Chapter 21

Continuations and Compilation: Explicating the Stack

We have already seen one important application of continuations, namely to the problem of improving the structure of programs running over stateless protocols. But continuations and CPS long predate the Web; they have long since been used in the gut of compilers. We now study this use in considerably more detail. First, we will study how to make the stack, which is currently represented procedurally, more recognizable as a data structure.

21.1 Examples

The following examples demonstrate how we can employ the CPS representation of a program to make the stack an explicit data structure.

21.1.1 Factorial

Here’s an implementation of the factorial function:

```
(define fact
  (lambda (n)
    (if (= n 0)
      1
      (* n (fact (- n 1))))))
```

You should be able to convince yourself that the following is the same program in CPS:

```
(define fact/k
  (lambda (n k)
    (if (= n 0)
      (k 1)
      (fact/k (- n 1) (lambda (v) (k (* n v)))))))
```
(define fact
  (lambda (n)
    (fact/k n (lambda (x) x)))))

To make the stack more explicit, we’ll give specific names to the stack manipulatives and constant:

(define (fact/stack n stack)
  (if (zero? n)
      (Pop stack 1)
      (fact/stack (− n 1)
        (Push stack (lambda (val) (Pop stack (+ n val)))))))

(define (Pop stack value)
  (stack value))

(define (Push stack receiver)
  (lambda (v) (receiver v)))

(define (fact n) (fact/stack n EmptyStack))

(define (EmptyStack value)
  value)

Merely by assigning names to the same receivers, we’ve made it possible to change their representations without affecting the original factorial code: that is, we have gained a degree of representation independence. Because we know the stack is really a sequence of stack records, which is easy to represent as a list, we’ll use lists to denote stacks. Notice that we only need to change the receiver abstractions; the two factorial procedures (fact and fact/k) stay unchanged:

(define (Pop stack value)
  ((first stack) value))

(define (Push stack receiver)
  (cons receiver stack))

(define EmptyStack (cons (lambda (value) value) empty))

Now we have a version where the stacks themselves are lists, but the elements of the lists are still functions, which is unsatisfying as an account of the stack’s actual behavior (on hardware). Let’s instead use elements of a datatype to represent the stack frame types:

(define-type StackFrame
  [stack-rec-mult (n number?)] ;; a multiplication record
  [stack-rec-empty]) ;; the bottom of the stack

(define (Pop stack value)
  (local ((define top-rec (first stack))
    (type-case StackFrame top-rec
21.1. EXAMPLES

(stack-rec-mult \(n\) (Pop (rest stack) (* \(n\) value)))
(stack-rec-empty () value))

(define (Push stack new-record)
  (cons new-record stack))

(define EmptyStack (list (stack-rec-empty)))

(define (fact/stack/rec \(n\) stack)
  (if (zero? \(n\))
    (Pop stack 1)
    (fact/stack/rec (\(-\) \(n\) 1) (Push stack (stack-rec-mult \(n\))))))

(define (fact \(n\)) (fact/stack/rec \(n\) EmptyStack))

Notice how converting the program into CPS and transforming the representation has slowly taken us closer and closer to the very program a compiler might generate! This is a promising direction to consider.

21.1.2 Tree Sum

Now that we're done warming up, let's consider an interesting example: a procedure that consumes a tree of numbers and sums all the numbers in the tree. The source program is straightforward:

(define-type Tree
  [empty-tree]
  [node (n number?)
    (left Tree?)
    (right Tree?)])

(define (tree-sum atree)
  (type-Case Tree atree
    [empty-tree () 0]
    [node (n left right) (+ n
      (tree-sum left)
      (tree-sum right))]))

Converting it to CPS:

(define (tree-sum/k atree k)
  (type-case Tree atree
    [empty-tree () (k 0)]
    [node (n left right) (lambda (lv)
      (tree-sum/k left (lambda (rv)
        (k (+ n lv rv))))))))
(define (tree-sum atree)
  (tree-sum/k atree (lambda (x) x)))

Applying the same transformations as before gives us this:

(define-type StackFrame
  [rec-bottom]
  [rec-add-left (node-val number?)
    (right-tree Tree?)
  ]
  [rec-add-right (node-val number?)
    (left-value number?)])

