Logic Column 8

Column Editor: Jon G. Riecke
Bell Laboratories, Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974
riecke@bell-labs.com

Continuations, functions and jumps*

Hayo Thielecke
ht@dcs.qmw.ac.uk
Department of Computer Science
Queen Mary and Westfield College, University of London
London E1 4NS, UK

1 Introduction

Practically all programming languages have some form of control structure or jumping. The more advanced forms of control structure tend to resemble function calls, so much so that they are usually not even described as jumps. Consider for example the library function `exit` in C. Its use is much like a function, in that it may be called with an argument; but the whole point of `exit` is of course that its behaviour is utterly non-functional, in that it jumps out of arbitrarily many surrounding blocks and pending function calls. Such a "non-returning function" or "jump with arguments" is an example of a *continuation* in the sense which we are interested in here.

On the other hand, a simple but fundamental idea in compiling is that a function call is broken down into two jumps: one from the caller to the callee for the call itself, and another jump back from the callee to the caller upon returning. (The `return` statement in C is in fact listed among the "jump statements" [5].) This is most obvious for `void`-accepting and -returning functions, but it generalizes to other functions as well, if one is willing to understand "jump" in the broader sense of jump with arguments, *i.e.* continuation.

In this view, then, continuations are everywhere. Continuations have been used in many different settings, in which they appear in different guises, ranging from mathematical functions to machine addresses. Rather than confine ourselves to a definition of what a continuation is, we will focus on continuation-passing style (CPS), as it brings out the commonalities. The CPS transform compresses a great deal of insight into three little equations in λ-calculus. Making sense of it intuitively, however, requires some background knowledge and a certain fluency. The purpose of this article, therefore, is to help the reader uncompress the CPS transform by way of a rational reconstruction from jumps.

Sample program

```
void p(char x)
{
    putchar(x);
}


main()
{
    p('a');
    p('b');
}
```

Step 1: `goto`

```
void p(char x,
       void *k)
{
    putchar(x);
    goto *k;
}

main()
{
    p('a', &&L1);
L1: p('b', &&L2);
L2: exit();
}
```

Step 2: non-returning functions

```
void p(char x,
       void (*k)())
{
    putchar(x);
    (*k)();
}

void L1(){p('b', &exit);}

main()
{
    p('a', &L1);
}
```

Figure 1: An example of van Wijngaarden's CPS transformation, applied to a C function `p`

In the sequel, we will attempt to illustrate the correspondence between continuations and jumps (even in the guise of the abhorred `goto`). The intent is partly historical, to retrace some of the analysis of jumps that led to the discoveries of continuations. At the same time, the language of choice for many researchers during the (pre)history of continuations, ALGOL 60, is not so different from today's mainstream languages (*i.e.* C); we hope that occasional snippets of C may be more easily accessible to many readers than a functional language would be. So in each of the four sections (Sections 2–5 below) that make up the body of this paper, some C code will be used to give a naive but concrete example of the issue under consideration, before generalizing to a more abstract setting.

2 Continuation-passing style

The basic idea of continuation-passing is (perhaps deceptively) simple. Just as universal computation could be realized in many different media, including fanciful ones such as a horde of dutiful clerks with finite notepads, one could imagine continuation-passing to be realized by a scheme of postal communication where every letter to which a reply is expected needs to enclose a self-addressed envelope for the reply. Here the self-addressed envelope implements a return continuation. In fact, *return addresses*, in both the non-technical and the programming language sense, are arguably the most approachable instances of continuations.

Slightly more technically, for continuation-passing style, a function call is transformed into a jump with arguments to the callee, such that one of the argument is a return address, *i.e.* a continuation that enables the callee to jump back to the caller. To match this, all function definitions need to be transformed to take the return address as an extra argument. The systematic addition of return addresses is described in an early paper by van Wijngaarden [17], which Reynolds [12] credits with the earliest use of continuation-passing style. Van Wijngaarden (cited in [12]) describes the introduction of continuation parameters thus:

> Provide each procedure declaration with an extra formal parameter — specified **label** — and insert at the end of its body a **goto** statement leading to that formal parameter.

> Correspondingly, label the statement following a procedure statement, if not labeled already, and provide that label as the corresponding extra actual parameter.

In modern terminology, this additional formal parameter is called a continuation, and the transform that introduces these continuation parameters is called a continuation-passing style (CPS) transform [14].

