Continuation

In computer science, a **continuation** is an abstract representation of the control state of a computer program. A continuation implements (reifies) the program control state, i.e. the continuation is a data structure that represents the computational process at a given point in the process's execution; the created data structure can be accessed by the programming language, instead of being hidden in the runtime environment. Continuations are useful for encoding other control mechanisms in programming languages such as exceptions, generators, coroutines, and so on.

The "**current continuation**" or "continuation of the computation step" is the continuation that, from the perspective of running code, would be derived from the current point in a program's execution. The term *continuations* can also be used to refer to **first-class continuations**, which are constructs that give a programming language the ability to save the execution state at any point and return to that point at a later point in the program, possibly multiple times.

Contents

History

The earliest description of continuations was made by Adriaan van Wijngaarden in September 1964. Wijngaarden spoke at the IFIP Working Conference on Formal Language Description Languages held in Baden bei Wien, Austria. As part of a formulation for an Algol 60 preprocessor, he called for a transformation of proper procedures into continuation-passing style,[1] though he did not use this name, and his intention was to simplify a program and thus make its result more clear.

Christopher Strachey, Christopher P. Wadsworth and John C. Reynolds brought the term *continuation* into prominence in their work in the field of denotational semantics that makes extensive use of continuations to allow sequential programs to be analysed in terms of functional programming semantics.[1]

Steve Russell[2] invented the continuation in his second Lisp implementation for the IBM 704, though he did not name it.[3]

Reynolds (1993) gives a complete history of the discovery of continuations.

First-class continuations

First-class continuations are a language's ability to completely control the execution order of instructions. They can be used to jump to a function that produced the call to the current function, or to a function that has previously exited. One can think of a first-class continuation as saving the *execution* state of the program. It is important to note that true first-class continuations do not save program data – unlike a process image – only the execution context. This is illustrated by the "continuation sandwich" description:

> *Say you're in the kitchen in front of the refrigerator, thinking about a sandwich. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there's a sandwich on the counter, and all the materials used to make it are gone. So you eat it. :-)*[4]

In this description, the sandwich is part of the program *data* (e.g., an object on the heap), and rather than calling a "make sandwich" routine and then returning, the person called a "make sandwich with current continuation" routine, which creates the sandwich and then continues where execution left off.

Scheme was the first full production system (1969-1970), providing first "catch"[1] and then call/cc. Bruce Duba introduced call/cc into SML.

Continuations are also used in models of computation including denotational semantics, the actor model, process calculi, and lambda calculus. These models rely on programmers or semantics engineers to write mathematical functions in the so-called continuation-passing style. This means that each function consumes a function that represents the rest of the computation relative to this function call. To return a value, the function calls this "continuation function" with a return value; to abort the computation it returns a value.

Functional programmers who write their programs in continuation-passing style gain the expressive power to manipulate the flow of control in arbitrary ways. The cost is that they must maintain the invariants of control and continuations by hand, which can be a highly complex undertaking (but see 'continuation-passing style' below).

Uses

Continuations simplify and clarify the implementation of several common design patterns, including coroutines/green threads and exception handling, by providing the basic, low-level primitive which unifies these seemingly unconnected patterns. Continuations can provide elegant solutions to some difficult high-level problems, like programming a web server that supports multiple pages, accessed by the use of the

forward and back buttons and by following links. The Smalltalk Seaside web framework uses continuations to great effect, allowing one to program the web server in procedural style, by switching continuations when switching pages.

More complex constructs for which *"continuations provide an elegant description"*[1] also exist. For example, in C, longjmp can be used to jump from the middle of one function to another, provided the second function lies deeper in the stack (if it is waiting for the first function to return, possibly among others). Other more complex examples include coroutines in Simula 67, Lua, and Perl; tasklets in Stackless Python; generators in Icon and Python; continuations in Scala (starting in 2.8); fibers in Ruby (starting in 1.9.1); the backtracking mechanism in Prolog; monads in functional programming; and threads.