(define (tree-sum/rec atree stack)
  (type-case Tree atree
    [empty-tree () (Pop stack 0)]
    [node (n left right)
      (tree-sum/rec left (Push stack (rec-add-left n right))))])

(define (Pop stack value)
  (local ([define top-rec (first stack)])
    (type-case StackFrame top-rec
      [rec-bottom () value]
      [rec-add-left (node-val right)
        (tree-sum/rec right (Push (rest stack)
          (rec-add-right node-val value)))]
      [rec-add-right (node-val lv) (Pop (rest stack) (+ node-val lv value))])))

(define (Push stack record)
  (cons record stack))

(define EmptyStack (list (rec-bottom)))

(define (tree-sum atree)
  (tree-sum/rec atree EmptyStack))

Note in particular that the Pop procedure here executes a Push. This is more typical of the general case—the pattern of recursion factorial is unusual in being strictly linear.

### 21.1.3 Filtering Positive Numbers

Let’s begin with the following program: a first-order version of filter that only retains positive numbers.

(define (filter-pos l)
  (cond
    [(empty? l) empty]
    [else
21.1. EXAMPLES

(\(\text{if (}> (\text{first} \ l) \ 0)\)
\(\ (\text{cons (first} \ l) (\text{filter-pos (rest} \ l)))\)
\(\ (\text{filter-pos (rest} \ l)))))\)

Its representation in CPS is

\(\text{(define (filter-pos/k} \ l \ k)\)
\(\ (\text{cond}\)
\(\ [\ (\text{empty?} \ l) \ (k \ \text{empty})]\)
\(\ [\ \text{else}\)
\(\ (\text{if (}> (\text{first} \ l) \ 0)\)
\(\ (\text{filter-pos/k} \ (\text{rest} \ l)\)
\(\ (\text{lambda} \ (v)\)
\(\ (k \ (\text{cons (first} \ l) \ v))))\)
\(\ (\text{filter-pos/k} \ (\text{rest} \ l) \ k))))\)

\(\text{(define (filter-pos} \ l)\)
\(\ (\text{filter-pos/k} \ l \ (\text{lambda} \ (x) \ x)))\)

The next thing to do is change the representation of the stack. We’ll proceed directly to a list of structures. First the type declaration for the different kinds of stack frames:

\(\text{(define-type StackFrame}\)
\(\ [\text{terminal-frame}]\)
\(\ [\text{filter-frame} \ (n \ \text{number?})]]\)

And now the code that uses this:

\(\text{(define (filter-pos/k} \ l \ \text{stack})\)
\(\ (\text{cond}\)
\(\ [\ (\text{empty?} \ l) \ (\text{Pop empty stack})]\)
\(\ [\ \text{else}\)
\(\ (\text{if (}> (\text{first} \ l) \ 0)\)
\(\ (\text{filter-pos/k} \ (\text{rest} \ l)\)
\(\ (\text{Push (filter-frame (first} \ l)) \ \text{stack})\))
\(\ (\text{filter-pos/k} \ (\text{rest} \ l) \ \text{stack}))))\)

\(\text{(define (filter-pos} \ l)\)
\(\ (\text{filter-pos/k} \ l \ \text{EmptyStack})))\)

\(\text{(define EmptyStack} \ \text{(list (terminal-frame)))}\)
\(\text{(define (Push frame stack) (cons frame stack)))}\)
\(\text{(define (Pop value stack)}\)
\(\ (\text{type-case StackFrame} \ (\text{first stack})\)
\(\ [\text{terminal-frame} \ () \ \text{value}]\)
\(\ [\text{filter-frame} \ (n) \ (\text{Pop (cons n value) (rest stack))))]})\)
21.2 Tail Calls

Study the code in \texttt{filter-pos/k} carefully. There are two recursive calls in that body, and they differ slightly. By reading the original version of the program (\texttt{filter-pos}), it’s easy to tell that the difference is whether or not we want to retain the first element of the list.