As a first illustration of the transformation, consider an example C program (on the left in Figure 1) containing a function definition and two calls to that function. In the middle column we use the Gnu dialect of C, which like ALGOL 60 has labels as values. The "`&&`" operator in Gnu C takes the address (of type `void *`) of a label. After this transformation, the function `p` never returns: `p` jumps to the return continuation `k` instead of ever reaching its closing `}`. Likewise, had there been any `return`-statements (absent in ALGOL 60), we would also have replaced them with an explicit jump `goto *k;`. In fact, the calls to `p` itself could be replaced with `goto`s, if we had some mechanism for passing arguments along with the jump. The resulting program is hardly one that anyone would care to write, being maximally unstructured in its use of `goto`s, but that is part of the point: it is closer to what might be generated in the early phase of compiling.

A further transformation can then be applied, turning all labels into pointers to functions (see the right column in Figure 1), essentially by rewriting `L:` ... as `void L(){...}`. If we transcribe mechanically, we would write for `L1`,

```
void L1(){p('b',&L2);}
```

But `L2` is only there to call `exit`, so it is simpler to optimize,

```
void L1(){p('b',&exit);}
```

The return type `void` is actually arbitrary here, as the function never returns. A `goto` is rewritten as a call to the function generated from the corresponding label; for instance, `goto *k;` is replaced by `(*k)();`. The C syntax may look similar at first sight, but note that in the first case, `k` is a pointer to `void` (that is, a machine address), while in the second case, `k` is (the address of) a function. For the purpose of this example, both are good enough implementations of continuations.

A certain amount of continuation-passing is possible in many languages that allow addresses, function pointers, or some other serviceable implementation of continuations to be passed as arguments. But for a more complete version, we now turn to the λ-calculus. The version of the CPS transform we give here is based on Plotkin's (call-by-value) transform [10], except that we use as the target language a small calculus idealizing the intermediate language from Appel's account of the Standard ML of New Jersey compiler [1, 16]. A CPS term M consists of a jump $kx_1 \dots x_n$ to k with arguments $x_1 \dots x_n$, together with some bindings of the form $\texttt{where}\ fx_1 \dots x_n = M'$. The grammar is given by the single rule,

$$M ::= xx^* \,(\texttt{where}\ xx^* = M)^*$$

To sketch the intended meaning, suffice it to say here that a jump should be reduced by fetching the term jumped to, and substituting the actual for the formal parameters (avoiding name capture):

$$\begin{array}{rl} & fx_1 \dots x_n\ \dots\ \texttt{where}\ fy_1 \dots y_n = M \dots \\ \to & M[y_1 := x_1, \dots, y_n := x_n]\ \dots\ \texttt{where}\ fy_1 \dots y_n = M \dots \end{array}$$

A λ-calculus term M relative to continuation k is transformed into a CPS term $[\![M]\!]k$ as follows:

$$\begin{array}{rcl} [\![x]\!]k & = & kx \\ [\![\lambda x.M]\!]k & = & kf \ \texttt{where} \ fxh = [\![M]\!]h \\ [\![MN]\!]k & = & [\![M]\!]m \ \texttt{where} \ mf = ([\![N]\!]n \ \texttt{where} \ nx = fxk) \end{array}$$

The crucial clause is the one for $\lambda x.M$, as it shows how functions are broken down into continuations. A function is translated into a (two-argument) continuation, to which an argument x and a return continuation h can be passed; the body M of the function is transformed in such a way that it will return its value by explicitly passing it to the return continuation. For instance, if the body is just x, it becomes hx.

To establish the link with the van Wijngaarden CPS earlier, consider as an example the term $(pa; pb)$, which, as usual, is shorthand for $(\lambda().pb)\,(pa)$. Eliding details and simplifications, we would CPS transform this into

$$[\![pa; pb]\!]e = pal_1 \ \texttt{where} \ l_1() = pbe$$

essentially as in Figure 1. CPS requires us to supply something as the outermost, or top-level, continuation. Here we use a free variable e, analogous to the use of C's "top-level continuation" `&exit` in the C code.

The jumps with arguments in the CPS notation can be translated into functions, that is, the CPS notation can be read as macros for λ-terms:

$$\begin{array}{rcl} (fx_1 \ldots x_n)^\circ & = & fx_1 \ldots x_n \\ (M \ \texttt{where} \ fx_1 \ldots x_n = N)^\circ & = & (\lambda f.M^\circ)(\lambda x_1 \ldots x_n.N^\circ) \end{array}$$

As with the transformation in Figure 1, we found it convenient to factor the CPS into the introduction of continuations as the first step, and implementation of the continuations as functions as a separate second step. We will say a little more about each in the following two sections.