Examples

The Scheme programming language includes the control operator call-with-current-continuation (abbreviated as: call/cc) with which a Scheme program can manipulate the flow of control:

```
(define the-continuation #f)

(define (test)
  (let ((i 0))
    ; call/cc calls its first function argument, passing
    ; a continuation variable representing this point in
    ; the program as the argument to that function.
    ;
    ; In this case, the function argument assigns that
    ; continuation to the variable the-continuation.
    ;
    (call/cc (lambda (k) (set! the-continuation k)))
    ;
    ; The next time the-continuation is called, we start here.
    (set! i (+ i 1))
    i))
```

Using the above, the following code block defines a function `test` that sets `the-continuation` to the future execution state of itself:

```
> (test)
1
> (the-continuation)
2
> (the-continuation)
3
> ; stores the current continuation (which will print 4 next) away
> (define another-continuation the-continuation)
> (test) ; resets the-continuation
1
> (the-continuation)
2
> (another-continuation) ; uses the previously stored continuation
4
```

For a gentler introduction to this mechanism, see call-with-current-continuation.

Coroutines

This example shows a possible usage of continuations to implement coroutines as separate threads.[5]

```
;;; A naive queue for thread scheduling.
;;; It holds a list of continuations "waiting to run".
```

```
(define *queue* '())

(define (empty-queue?)
  (null? *queue*))

(define (enqueue x)
  (set! *queue* (append *queue* (list x))))

(define (dequeue)
  (let ((x (car *queue*)))
    (set! *queue* (cdr *queue*))
    x))

;;; This starts a new thread running (proc).

(define (fork proc)
  (call/cc
   (lambda (k)
     (enqueue k)
     (proc))))

;;; This yields the processor to another thread, if there is one.

(define (yield)
  (call/cc
   (lambda (k)
     (enqueue k)
     ((dequeue)))))

;;; This terminates the current thread, or the entire program
;;; if there are no other threads left.

(define (thread-exit)
  (if (empty-queue?)
      (exit)
      ((dequeue))))
```

The functions defined above allow for defining and executing threads through cooperative multitasking, i.e. threads that yield control to the next one in a queue:

```
;;; The body of some typical Scheme thread that does stuff:

(define (do-stuff-n-print str)
  (lambda ()
    (let loop ((n 0))
      (format #t "~A ~A\n" str n)
      (yield)
      (loop (+ n 1)))))

;;; Create two threads, and start them running.
(fork (do-stuff-n-print "This is AAA"))
(fork (do-stuff-n-print "Hello from BBB"))
(thread-exit)
```

The previous code will produce this output:

```
This is AAA 0
Hello from BBB 0
This is AAA 1
Hello from BBB 1
This is AAA 2
Hello from BBB 2
...
```

Implementation

A program must allocate space in memory for the variables its functions use. Most programming languages use a call stack for storing the variables needed because it allows for fast and simple allocating and automatic deallocation of memory. Other programming languages use a heap for this, which allows for flexibility at a higher cost for allocating and deallocating memory. Both of these implementations have benefits and drawbacks in the context of continuations.[6]

Programming language support

Many programming languages exhibit first-class continuations under various names; specifically:

- Common Lisp: cl-cont (http://common-lisp.net/project/cl-cont/). One can also use custom macros
- C# / VB.NET: `async` and `await`: "sign up the rest of method as the continuation, and then return to your caller immediately; the task will invoke the continuation when it completes." Asynchronous Programming for C# (http://msdn.microsoft.com/en-us/vstudio/gg316360)
- Factor: `callcc0` and `callcc1`
- Haskell: The Continuation monad in `Control.Monad.Cont (http://hackage.haskell.org/packages/archive/mtl/2.0.1.0/doc/html/Control-Monad-Cont.html)`
- Haxe: haxe-continuation (https://github.com/Atry/haxe-continuation)
- Icon, Unicon : `create, suspend, @` operator: coexpressions
- Java: Lightwolf (http://lightwolf.sourceforge.net/index.html) javaflow (http://commons.apache.org/sandbox/commons-javaflow/) (requires bytecode manipulation at runtime or compile time)
- Kotlin : `Continuation`
- JavaScript Rhino : `Continuation`
- Parrot: `Continuation` PMC; uses continuation-passing style for all control flow
- Perl: Coro (https://metacpan.org/module/Coro) and Continuity (https://metacpan.org/module/Continuity)
- Pico: `call(exp())` and `continue(aContinuation, anyValue)`
- Python: PyPy's `_continuation.continulet (http://pypy.readthedocs.org/en/latest/stackless.html)`
- Racket: `call-with-current-continuation` (commonly shortened to `call/cc`)
- Ruby: `callcc`
- Scala: `scala.util.continuations` provides `shift`/`reset`
- Scheme: `call-with-current-continuation` (commonly shortened to `call/cc`)
- Smalltalk: `Continuation currentDo:`; in most modern Smalltalk environments continuations can be implemented without additional VM support.
- Standard ML of New Jersey: `SMLofNJ.Cont.callcc`
- Unlambda: `c`, the flow control operation for call with current continuation