Now let’s look at the corresponding calls in the explicit stack version. The version that retains the first element becomes

\begin{verbatim}
(filter-pos/k (rest l)
  (Push (filter-frame (first l))
    stack))
\end{verbatim}

while the other version is

\begin{verbatim}
(filter-pos/k (rest l) stack))
\end{verbatim}

This suggests the following:

- The stack exists solely to evaluate arguments, not for making function calls. It so happens that in this case, the invocation of \texttt{filter-pos} is itself a mere “argument evaluation”—from the perspective of the pending \texttt{cons}.

- The actual function call is itself simply a direct jump to the code of the function—that is, it’s a “goto”.

- Converting the program to CPS helps us clearly see which calls are just gotos, and which ones need stack build-up. The ones that are just gotos are those invocations that use the same receiver argument as the one they received. Those that build a more complex receiver are relying on the stack.

Procedure calls that do not place any burden on the stack are called \textit{tail calls}. Converting a program to CPS helps us identify tail calls, though it’s possible to identify them from the program source itself. An invocation of \( g \) in a procedure \( f \) is a tail call if, in the control path that leads to the invocation of \( g \), the value of \( f \) is determined by the invocation of \( g \). In that case, \( g \) can send its value directly to whoever is expecting \( f \)’s value; this verbal description is captured precisely in the CPSed version (since \( f \) passes along its receiver to \( g \), which sends its value to that receiver). This insight is employed by compilers to perform \textit{tail call optimization} (abbreviated TCO, and sometimes referred to as \textit{last call optimization}), whereby they ensure that tail calls incur no stack growth.

Here are some consequences of TCO:

- With TCO, it no longer becomes necessary for a language to provide looping constructs. Whatever was previously written using a custom-purpose loop can now be written as a recursive procedure. So long as all recursive calls are tail calls, the compiler will convert the calls into gotos, accomplishing the same efficiency as the loop version. For instance, here’s a very simple version of a \texttt{for} loop, written using tail calls:

\begin{verbatim}
(define (for init condition change body result)
  (if (condition init)
    (for (change init))
    result))
\end{verbatim}
By factoring out the invariant arguments, we can write this more readably as

\[
\text{(define (for init condition change body result)}
\text{\local [(define (loop init result)}
\text{\if (condition init)
\text{\loop (change init)
\text{\body init result))
\text{\result))]
\text{\loop init result))))}
\]

To use this as a loop, write

\[
\text{(for 10 positive? sub1 + 0)}
\]

which evaluates to 55. It’s possible to make this look more like a traditional `for` loop using macros, which we will discuss in Section 33. In either case, notice how similar this is to a `fold` operator! Indeed, `foldl` employs a tail call in its recursion, meaning it is just as efficient as looping constructs in more traditional languages.

- Put differently, thanks to TCO, the set of looping constructs is extensible, not limited by the imagination of the language designer. In particular, it becomes easy to create loops (or iterators) over new data structures without suffering an undue performance penalty.

- While TCO is traditionally associated with languages such as Scheme and ML, there’s no reason they must be. It’s perfectly possible to have TCO in any language. Indeed, as our analysis above has demonstrated, TCO is the natural consequence of understanding the true meaning of function calls. A language that deprives you of TCO is cheating you of what is rightfully yours—stand up for your rights! Because so many language designers and implementors habitually mistreat their users by failing to support TCO, however, programmers have become conditioned to think of all function calls as inherently expensive, even when they are not.

- A special case of a tail call is known as tail recursion, which occurs when the tail call within a procedure is to the same procedure. This is the behavior we see in the procedure `for` above. Keep in mind, however, that tail recursion optimization is only a special case. While it is an important special case (since it enables the implementation of linear loops), it is neither the most interesting case nor, more importantly, the only useful one.

Sometimes, programmers will find it natural to split a computation across two procedures, and use tail
calls to communicate between them. This leads to very natural program structures. A programmer using a language like Java is, however, forced into an unpleasant decision. If they split code across methods, they pay the penalty of method invocations that use the stack needlessly. But even if they combine the code into a single procedure, it's not clear that they can easily turn the two code bodies into a single loop. Even if they do, the structure of the code has now been altered irrevocably. Consider the following example:

```
(define (even? n)
  (if (zero? n)
      true
      (odd? (sub1 n))))

(define (odd? n)
  (if (zero? n)
      false
      (even? (sub1 n))))
```

Try writing this entirely through loops!