3 From functions to jumps

At first sight, it may seem rather odd that conversion to CPS transforms functions into even more functions — but one should keep in mind how much simpler the resulting functions are. One does not even have to consider them as functions at all, as they are interchangeable with jumps.

The original function `p` has been transformed into a function that never returns. So when we write `p('a',&&L1);` to call `p`, we only want to use part of the procedure call mechanism, the ability to pass arguments, but not the ability to return from the call. (In fact, we inserted an `exit();` to get rid of the pile-up of unwanted returns.) If we had the ability to pass arguments along with a jump, say by way of a macro for pushing onto a call stack, then `p` itself could be transformed into a label. Calls to it would then read somewhat like this:

```
        push('a');
        push(&&L1);
        goto *p;
    L1: ;
```

We would still be transforming into a subset of C, but to one that is so restricted that, as far as control structure in concerned, it is closer to assembly language than C. (This subset is still far

removed from assembly in that it uses variables rather than explicit memory access, but this too could be brought closer to the machine by subsequent transformations, such as closure conversion [1], or register allocation.)

Despite these connections to compiling, the breaking down of functions into jumps does not force us to adopt a low-level, or overly machine-dependent viewpoint; it can in fact be expressed at quite an abstract level, such as types, or categories [16]. Specifically, let us consider a type system for the CPS language. We need only one constructor, $\neg$, so that types can be given by the grammar $\tau ::= \neg(\tau^*)$. At the type level, the breaking down of functions is given by the transform $[\![\tau_1 \to \tau_2]\!] = \neg([\![\tau_1]\!], \neg[\![\tau_2]\!])$. The closest approximation to $\neg\tau$ in C is `void(`τ`)`, conflating functions returning no value with those that do not return at all. In the C example, the type of `p`, `void(char)`, was transformed into `void(char,void(void))`.

This may make it clearer that we can translate the `where`-notation we used for CPS not only into the λ-calculus, but into a more primitive calculus *without* functions. Specifically, Milner's π-calculus [8] and its descendants depend on this aspect of continuations for their ability to encode "functions as processes" [9, 2, 16]. In such a process interpretation of continuations, we would read the jump "$f x_1 \ldots x_n$" as "send the values $x_1 \ldots x_n$ on channel f". The binding "`where` $f x_1 \ldots x_n = M$" would be read as "make a new process that can be sent values along a new channel f, and such that upon receiving $x_1 \ldots x_n$, this process will become unblocked and run M". In π-calculus notation, this interpretation amounts to the following macro translation:

$$\begin{aligned} (f x_1 \ldots x_n)^\bullet &= \overline{f}\langle x_1 \ldots x_n \rangle \\ (M \texttt{ where } f x_1 \ldots x_n = N)^\bullet &= (\nu f)\,(M^\bullet \mid !f(x_1 \ldots x_n).N^\bullet) \end{aligned}$$

The concurrency in the π-calculus plays no part here, but it is essential that the addresses of processes can be sent on channels.

In sum, this section concentrated on the breaking down of functions into jumps, or "λ as `goto` with arguments" [14, 1]. Though different authors have tended to emphasize different aspects of CPS as a useful tool in compiling, an added bonus of the CPS transform in the style of Appel [1] is that only atomic data (such as variables) need to be passed as arguments. While not a feature of CPS as such, this restricted calling mechanism is crucial for the translation to the π-calculus, which is a "name-passing" calculus. (Even in the C code in Figure 1 there was an element of this, in that only addresses were passed as arguments.)

4 From jumps to functions

We have sketched in the previous section how the CPS transform can be used to compile functions into more primitive jumps with arguments (corresponding to the first step in Figure 1.) Concentrating now on the second step, this section addresses how, conversely, the transform can be used to eliminate `goto` in favour of function call. Recall that in the second step of van Wijngaarden's transform, every `goto` was replaced by a function call. If the original program had already contained `goto`s, then these too would be replaced by function calls. For instance, suppose we had used a `goto` to skip the second call to `p` by jumping straight to the end:

```
main()
{
    p('a');
    goto end;
    p('b');
```

```
end: ;
}
```

After the transform, we have a program in which the second call to `p` appears only as part of a continuation that is never called.

```
void L1(){p('b', &end);}

void L(){(*end)();}

main()
{
    p('a', &L);
}
```

The non-functional behaviour of jumping past a statement is thus explicated in functional terms as simply ignoring the corresponding continuation. Moreover, as the resulting program contains no jumps, it would be much easier to assign a meaning in terms of mathematical (rather than C) functions to it than to the original. Rather than use a CPS transformation, one could also build the continuation-passing into the mathematical model itself; the principle remains much the same.

