WIKIPEDIA

Continuation-passing style

In functional programming, **continuation-passing style** (**CPS**) is a style of programming in which control is passed explicitly in the form of a continuation. This is contrasted with direct style, which is the usual style of programming. Gerald Jay Sussman and Guy L. Steele, Jr. coined the phrase in AI Memo 349 (1975), which sets out the first version of the Scheme programming language.[1][2] John C. Reynolds gives a detailed account of the numerous discoveries of continuations.[3]

A function written in continuation-passing style takes an extra argument: an explicit "continuation"; *i.e.*, a function of one argument. When the CPS function has computed its result value, it "returns" it by calling the continuation function with this value as the argument. That means that when invoking a CPS function, the calling function is required to supply a procedure to be invoked with the subroutine's "return" value. Expressing code in this form makes a number of things explicit which are implicit in direct style. These include: procedure returns, which become apparent as calls to a continuation; intermediate values, which are all given names; order of argument evaluation, which is made explicit; and tail calls, which simply call a procedure with the same continuation, unmodified, that was passed to the caller.

Programs can be automatically transformed from direct style to CPS. Functional and logic compilers often use CPS as an intermediate representation where a compiler for an imperative or procedural programming language would use static single assignment form (SSA).[4] SSA is formally equivalent to a subset of CPS (excluding non-local control flow, which does not occur when CPS is used as intermediate representation).[5] Functional compilers can also use A-normal form (ANF) (but only for languages requiring eager evaluation), rather than with 'thunks' (described in the examples below) in CPS. CPS is used more frequently by compilers than by programmers as a local or global style.

Contents

Examples
- CPS in Haskell
- Continuations as objects

Tail calls

Use and implementation

Use in other fields

See also

Notes

References

Examples

In CPS, each procedure takes an extra argument representing what should be done with the result the function is calculating. This, along with a restrictive style prohibiting a variety of constructs usually available, is used to expose the semantics of programs, making them easier to analyze. This style also makes it easy to express unusual control structures, like catch/throw or other non-local transfers of control.

The key to CPS is to remember that (a) *every* function takes an extra argument known as its continuation, and (b) every argument in a function call must be either a variable or a lambda expression (not a more complex expression). This has the effect of turning expressions "inside-out" because the innermost parts of the expression must be evaluated first, thus CPS makes explicit the order of evaluation as well as the control flow. Some examples of code in direct style and the corresponding CPS appear below. These examples are written in the Scheme programming language; by convention the continuation function is represented as a parameter named "k":

Direct style

```
(define (pyth x y)
 (sqrt (+ (* x x) (* y y))))
```

```
(define (factorial n)
 (if (= n 0)
     1     ; NOT tail-
recursive
     (* n (factorial (- n
1)))))
```

```
(define (factorial n)
 (f-aux n 1))
(define (f-aux n a)
 (if (= n 0)
     a        ; tail-recursive
     (f-aux (- n 1) (* n a))))
```

Continuation passing style

```
(define (pyth& x y k)
 (*& x x (lambda (x2)
          (*& y y (lambda (y2)
                   (+& x2 y2 (lambda (x2py2)
                              (sqrt& x2py2 k))))))))
```

```
(define (factorial& n k)
 (=& n 0 (lambda (b)
          (if b                    ; growing continuation
              (k 1)                ; in the recursive
call
              (-& n 1 (lambda (nm1)
                       (factorial& nm1 (lambda (f)
                                        (*& n f
k)))))))))
```

```
(define (factorial& n k) (f-aux& n 1 k))
(define (f-aux& n a k)
 (=& n 0 (lambda (b)
          (if b                    ; unmodified
continuation
              (k a)                ; in the recursive
call
              (-& n 1 (lambda (nm1)
                       (*& n a (lambda (nta)
                                (f-aux& nm1 nta
k)))))))))
```

Note that in the CPS versions, the primitives used, like +& and *& are themselves CPS, not direct style, so to make the above examples work in a Scheme system we would need to write these CPS versions of primitives, with for instance *& defined by:

```
(define (*& x y k)
 (k (* x y)))
```

To do this in general, we might write a conversion routine:

```
(define (cps-prim f)
 (lambda args
  (let ((r (reverse args)))
   ((car r) (apply f
             (reverse (cdr r)))))))
(define *& (cps-prim *))
(define +& (cps-prim +))
```

In order to call a procedure written in CPS from a procedure written in direct style, it is necessary to provide a continuation that will receive the result computed by the CPS procedure. In the example above (assuming that CPS-style primitives have been provided), we might call `(factorial& 10 (lambda (x) (display x) (newline)))`.

