\ 3)\ (\lambda a k'.[\![\#1\ a]\!]\ k') \\
= & & (\overline{\lambda} f.\Big(\overline{\lambda} x_1.(\overline{\lambda} x_2.\ \textbf{let } b = (x_1, x_2) \textbf{ in } (\overline{\lambda} v.(f\ v\ k))\ b)\ 4\Big)\ 3)\ (\lambda a k'.[\![a]\!](\overline{\lambda} t.\ \textbf{let } y = \#1\ t \textbf{ in } k'\ t))
\end{array}$$

Clearly, the translation generates a lot of administrative lambdas, which will be quite expensive if they are compiled into machine code. To make the code more efficient and compact, we will optimize it using some simple rewriting rules to eliminate administrative lambdas.

$\overline{\beta}$-Reduction

We can eliminate unnecessary application to a variable, by copy propagation:

$$(\overline{\lambda} x.e)\ y \longrightarrow e\{y/x\}$$

Other unnecessary administrative lambdas can be converted into **let**s:

$$(\overline{\lambda} x.c)v \longrightarrow \textbf{let } x = v \textbf{ in } c$$

We can also perform administrative η-reductions:

$$\overline{\lambda} x.k\ x \longrightarrow k$$

If we apply these rules to the expression above, we get

$$
\begin{array}{l}
\textbf{let } f = (\lambda k' a.\, \textbf{let } y = \#1\ a \textbf{ in } k'\ y) \textbf{ in} \\
\quad \textbf{let } x_1 = \textbf{3} \textbf{ in} \\
\quad\quad \textbf{let } x_2 = \textbf{4} \textbf{ in} \\
\quad\quad\quad \textbf{let } x_3 = (x_1,\ x_2) \textbf{ in} \\
\quad\quad\quad\quad f\ b\ k
\end{array}
$$

This is starting to look a lot more like our target language.

The idea of separating administrative terms from real terms and performing a compile-time *partial evaluation* is powerful and can be used in many other contexts. Here, it allows us to write a very simple CPS conversion that treats all continuations uniformly, and perform a number of control optimizations.

Note that we may not be able to remove all administrative lambdas. Any that cannot be reduced using the rules above are converted into real lambdas.

4.1 Tail call optimization *

A tail call is a function call that determines the result of another function. A tail-recursive function is one whose recursive calls are all tail calls. Continuations make tail calls easy to optimize. For example, the following program has a tail call from f to g:

$$
\begin{array}{l}
\textbf{let } g = \lambda x. \#1\ x \textbf{ in} \\
\quad \textbf{let } g = \lambda x. g\ x \\
\quad\quad \textbf{in} \\
\quad\quad\quad f(2)
\end{array}
$$

The translation of the body of f is $(g\ (\overline{\lambda y}.k'\ y)\ x)$, which permits optimization by η-reduction to $(g\ k\ x)$. In this optimized code, g does not bother to return to f, but rather jumps directly back to f's caller. This is an important optimization for functional programming languages, where tail-recursive calls that take up linear stack space are converted into loops that take up constant stack space.

5 Intermediate Language 1 → Intermediate Language 2

The next step is the translation from Intermediate language 1 to Intermediate Language 2. In this intermediate language, all lambdas are at the top level, with no nesting:

$$
\begin{array}{rcl}
P & ::= & \textbf{let } x_f = \lambda k x_1 \ldots x_n.\, c \textbf{ in } P \\
 & & \mid \textbf{let } x_c = \lambda x_1 \ldots x_n.\, c \textbf{ in } P \\
 & & \mid c \\
c & ::= & \textbf{let } x = e \textbf{ in } c \ \mid\ x_0\ x_1 \ldots\ x_n \\
e & ::= & n \ \mid\ x \ \mid\ \textbf{halt} \ \mid\ x_1 + x_2 \ \mid\ (x_1, x_2, \ldots, x_n) \ \mid\ \#n\ x
\end{array}
$$

The translation requires the construction of closures that capture all the free variables of the lambda abstractions in intermediate language 1. We have covered closure conversion earlier; it too can be phrased as a translation that exploits compile-time partial evaluation.

6 Intermediate Language 2 → Assembly

The translation is given below. Note: ra is the name of the dedicated register that holds the return address.

$$
\begin{aligned}
\mathcal{P}[\![p]\!] &= \textbf{program for p} \\
\mathcal{C}[\![c]\!] &= \textbf{sequence of commands } c_1; c_2; \ldots; c_n \\
\\
\mathcal{P}[\![s]\!] &= \textbf{main} : \mathcal{C}[\![c]\!]; \textbf{halt} : \\
\mathcal{P}[\![\textbf{let } x_f = \lambda k x_1 \ldots x_n.\, c \textbf{ in } p]\!] &= x_f : \textbf{mov } k, ra; \\
&\quad \textbf{mov } x_1, a_1; \\
&\quad \vdots \\
&\quad \textbf{mov } x_n, a_n; \\
&\quad \mathcal{C}[\![c]\!]; \\
&\quad \mathcal{P}[\![p]\!] \\
\mathcal{P}[\![\textbf{let } x_c = \lambda x_1 \ldots x_n.\, c \textbf{ in } p]\!] &= x_c : \textbf{mov } x_1, a_1; \\
&\quad \vdots \\
&\quad \textbf{mov } x_n, a_n; \\
&\quad \mathcal{C}[\![c]\!]; \\
&\quad \mathcal{P}[\![p]\!] \\
\mathcal{C}[\![\textbf{let } x_1 = x_2 \textbf{ in } c]\!] &= \textbf{mov } x_1, x_2; \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\textbf{let } x_1 = x_2 + x_3 \textbf{ in } c]\!] &= \textbf{add } x_1, x_2, x_3; \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\textbf{let } x_0 = (x_1, x_2, \ldots, x_n) \textbf{ in } c]\!] &= \textbf{cons } x_0, x_1, x_2, \ldots, x_n; \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![\textbf{let } x_1 = \#n\ x_2 \textbf{ in } c]\!] &= \textbf{load } x_1, x_2[n]; \mathcal{C}[\![c]\!] \\
\mathcal{C}[\![x_0\ k\ x_1\ \ldots\ x_n]\!] &= \textbf{mov } ra, k; \\
&\quad \textbf{mov } a_1, x_1; \\
&\quad \vdots \\
&\quad \textbf{mov } a_n, x_n; \\
&\quad \textbf{jump } x_0
\end{aligned}
$$

At this point, we are still assuming an infinite supply of registers. We need to do register allocation and possibly spill registers to a stack to obtain working code.

While this translation is very simple, it is possible to do a better job of generating calling code. For example, we are doing a lot of register moves when calling functions and when starting the function body. These could be optimized.