Apart from giving a functional meaning to `goto`, the CPS transform lets one see how restrictive `goto` is in that it does not take arguments. All the continuations generated by transforming `goto`s are `void`-accepting. That is to say, continuations that accept arguments suggest adding constructs to the source language to denote them. A straightforward way to achieve this generalization is to allow labels not only in front of statements, but anywhere in an expression. This generalization seems even more natural if one starts from a language based on the λ-calculus, where everything is an expression.

Whereas labels in imperative languages label statements, in λ-calculus, they need to label expressions. To make continuations available to the programmer, one could label a subexpression as in $(\ldots(\texttt{L:}\ M)\ldots)$. More accurately, it is not M itself that is labeled, but the position where M is implicitly slotted into some surrounding expression. Inside M, the label would be available as the target of a jump `goto L`. Expressions could then be interspersed with jumps, say:

$$100 + (\texttt{L:}\ (10 + ((\texttt{goto L})\ 1)))$$

with the intended meaning of "jump with the argument 1 to the place labeled `L` where the number to be added to 100 is expected", giving the result 101. 10 is not added to the result, as we jumped past it. The result is as if we had jumped between these statements:

```
    sum = 1;
    goto L;
    sum += 10;
L:  sum += 100;
```

This narrative may no doubt sound fanciful, but it is not that far removed from what actually happened in the history of programming languages. Generalizing `goto` to the λ-calculus led Landin to invent the **J**-operator [6, 7], and the construct for labeling the position of a subexpression is in fact Reynolds's `escape` operator [11]; one writes `escape L in` M in place of `L:` M. In a minor notational twist, `escape L in` M is nowadays written as `callcc`$(\lambda L.M)$, where `callcc` abbreviates "call with current continuation". For example, in Standard ML of New Jersey the above example looks like this:

```
100 + callcc(fn L => 10 + throw L 1)
```

Hence `callcc(fn L => ...)` is analogous to labeling `L: ...`, and `throw L...` to `goto L;`. The control operators `callcc` and `throw` are added to the λ-calculus by extending the CPS transform as follows:

$$\begin{aligned} [\![\texttt{callcc}\, M]\!]k &= [\![M]\!]m \ \texttt{where}\ mf = fkk \\ [\![\texttt{throw}\, M\, N]\!]k &= [\![M]\!]m \ \texttt{where}\ mh = ([\![N]\!]n \ \texttt{where}\ nx = hx) \end{aligned}$$

Only one programming language, Scheme [4], has such first-class continuations as part of its standard. But control constructs in countless other languages can be understood in terms of continuations. Often they can be seen as an idiom of `callcc`, possibly subject to certain implementation restrictions. The situation for continuations is in this regard quite analogous to that for functions: functions (or λ-calculus as a language for first-class functions) can be used to elucidate many constructs, whether or not the programming language under consideration itself has first-class functions. Many languages lack true first-class functions because, due to implementation restrictions, functions may not be returned as the result of functions. There is again an analogy with continuations here, because traditional control constructs like `goto` and `longjmp` in C [5] are less powerful than `callcc` for similar implementation restrictions on what can be returned from a function, as will be illustrated in the next section.

5 The impact of adding continuations to a language

It is sometimes said that, due to the existence of CPS transforms into the λ-calculus, control constructs such as `callcc` add nothing to a programming language, in that it can all be translated to the λ-calculus — a point of view not unlike the position that all Turing-complete languages are equivalent, as they can all compute exactly the same functions. To counter that impression, we will briefly sketch how radically the semantics of a programming language is altered by the addition of a continuation construct like `callcc`.

The fact that continuations have *indefinite extent* [4] makes `callcc` so powerful: a continuation can be invoked anywhere it is known, and it does not go away after being invoked, so that it can be invoked again. As an illustration of what is implied by indefinite extent here, and why it may seem mildly surprising in the context of jumps, we again start out by looking at Gnu C. This time, though, we do this for contrast as much as similarity: usages of jumps that are discouraged even in a language as permissive as Gnu C make perfect sense with `callcc`.