There is some variety between compilers in the way primitive functions are provided in CPS. Above we have used the simplest convention, however sometimes boolean primitives are provided that take two thunks to be called in the two possible cases, so the `(=& n 0 (lambda (b) (if b ...)))` call inside `f-aux&` definition above would be written instead as `(=& n 0 (lambda () (k a)) (lambda () (-& n 1 ...)))`. Similarly, sometimes the `if` primitive itself is not included in CPS, and instead a function `if&` is provided which takes three arguments: a boolean condition and the two thunks corresponding to the two arms of the conditional.

The translations shown above show that CPS is a global transformation. The direct-style *factorial* takes, as might be expected, a single argument; the CPS *factorial&* takes two: the argument and a continuation. Any function calling a CPS-ed function must either provide a new continuation or pass its own; any calls from a CPS-ed function to a non-CPS function will use implicit continuations. Thus, to ensure the total absence of a function stack, the entire program must be in CPS.

CPS in Haskell

In this section we will write a function `pyth` that calculates the hypotenuse using the Pythagorean theorem. A traditional implementation of the `pyth` function looks like this:

```
pow2 :: Float -> Float
pow2 a = a ** 2

add :: Float -> Float -> Float
add a b = a + b

pyth :: Float -> Float -> Float
pyth a b = sqrt (add (pow2 a) (pow2 b))
```

To transform the traditional function to CPS, we need to change its signature. The function will get another argument of function type, and its return type depends on that function:

```
pow2' :: Float -> (Float -> a) -> a
pow2' a cont = cont (a ** 2)

add' :: Float -> Float -> (Float -> a) -> a
add' a b cont = cont (a + b)

-- Types a -> (b -> c) and a -> b -> c are equivalent, so CPS function
-- may be viewed as a higher order function
sqrt' :: Float -> ((Float -> a) -> a)
sqrt' a = \cont -> cont (sqrt a)

pyth' :: Float -> Float -> (Float -> a) -> a
pyth' a b cont = pow2' a (\a2 -> pow2' b (\b2 -> add' a2 b2 (\anb -> sqrt' anb cont)))
```

First we calculate the square of *a* in `pyth'` function and pass a lambda function as a continuation which will accept a square of *a* as a first argument. And so on until we reach the result of our calculations. To get the result of this function we can pass `id` function as a final argument which returns the value that was passed to it unchanged: `pyth' 3 4 id == 5.0`.

The mtl library, which is shipped with GHC, has the module `Control.Monad.Cont`. This module provides the Cont type, which implements Monad and some other useful functions. The following snippet shows the `pyth'` function using Cont:

```
pow2_m :: Float -> Cont a Float
pow2_m a = return (a ** 2)

pyth_m :: Float -> Float -> Cont a Float
pyth_m a b = do
  a2 <- pow2_m a
  b2 <- pow2_m b
  anb <- cont (add' a2 b2)
  r <- cont (sqrt' anb)
  return r
```

Not only has the syntax become cleaner, but this type allows us to use a function `callCC` with type `MonadCont m => ((a -> m b) -> m a) -> m a`. This function has one argument of a function type; that function argument accepts the function too, which discards all computations going after its call. For example, let's break the execution of the `pyth` function if at least one of its arguments is negative returning zero:

```
pyth_m :: Float -> Float -> Cont a Float
pyth_m a b = callCC $ \exitF -> do -- $ sign helps to avoid parentheses: a $ b + c == a (b +
c)
  when (b < 0 || a < 0) (exitF 0.0) -- when :: Applicative f => Bool -> f () -> f ()
  a2 <- pow2_m a
  b2 <- pow2_m b
  anb <- cont (add' a2 b2)
  r <- cont (sqrt' anb)
  return r
```

Continuations as objects

Programming with continuations can also be useful when a caller does not want to wait until the callee completes. For example, in user-interface (UI) programming, a routine can set up dialog box fields and pass these, along with a continuation function, to the UI framework. This call returns right away, allowing the application code to continue while the user interacts with the dialog box. Once the user presses the "OK" button, the framework calls the continuation function with the updated fields. Although this style of coding uses continuations, it is not full CPS.

```
function confirmName() {
    fields.name = name;
    framework.Show_dialog_box(fields, confirmNameContinuation);
}

function confirmNameContinuation(fields) {
    name = fields.name;
}
```

