Continuations

Advanced Compiler Construction
Michel Schinz – 2013-05-16

Motivating example

`list-any?`

The `list-any?` function takes a predicate and a list as arguments and returns true iff the list contains at least one element satisfying the predicate.

Recursively, it can be defined as follows in L_3:

```
(defrec list-any?
  (fun (p l)
    (and (not (list-empty? l))
         (or (p (list-head l))
             (list-any? p (list-tail l))))))
```

Can we define `list-any?` using `list-for-each?`

`list-any?`

If we attempt to define `list-any?` in terms of `list-for-each`, we get stuck:

```
(def list-any?
  (fun (p l)
    (list-for-each (fun (e)
                     (if (p e) ???))
                   l)
    #f))
```

Question: would adding a `return` statement to L_3 be the solution?

Continuation passing style

To avoid changing the L_3 language, we can change the convention used to return from a function, as follows:

> All functions get an additional function as argument. To return, they invoke that function with their return value.

When a program has that form, it is said to be in **continuation passing style** (**CPS**). The additional function passed to all functions is called their continuation, because it describes how the program should continue after the function is finished.

Notice that in CPS, no function ever returns in the traditional sense, and all calls are tail calls!

`list-any?` in CPS

When the whole program is written in CPS, it becomes easy for `list-any?` to return immediately to its caller by invoking its continuation!

```
(def list-any?CPS
  (fun (pCPS l any-k)
    (list-for-eachCPS
     (fun (e body-k)
       (pCPS e (fun (r)
                 (if r
                     (any-k #t)
                     (body-k #u)))))
     l
     (fun (ignored) (any-k #f)))))
```

Immediately return from `list-any?`

Unfortunately, the program has become difficult to read...

Exercise

We have seen that in programs that are in CPS, all calls are tail calls. As a simple example, observe that all of the five calls in `list-any?`$_{\text{CPS}}$ are indeed in tail position.

We have also seen earlier that tail calls can be compiled so that they do not consume any stack space.

Therefore, by transforming a program to CPS, we can obtain a version that does not use any stack at all!

Where did the stack go?

Continuations

Ideally, we should be able to obtain the current continuation at any time without having to write the program in CPS ourselves...

In languages providing that feature, the primitive to obtain the current continuation is generally called `call-with-current-continuation` or `call/cc`.

As its name implies, `call/cc` takes a function and invokes it with the current continuation as argument.

`list-any?` with `call/cc`

With `call/cc`, `list-any?` can be rewritten as follows:

```
(def list-any?
  (fun (p l)
    (call/cc
     (fun (any-k)
       (list-for-each
        (fun (e) (if (p e) (any-k #t)))
        l)
       #f))))
```

Here, `call/cc` enables us to get the only continuation we are really interested in: that of `list-any?`. And we can obtain it without having to transform the whole program to CPS by hand.

Advanced uses of continuations

Generators

Continuations can also be used to implement more sophisticated control structures, like generators (a.k.a iterators) in the style of Python or C#.

Trivial example use:

```
(def ping-pong
  (generator
    (fun (gen)
      (generator-yield gen "ping")
      (generator-yield gen "pong"))))
(generator-next ping-pong) ⇒ "ping"

(generator-next ping-pong) ⇒ "pong"

(generator-next ping-pong) ⇒ #u
```

Preliminaries: references

To implement generators, we make use of mutable references defined as follows:

```
(def ref-make
  (fun (init)
    (let ((r (@block-alloc-4 1)))
      (@block-set! r 0 init)
      r)))
(def ref-get
  (fun (ref)
    (@block-get ref 0)))
(def ref-set!
  (fun (ref new-value)
    (@block-set! ref 0 new-value)))
(def ref-swap!
  (fun (ref new-value)
    (let ((old-value (ref-get ref)))
      (ref-set! ref new-value)
      old-value)))
```

Generators

A generator is a reference containing alternatively the continuation of the generator function – when the main program is executing – and the continuation of the main program – when the generator function is executing.

```
(def swap-control
  (fun (k-ref val)
    (call/cc
     (fun (k)
       ((ref-swap! k-ref k)
        val)))))

(def swap-for-ever
  (fun (gen)
    (swap-control gen #u)
    (swap-for-ever gen)))
```

```
(def generator
  (fun (f)
    (ref-make
     (fun (gen)
       (f gen)
       (swap-for-ever gen)))))

(def generator-yield
  swap-control)

(def generator-next
  (fun (gen)
    (swap-control gen gen)))
```

And more...

Continuations are the ultimate control operator. Using them, one can implement:

- all forms of local and non-local `goto`s (i.e. the equivalent of Java's `break`, `continue`, `goto` and `return`, as well as the equivalent of C's `setjmp` and `longjmp`),
- exceptions,
- generators,
- coroutines,
- threads,
- etc.