See Figure 2 for a Gnu C function that returns a local label as its result, so that after calling that function, one can jump back into it. The program duly prints

```
The function returned; now jump back into it.
Jumped back into the function.
```

but leaves the stack in disarray. It may hence seem as if the very idea of returning labels from a function so as to jump back into the function is inherently meaningless. In fact, any difficulties are not due to the jumping as such, but to the fact that in languages like C, anything local to a function cannot be returned as its result without some risk of smashing the stack. It is not essential whether the value returned is a local label or the address of a local variable, say:

```
int *a(int x) { return &x; }
```

```
void *label_as_result()
{
    return &&L;
 L: printf("Jumped back into the function. \n");
}

main()
{
    void *p;
    p = label_as_result();
    printf("The function returned; now jump back into it.\n");
    goto *p;
}
```

Figure 2: Jumping back into a function in Gnu C

But in a language with first-class functions and `callcc`, no such implementation restrictions apply. Just as with the label as a result in C, we can return a continuation introduced by `callcc` from a function so as to jump back into the function. When we did this with `goto`, the stack was smashed, but with `callcc` the function just returns *again*. Consider the following function:

$$\lambda().\texttt{callcc}(\lambda k.\lambda x.\texttt{throw}\, k\, (\lambda y.x))$$

The continuation k is returned as part of the result, roughly analogous to returning a label as the result in C. Because it is arguably the simplest case of such control behaviour, this function has been used for several counterexamples in the literature. In particular, consider the two expressions

$$(\lambda x.pxx)\, M \qquad \text{and} \qquad (\lambda x.\lambda y.pxy)\, M\, M$$

One could argue that there is no way to distinguish these two expressions in a language without assignment, not even by jumping. For, should a jump occur during the evaluation of M, both expressions will boil down to a jump. What is remarkable is that this argument is essentially correct — but not for continuations. More precisely, one can formalize the argument to show that in a language with a different form of jumping, namely exceptions, the two expressions are indistinguishable [13]. In a language with `callcc`, by contrast, the expressions can be distinguished by using the function above, which returns twice.

The point we would like to make here is not so much *that* this obvious-seeming equivalence fails, but *why* it fails: in the presence of continuations, function call is not like a substitution of the actual for the formal parameters, or some variant thereof. Rather, it is a jump with arguments, that may return by jumping back to the caller — or not, or more than once.

6 Conclusions

The breaking down of functions by means of continuations has been emphasized in CPS-based compiling for decades. But in programming language theory, it seems to have been comparatively neglected, so that one could be forgiven for thinking of continuations as little more than a technical device for the domestication of `goto`. We found the "λ as `goto` plus arguments" view [14] so

fundamental for understanding continuations that we chose here to sacrifice other aspects, however important. For instance, in denotational semantics, where the term "continuation" was actually coined [15], one would say that a continuation is a function mapping values to final answers, while in type theory one would stress that CPS transforms correspond to double-negation translations, and that control operators such as `callcc` have "classical" types [3].

Writing about the "discoveries of continuations", Reynolds [12] describes how continuations were not only discovered but even inadvertently *re*-discovered several times. The encodings of λ-calculus into π-calculus can, with hindsight, be seen as an instance of continuation-passing style. At the moment, there seems to be another such rediscovery under way, this time under the banner of λ-calculi for classical logic. Such rediscoveries are perhaps to be expected, given the protean nature of continuations, and witness to their importance as a fundamental concept.

References

[1] Andrew Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Gérard Boudol. Pi-calculus in direct style. In *ACM Symposium on Principles of Programming Languages*, 1997.

[3] Timothy G. Griffin. A formulae-as-types notion of control. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, CA USA, 1990.

[4] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(3):7–105, 1998.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 2nd edition, 1988.

[6] Peter J. Landin. A generalization of jumps and labels. Report, UNIVAC Systems Programming Research, August 1965.

[7] Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2), 1998. Reprint of [6].

[8] Robin Milner. The polyadic π-calculus: A tutorial. LFCS Report ECS-LFCS-91-180, LFCS, University of Edinburgh, October 1991.

[9] Robin Milner. Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141, 1992.

[10] Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[11] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, August 1972.

[12] John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3/4):233–247, November 1993.

[13] Jon G. Riecke and Hayo Thielecke. Typed exceptions and continuations cannot macro-express each other. In *Proc. ICALP '99*, 1999.

[14] Guy Steele. Rabbit: A compiler for Scheme. Technical Report AI TR 474, MIT, May 1978.

[15] Christopher Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, January 1974.

[16] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, 1997. Also available as technical report ECS-LFCS-97-376.

[17] Adriaan van Wijngaarden. Recursive definition of syntax and semantics. In T. B. Steel, Jr, editor, *Formal Language Description Languages for Computer Programming, Proceedings of an IFIP Working Conference*, pages 13–24, 1964.