A similar idea can be used when the function must run in a different thread or on a different processor. The framework can execute the called function in a worker thread, then call the continuation function in the original thread with the worker's results. This is in Java using the Swing UI framework:

```
void buttonHandler() {
    // This is executing in the Swing UI thread.
    // We can access UI widgets here to get query parameters.
    final int parameter = getField();
    new Thread(new Runnable() {
        public void run() {
            // This code runs in a separate thread.
            // We can do things like access a database or a
            // blocking resource like the network to get data.
            final int result = lookup(parameter);

            javax.swing.SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    // This code runs in the UI thread and can use
                    // the fetched data to fill in UI widgets.
                    setField(result);
                }
            });
        }
    }).start();
}
```

Using Java 8 lambda notation above code shortens to (`final` keyword might be skipped):

```
void buttonHandler() {
    int parameter = getField();

    new Thread(() -> {
        int result = lookup(parameter);
        javax.swing.SwingUtilities.invokeLater(() -> setField(result));
    }).start();
}
```

Tail calls

Every call in CPS is a tail call, and the continuation is explicitly passed. Using CPS without tail call optimization (TCO) will cause not only the constructed continuation to potentially grow during recursion, but also the call stack. This is usually undesirable, but has been used in interesting ways—see the Chicken Scheme compiler. As CPS and TCO eliminate the concept of an implicit function return, their combined use can eliminate the need for a run-time stack. Several compilers and interpreters for functional programming languages use this ability in novel ways.[6]

Use and implementation

Continuation passing style can be used to implement continuations and control flow operators in a functional language that does not feature first-class continuations but does have first-class functions and tail-call optimization. Without tail-call optimization, techniques such as trampolining, i.e. using a loop that iteratively invokes thunk-returning functions, can be used; without first-class functions, it is even possible to convert tail calls into just gotos in such a loop.

Writing code in CPS, while not impossible, is often error-prone. There are various translations, usually defined as one- or two-pass conversions of pure lambda calculus, which convert direct style expressions into CPS expressions. Writing in trampolined style, however, is extremely difficult; when used, it is usually the target of some sort of transformation, such as compilation.

Functions using more than one continuation can be defined to capture various control flow paradigms, for example (in Scheme):

```
(define (/& x y ok err)
 (=& y 0.0 (lambda (b)
            (if b
            (err (list "div by zero!" x y))
            (ok (/ x y))))))
```

It is of note that CPS transform is conceptually a Yoneda embedding.[7] It is also similar to the embedding of lambda calculus in π-calculus.[8][9]

Use in other fields

Outside of computer science, CPS is of more general interest as an alternative to the conventional method of composing simple expressions into complex expressions. For example, within linguistic semantics, Chris Barker and his collaborators have suggested that specifying the denotations of sentences using CPS might explain certain phenomena in natural language.[10]

In mathematics, the Curry–Howard isomorphism between computer programs and mathematical proofs relates continuation-passing style translation to a variation of double-negation embeddings of classical logic into intuitionistic (constructive) logic. Unlike the regular double-negation translation, which maps atomic propositions p to $((p \to \bot) \to \bot)$, the continuation passing style replaces $\bot$ by the type of the final expression. Accordingly, the result is obtained by passing the identity function as a continuation to the CPS-style expression, as in the above example.

Classical logic itself relates to manipulating the continuation of programs directly, as in Scheme's call-with-current-continuation control operator, an observation due to Tim Griffin (using the closely related C control operator).[11]

See also

- Tail recursion through trampolining

Notes

1. Sussman, Gerald Jay; Steele, Guy L., Jr. (December 1975). "Scheme: An interpreter for extended lambda calculus" (https://en.wikisource.org/wiki/Scheme:_An_interpreter_for_extended_lambda_calculus). *AI Memo*. **349**: 19. "That is, in this **continuation-passing programming style**, *a function always "returns" its result by "sending" it to another function*. This is the key idea."
2. Sussman, Gerald Jay; Steele, Guy L., Jr. (December 1998). "Scheme: A Interpreter for Extended Lambda Calculus" (http://www.brics.dk/~hosc/local/HOSC-11-4-pp405-439.pdf) (reprint). *Higher-Order and Symbolic Computation*. **11** (4): 405–439. doi:10.1023/A:1010035624696 (https://doi.org/10.1023%2FA%3A1010035624696). "We believe that this was the first occurrence of the term "**continuation-passing style**" in the literature. It has turned out to be an important concept in source code analysis and transformation for compilers and other metaprogramming tools. It has also inspired a set of other "styles" of program expression."
3. Reynolds, John C. (1993). "The Discoveries of Continuations". *Lisp and Symbolic Computation*. **6** (3–4): 233–248. CiteSeerX 10.1.1.135.4705 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.135.4705). doi:10.1007/bf01019459 (https://doi.org/10.1007%2Fbf01019459).
4. * Appel, Andrew W. (April 1998). "SSA is Functional Programming". *ACM SIGPLAN Notices*. **33** (4): 17–20. CiteSeerX 10.1.1.34.3282 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.3282). doi:10.1145/278283.278285 (https://doi.org/10.1145%2F278283.278285).