Implementing continuations

Implementing continuations

There are basically two techniques to implement continuations:

1. The high-level technique: the compiler transforms the whole program to CPS, in which `call/cc` can be trivially defined.
2. The low-level technique: `call/cc` is implemented at the level of the target architecture, by manipulating the call stack.

We will look at each in turn.

Transformation to CPS

Basic CPS transformation

The basic transformation works by turning all expressions of the source program into functions expecting the current continuation as argument.

That continuation is applied to the value of the expression.

Basic CPS for μCL$_3$

To illustrate the basic CPS transformation, here are the rules for a subset of CL3 called μCL3 and composed only of variables, function abstraction and function application:

⟦n⟧ *where* n *is a name* =

```
   (fun (k) (k n))
```

⟦(fun (n$_1$ n$_2$ ...) e)⟧ =

```
   (fun (k) (k (fun (n1 n2 ... k2) (⟦e⟧ k2))))
```

⟦(e e$_1$ e$_2$ ...)⟧ =

```
   (fun (k)
     (⟦e⟧ (fun (f)
            (⟦e1⟧ (fun (v1) ... (f v1 ... k))))))
```

(As usual, underlined names are fresh).

Administrative redexes

The basic translation to CPS produces a lot of reducible expressions (a.k.a. redexes) that did not appear in the original term. They are known as **administrative redexes**. The example translation below produces four such redexes, which are numbered:

```
⟦(g x)⟧ =
  (fun (k1)
    ((fun (k2) (k2 g)¹)
     (fun (f)
       ((fun (k3) (k3 x)²)
        (fun (v1) (f v1 k1))³))⁴))
```

Reducing them produces the simpler term:

```
(fun (k1) (g x k1))
```

Optimized CPS transform.

The optimized CPS transformation avoids the creation of administrative redexes. To do that, it takes an additional argument, the context in which the translated term has to appear.

Plugging the hole of that context with a term plays a role similar to the application of an administrative redex in the simple CPS transformation. However, since the plugging happens during the transformation, no administrative redex is created.

Optimized CPS for μCL_3

The optimized CPS translation for μCL_3 looks as follows:

$[\![$`n`$]\!]\ C$ *where* `n` *is a name* =

$C\{$`n`$\}$

$[\![$`(fun (`$n_1\ n_2 \ldots$`) e)`$]\!]\ C =$

$C\{$`(fun (`$n_1\ n_2 \ldots k$`) (`$[\![e]\!](\lambda r$ `(k r)))`$\}$

$[\![$`(e `$e_1\ e_2 \ldots$`)`$]\!]\ C =$

$[\![e]\!](\lambda f\ ([\![e_1]\!]\ (\lambda v_1 \ldots$ `(f `$v_1 \ldots$ `(fun (r) C{r}))`$)))$

Applied to the earlier example, this optimized translation produces no administrative redexes, as expected:

$[\![$`(g x)`$]\!]\ C =$ `(g x (fun (r) C{r}))`

Defining `call/cc`

The CPS version of `call/cc`, i.e. $\texttt{call/cc}_{\text{cps}}$, can be defined easily:

```
(def call/cc_cps
  (fun (f k)
    (f (fun (res ignored-k) (k res))
       k)))
```

reified continuation

Notice how the reified continuation ignores its own continuation (`ignored-k`) and invokes the captured one (`k`) instead.

Stack manipulation

Continuations as call stacks

A continuation represents a call stack:

- Obtaining a continuation amounts to obtaining a handle on the current call stack.
- Invoking a continuation amounts to replacing the current call stack with that of the continuation.

It is therefore possible to implement continuations as manipulations of the call stack, provided that the target architecture allows it.

(This is often not possible: neither high-level VMs like the JVM, nor low-level languages like C offer direct manipulation of call stacks.)

Stacks and continuations

In languages without first-class continuations, function calls and returns happen in LIFO order. Thus, activation frames can be allocated on a stack and not on the heap, which is typically more efficient.

In a language with first-class continuations, this is no longer true! Activation frames must then either be allocated on the garbage-collected heap, or the stack must be copied when the current continuation is obtained.

As an example, the L_3VM allocates all activation frames on the heap, which means that implementing first class continuations on it is relatively simple.

Continuations in the L_3 compiler

Continuations in CPS/L_3

CPS/L_3 also offers a notion of continuations, but unlike the continuations we have examined here, they are not first class! That is, they can neither be stored into variables, nor passed as arguments to functions, except for the return continuation.

These restrictions mean that the continuations of CPS/L_3 cannot be used to implement `call/cc` for L_3. On the other hand, CPS/L_3 continuations can be compiled efficiently, as they represent simple code blocks with arguments.

The translation from CL_3 to CPS/L_3 performed by the L_3 compiler is an adaptation of the optimized CPS translation we examined earlier.