Therefore, even if a language gives you tail recursion optimization, remember that you are getting less than you deserve. Indeed, it sometimes suggests an implementor who realized that the true nature of function calls permitted calls that consumed no new stack space but, due to ignorance or a failure of imagination, restricted this power to tail recursion only. The primitive you really want a language to support is tail call optimization. With it, you can express solutions more naturally, and can also build very interesting abstractions of control flow patterns.

- Note that CPS converts every program into a form where every call is a tail call!

**Exercise 21.2.1** If, in CPS, every call is a tail call, and the underlying language supports TCO (as Scheme does), does the CPS version of a program run in constant stack space even if the original does not? Discuss.

**Exercise 21.2.2** Java does not support TCO. Investigate why not.
**Hint:** Neither ignorance nor malice explains the original decision to not support TCO in Java.

### 21.3 On Stacks

As we have seen, the receiver corresponds directly to the stack. In particular, the receiver is a procedure that may refer to another procedure (that it closes over), which may refer to another procedure (that it closes over), and so on. Each of these procedures represents one stack frame (sometimes called an activation record, because it records an extant activation of a procedure in the running program). Returning a value

---

1They may not even communicate mutually. In the second version of the loop above, for invokes loop to initiate the loop. That call is a tail call, and well it should be, otherwise the entire loop will have consumed stack space. Because Scheme has tail calls, notice how effortlessly we were able to create this abstraction. If the language supported only tail recursion optimization, the latter version of the loop, which is more pleasant to read and maintain, would actually consume stack space against our will.
“pops” the stack; since we have made the stack explicit, the equivalent operation is to pass the value to be returned to the receiver. A procedure passes on its receiver without further embellishment precisely when it calls another procedure without pushing anything on to the stack.

We therefore see that the stack is used solely to store intermediate results. It plays no part in the actual invocation of a function. We can see this in the fact that some procedure invocations have no impact on the stack at all! This probably sets on its head everything you have been taught about stacks until now. This is an important and, perhaps, startling point:

**Stacks are not necessary for invoking functions.**

The stack only plays a role in evaluating the argument to the function; once that argument has been reduced to a value (in an eager language), the stack has no further role with respect to invoking a function. The actual invocation itself is merely a jump to an address holding the code of the function: it’s a “goto”.

Converting a program to CPS thus accomplishes two things. First, it exposes something—the stack—normally hidden during the evaluation process; this is an instance of reflection. The transformation also makes this a value that a programmer can manipulate directly (even changing the meaning of the program as a result); this is known as reification.

### Reflection and Reification

Reflection and reification are very powerful programming concepts. Most programmers encounter them only very informally or in a fairly weak setting. For instance, Java offers a very limited form of reflection (a programmer can, for instance, query the names of methods of an object), and some languages reify very low-level implementation details (such as memory addresses in C). Few languages reify truly powerful computational features; the ones that do enable entirely new programming patterns that programmers accustomed only to more traditional languages usually can’t imagine. Truly smart programmers sometimes create their own languages with the express purpose of reifying some useful hidden element, and implement their solution in the new language, to create a very powerful kind of reusable abstraction. A classic instance of this is Web programmers who have reified stacks to enable a more direct programming style.

## 21.4 Consolidation

We’ve seen a sequence of transformations: conversion to CPS; abstracting the representation of the stack; and converting the stack into a list of structures representing the activation records. The result of all this is a program quite different in both style and substance from the original. In style, it has changed its form considerably, while in substance, its representations are all much lower-level. As a result, we are slowly converting high-level Scheme into programs that can run in just about any low-level language. Hopefully you can see where this is going!

Notice what we **not** doing: we are not explicitly writing a compiler. Instead, we are showing you how a compiler would operate: each transformed program is the result of a transformation we would code into the compiler. Actually writing all those transformations is a little more work, and doing so would distract from the mission of this course; courses on compiler construction force you to confront the details that arise in implementing these steps.