In any language which supports closures and proper tail calls, it is possible to write programs in continuation-passing style and manually implement call/cc. (In continuation-passing style, call/cc becomes a simple function that can be written with lambda.) This is a particularly common strategy in Haskell, where it is easy to construct a "continuation-passing monad" (for example, the `Cont` monad and `ContT` monad transformer in the `mtl` library). The support for proper tail calls is needed because in continuation-passing style no function ever returns; *all* calls are tail calls.

In Web development

One area that has seen practical use of continuations is in Web programming.[7][8] The use of continuations shields the programmer from the stateless nature of the HTTP protocol. In the traditional model of web programming, the lack of state is reflected in the program's structure, leading to code constructed around a model that lends itself very poorly to expressing computational problems. Thus continuations enable code that has the useful properties associated with inversion of control, while avoiding its problems. Inverting back the inversion of control or, Continuations versus page-centric programming (https://pages.lip6.fr/Christian.Queinnec/PDF/www.pdf) is a paper that provides a good introduction to continuations applied to web programming.

Some of the more popular continuation-aware Web servers are the Racket Web Server (http://docs.racket-lang.org/web-server/), the UnCommon Web Framework (http://common-lisp.net/project/ucw) and Weblocks Web framework (http://common-lisp.net/project/cl-weblocks/) for Common Lisp, the Seaside framework for Smalltalk, Ocsigen/Eliom for OCaml, Continuity (https://metacpan.org/module/Continuity) for Perl, Wee (https://github.com/mneumann/wee) for Ruby, Tales Framework (https://archive.is/20130416024741/http://www.talesframework.org/) for Fantom and the Nagare framework (http://www.nagare.org/) for Python, Wt (http://webtoolkit.eu/wt) for C++, MFlow (https://github.com/agocorona/MFlow) for Haskell. The Apache Cocoon Web application framework also provides continuations (see the Cocoon manual (http://cocoon.apache.org/2.1/userdocs/flow/continuations.html)).

Kinds

Support for continuations varies widely. A programming language supports *re-invocable* continuations if a continuation may be invoked repeatedly (even after it has already returned). Re-invocable continuations were introduced by Peter J. Landin using his **J** (for Jump) operator that could transfer the flow of control back into the middle of a procedure invocation. Re-invocable continuations have also been called "re-entrant" in the Racket language. However this use of the term "re-entrant" can be easily confused with its use in discussions of multithreading.

A more limited kind is the *escape continuation* that may be used to escape the current context to a surrounding one. Many languages which do not explicitly support continuations support exception handling, which is equivalent to escape continuations and can be used for the same purposes. C's `setjmp/longjmp` are also equivalent: they can only be used to unwind the stack. Escape continuations can also be used to implement tail call elimination.

One generalization of continuations are delimited continuations. Continuation operators like `call/cc` capture the *entire* remaining computation at a given point in the program and provide no way of delimiting this capture. Delimited continuation operators address this by providing two separate control mechanisms: a *prompt* that delimits a continuation operation and a *reification* operator such as `shift` or `control`. Continuations captured using delimited operators thus only represent a slice of the program context.

Disadvantages

Continuations are the functional expression of the GOTO statement, and the same caveats apply.[9] While they are a sensible option in some special cases such as web programming, use of continuations can result in code that is difficult to follow. In fact, the esoteric programming language Unlambda includes call-with-current-continuation as one of its features solely because expressions involving it "tend to be hopelessly difficult to track down."[10] The external links below illustrate the concept in more detail.