5. * Kelsey, Richard A. (March 1995). "A Correspondence between Continuation Passing Style and Static Single Assignment Form". *ACM SIGPLAN Notices*. **30** (3): 13–22. CiteSeerX 10.1.1.489.930 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.489.930). doi:10.1145/202530.202532 (https://doi.org/10.1145%2F202530.202532).
6. Appel, Andrew W. (1992). Compiling with Continuations. Cambridge University Press. ISBN 0-521-41695-7.
7. Mike Stay, "The Continuation Passing Transform and the Yoneda Embedding" (http://golem.ph.utexas.edu/category/2008/01/the_continuation_passing_trans.html)
8. Mike Stay, "The Pi Calculus II" (http://golem.ph.utexas.edu/category/2009/09/the_pi_calculus_ii.html)
9. Boudol, Gérard (1997). "The π-Calculus in Direct Style". CiteSeerX 10.1.1.52.6034 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.52.6034).
10. Barker, Chris (2002-09-01). "Continuations and the Nature of Quantification" (http://www.semanticsarchive.net/Archive/902ad5f7/barker.continuations.pdf) (PDF). *Natural Language Semantics*. **10** (3): 211–242. doi:10.1023/A:1022183511876 (https://doi.org/10.1023%2FA%3A1022183511876). ISSN 1572-865X (https://www.worldcat.org/issn/1572-865X).
11. Griffin, Timothy (January 1990). *A formulae-as-type notion of control*. *Proceedings of the Conference on the Principles of Programming Languages*. **17**. pp. 47–58. doi:10.1145/96709.96714 (https://doi.org/10.1145%2F96709.96714). ISBN 978-0897913430.

References

- Continuation Passing C (CPC) - programming language for writing concurrent systems (http://www.pps.univ-paris-diderot.fr/~kerneis/software/), designed and developed by Juliusz Chroboczek and Gabriel Kerneis. github repository (https://github.com/kerneis/cpc)
- The construction of a CPS-based compiler for ML is described in: Appel, Andrew W. (1992). *Compiling with Continuations* (https://books.google.com/books?id=0Uoecu9ju4AC&dq). Cambridge University Press. ISBN 978-0-521-41695-5.
- Danvy, Olivier; Filinski, Andrzej (1992). "Representing Control, A Study of the CPS Transformation". *Mathematical Structures in Computer Science*. **2** (4): 361–391. CiteSeerX 10.1.1.46.84 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.84). doi:10.1017/S0960129500001535 (https://doi.org/10.1017%2FS0960129500001535).
- Chicken Scheme compiler, a Scheme to C compiler that uses continuation-passing style for translating Scheme procedures into C functions while using the C-stack as the nursery for the generational garbage collector
- Kelsey, Richard A. (March 1995). "A Correspondence between Continuation Passing Style and Static Single Assignment Form". *ACM SIGPLAN Notices*. **30** (3): 13–22. CiteSeerX 10.1.1.3.6773 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.3.6773). doi:10.1145/202530.202532 (https://doi.org/10.1145%2F202530.202532).
- Appel, Andrew W. (April 1998). "SSA is Functional Programming" (http://www.cs.princeton.edu/~appel/papers/ssafun.ps). *ACM SIGPLAN Notices*. **33** (4): 17–20. CiteSeerX 10.1.1.34.3282 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.34.3282). doi:10.1145/278283.278285 (https://doi.org/10.1145%2F278283.278285).
- Danvy, Olivier; Millikin, Kevin; Nielsen, Lasse R. (2007). "On One-Pass CPS Transformations" (http://www.brics.dk/RS/07/6/). BRICS, Department of Computer Science, University of Aarhus. p. 24. ISSN 0909-0878 (https://www.worldcat.org/issn/0909-0878). RS-07-6. Retrieved 26 October 2007.
- Dybvig, R. Kent (2003). *The Scheme Programming Language* (http://www.scheme.com/tspl3/). Prentice Hall. p. 64. Direct link: "Section 3.4. Continuation Passing Style" (http://scheme.

com/tspl3/further.html#./further:h4).