Linguistics

In "Continuations and the nature of quantification", Chris Barker introduced the "continuation hypothesis", that

> some linguistic expressions (in particular, QNPs [quantificational noun phrases]) have denotations that manipulate their own continuations.[11]

Barker argued that this hypothesis could be used to explain phenomena such as *duality of NP meaning* (e.g., the fact that the QNP "everyone" behaves very differently from the non-quantificational noun phrase "Bob" in contributing towards the meaning of a sentence like "Alice sees [Bob/everyone]"), *scope displacement* (e.g., that "a raindrop fell on every car" is interpreted typically as $\forall c \exists r, \mathbf{fell}(r, c)$ rather than as $\exists r \forall c, \mathbf{fell}(r, c)$), and *scope ambiguity* (that a sentence like "someone saw everyone" may be ambiguous between $\exists x \forall y, \mathbf{saw}(x, y)$ and $\forall y \exists x, \mathbf{saw}(x, y)$). He also observed that this idea is in a way just a natural extension of Richard Montague's approach in "The Proper Treatment of Quantification in Ordinary English" (PTQ), writing that "with the benefit of hindsight, a limited form of continuation-passing is clearly discernible at the core of Montague's (1973) PTQ treatment of NPs as generalized quantifiers".

The extent to which continuations can be used to explain other general phenomena in natural language is a topic of current research.[12]

See also

- Call-with-current-continuation
- Closure
- COMEFROM
- Continuation-passing style
- Control flow
- Coroutine
- Delimited continuation
- Denotational semantics
- GOTO
- Spaghetti stack
- Quajects, a type of object which allows selectable continuations (called 'callouts') to be set for methods on a per-object basis, through Dependency Injection.

References

1. Reynolds 1993
2. S.R. Russell noticed that *eval* could serve as an interpreter for LISP, promptly hand coded it, and we now had a programming language with an interpreter. (http://www-formal.stanford.edu/jmc/history/lisp/node3.html) —John McCarthy, *History of LISP*
3. "Steve "Slug" Russell" (http://www.computernostalgia.net/articles/steveRussell.htm). *Computer History*.
4. Palmer, Luke (June 29, 2004). "undo()? ("continuation sandwich" example)" (https://groups.google.com/group/perl.perl6.language/msg/b0cfa757f0ce1cfd). *perl.perl6.language (newsgroup)*. Retrieved 2009-10-04.

5. Haynes, C. T., Friedman, D. P., and Wand, M. 1984. Continuations and coroutines. In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (Austin, Texas, United States, August 06–08, 1984). LFP '84. ACM, New York, NY, 293-298.
6. "Call with current continuation for C programmers" (http://community.schemewiki.org/?call-with-current-continuation-for-C-programmers). *Community-Scheme-Wiki*. 12 October 2008.
7. "Reading list on XML and Web Programming" (https://web.archive.org/web/20100614042239/http://readscheme.org/xml-web/). Archived from the original (http://readscheme.org/xml-web/) on 2010-06-14. Retrieved 2006-08-03.
8. "Web Programming with Continuations" (https://wayback.archive-it.org/all/20120905083130/http://double.co.nz/pdf/continuations.pdf) (PDF). Archived from the original (http://double.co.nz/pdf/continuations.pdf) (PDF) on 2012-09-05. Retrieved 2012-09-05.
9. Quigley, John (September 2007). "Computational Continuations" (http://www.jquigley.com/files/talks/continuations.pdf) (PDF). p. 38.
10. Madore, David. "The Unlambda Programming Language" (http://www.madore.org/~david/programs/unlambda/). *www.madore.org*. Retrieved 19 June 2021.
11. Chris Barker, Continuations and the nature of quantification (http://www.semanticsarchive.net/Archive/902ad5f7/barker.continuations.pdf), 2002 Natural Language Semantics 10:211-242.
12. See for example Chris Barker, Continuations in Natural Language (http://www.cs.bham.ac.uk/~hxt/cw04/barker.pdf) (Continuations Workshop 2004), or Chung-chieh Shan, Linguistic Side Effects (http://homes.soic.indiana.edu/ccshan/brown/paper.pdf) (in "Direct compositionality, *ed. Chris Barker and Pauline Jacobson, pp. 132-163, Oxford University Press, 2007).*

Further reading

- Peter Landin. *A Generalization of Jumps and Labels* Report. UNIVAC Systems Programming Research. August 1965. Reprinted in Higher Order and Symbolic Computation, 11(2):125-143, 1998, with a foreword by Hayo Thielecke.
- Drew McDermott and Gerry Sussman. *The Conniver Reference Manual* MIT AI Memo 259. May 1972.
- Daniel Bobrow: *A Model for Control Structures for Artificial Intelligence Programming Languages* IJCAI 1973.
- Carl Hewitt, Peter Bishop and Richard Steiger. *A Universal Modular Actor Formalism for Artificial Intelligence* IJCAI 1973.
- Christopher Strachey and Christopher P. Wadsworth. *Continuations: a Mathematical semantics for handling full jumps* Technical Monograph PRG-11. Oxford University Computing Laboratory. January 1974. Reprinted in Higher Order and Symbolic Computation, 13(1/2):135—152, 2000, with a foreword by Christopher P. Wadsworth.
- John C. Reynolds. *Definitional Interpreters for Higher-Order Programming Languages* Proceedings of 25th ACM National Conference, pp. 717–740, 1972. Reprinted in Higher-Order and Symbolic Computation 11(4):363-397, 1998, with a foreword.
- John C. Reynolds. *On the Relation between Direct and Continuation Semantics* Proceedings of Second Colloquium on Automata, Languages, and Programming. LNCS Vol. 14, pp. 141–156, 1974.
- Reynolds, John C. (1993). "The discoveries of continuations" (https://www.cs.ru.nl/~freek/courses/tt-2011/papers/cps/histcont.pdf) (PDF). *Lisp and Symbolic Computation*. **6** (3/4): 233–248.
- Gerald Sussman and Guy Steele. *SCHEME: An Interpreter for Extended Lambda Calculus* AI Memo 349, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, December

1975. Reprinted in Higher-Order and Symbolic Computation 11(4):405-439, 1998, with a foreword.
- Robert Hieb, R. Kent Dybvig, Carl Bruggeman. *Representing Control in the Presence of First-Class Continuations* Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, pp. 66–77.
- Will Clinger, Anne Hartheimer, Eric Ost. *Implementation Strategies for Continuations* Proceedings of the 1988 ACM conference on LISP and Functional Programming, pp. 124–131, 1988. Journal version: Higher-Order and Symbolic Computation, 12(1):7-45, 1999.
- Christian Queinnec. *Inverting back the inversion of control or, Continuations versus page-centric programming* SIGPLAN Notices 38(2), pp. 57–64, 2003.

External links

- ACM SIGPLAN Workshop on Continuations 2011 (http://logic.cs.tsukuba.ac.jp/cw2011/) at the ICFP.
- Continuations for Curmudgeons (http://www.intertwingly.net/blog/2005/04/13/Continuations-for-Curmudgeons) by Sam Ruby
- Teach Yourself Scheme in Fixnum Days (https://web.archive.org/web/20100701180236/http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme-Z-H-15.html#node_chap_13) by Dorai Sitaram features a nice chapter on continuations.
- Continuations and Stackless Python (http://www.stackless.com/spcpaper.htm) by Christian Tismer
- On-line proceedings of the Fourth ACM SIGPLAN Workshop on Continuations (http://www.cs.bham.ac.uk/~hxt/cw04/cw04-program.html)
- On-line proceedings of the Second ACM SIGPLAN Workshop on Continuations (https://web.archive.org/web/20060501194520/http://www.brics.dk/~cw97/)
- Continuation, functions and jumps (http://www.cs.bham.ac.uk/~hxt/research/Logiccolumn8.pdf)
- http://okmij.org/ftp/continuations/ (http://okmij.org/ftp/continuations/) by Oleg Kiselyov
- https://wiki.haskell.org/Continuations
- Rhino With Continuations (https://web.archive.org/web/20100716142434/http://wiki.apache.org/cocoon/RhinoWithContinuations)
- Continuations in pure Java (https://web.archive.org/web/20100612212616/http://rifers.org/wiki/display/RIFE/Web+continuations) from the RIFE web application framework
- Debugging continuations in pure Java (http://rifers.org/theater/debugging_continuations) from the RIFE web application framework
- Comparison of generators, coroutines, and continuations, source of the above example (http://mail.python.org/pipermail/python-dev/1999-July/000467.html)