Continuations are a hidden gem found mostly in functional programming, but

can be used in languages that implement the use if first-class functions.

Wikipedia describes a continuation as "an abstract representation of the

control state of a computer program".[wiki-cont] Most of us would have to

ask "what in the world does that mean?" Well a continuation put more simply

is a way to represent/store the current state of the process being run. Many

people use continuations without even realizing it. A perfect example is

threads. Most widely used languages include the ability to create and

manipulate threads. When a new thread is created, the current state the

process is stored on that thread. So for example, a programmer creates a

thread and codes a simple Thread.sleep(1000); After 1000 milliseconds have

passed, the program comes back to the thread, and the state that was left

off at when the thread was run is restored.

Aside from threads, perhaps a continuation's most practical example within

web development. Traditional web development tries to control or store a

user's session with the use of session variables, or storing cookies, or

several other options. However, if not implemented wisely, this can lead to

significant problems. Say for example, you want to book a hotel in Rome. You

go to italianHotels.com, and find the perfect place, so you click book. When

it asks to confirm, you wife walks in and says she's already been to Rome

and wants to see someplace new. Rather than clicking back a few times, you

clone the tab, and enter in a new location. Venice looks great! Just as you

are about to book the hotel, your wife walks in and says she's sorry for not

letting you see Rome and wants to show you. So back you go to the other tab

you go and press "book" and confirm. Later, to your horror, you realize you

just booked a hotel in Venice! Since much of web development is not done

with continuations, this can be (and has been) a problem. The culprit is the

fact that cookies or session variables were stored on your computer, or

matching your IP address because that's the best way to correlate user with

state. With continuations however, session variables are stored implicitly.

This is done because continuations are storing state themselves by taking

advantage of the stack by using continuation-passing style.

Continuation-passing style, much like lazy programming is not a feature of a

language. Granted some languages do support the idea better than others, but

the process is not language specific. Typically when continuation-passing

style is demonstrated, it is done so with functional programming languages

like racket or scheme. Since the style is a bit of a paradigm shift, it is

best demonstrated in a succinct, functional way where each step of the

process can be shown. The basic idea behind continuation-passing style is

minimizing stack usage by never (or rarely) using it (in terms of frequency

f use, not memory usage). It does this by explicitly avoiding assignments

and returns. This perhaps is the most important part and requirement of

continuation passing style, and perhaps any programmer would stumble upon

its importance (and power) if required to program something under one single

constraint: no procedure is allowed to return to its caller--ever. Now if

that was the case, then even if the programmer didn't know what they were

doing, a smart one would likely by default begin to program using

continuation passing style. Perhaps an easier way to think about this is to

think of a return value of a function being a function itself instead of a

return. So if given the following identity function in "direct" style (or

rather, just non-continuation-passing style, in JavaScript syntax for

readability).

function id(x){

return x;

}

it would be obvious the purpose and behavior of the function because it is

already familiar to us. In order to convert this into a function that

utilizes continuation-passing style, we would convert the function to

function id(x, cc){

cc(x);

}

we see the basic idea behind what continuation-passing style does. Rather

than returning a value, it continues onto another function. If using true

continuation-passing style, rather than calling "cc(x);" we would call a

function closer to "cc(x, k);" where 'k' is some other function (commonly

called id()'s 'continuation'). In smaller programs, it is hard to see the

purpose or value behind using continuation-passing style. If we were to look

at a more complex function, however; its power would be more obvious. Take

the following formula, for example, to find the surface area of a cylinder.

2 pi r 2) + (2 pi r)\* h. Below are two examples of how to code it first in

direct style and second in continuation-passing style.

(define (surface rad height)

(+ (\* 2 (\* r r pi)) (\* h (\* 2 pi r)))

)

(define (surface r h cc)

(\* r r (lambda (rr)

(\* rr pi (lambda (rrpi)

(\* 2 rrpi (lambda (2rrpi)

(\* 2 pi (lambda(2pi))

(\* 2pi r (lambda(2pir)

(\* h 2pir (lambda(h2pir)

(+ h2pir 2rrpi cc)))))))))))

)

It is important to remember that all of these operators in the second

example '\*, +' would have to have been overridden to accept two operands,

and a function where the extra function is also a continuation. You can see

however that in contrast to our much simpler example, this continuation-

passing style is much different than say just a void function (though it is,

it is much more than that). In the first example the program would have to

break of the equation into pieces by itself--first going into the equation,

figuring out order of operations, using the stack a lot in order to compute

functions that have incomplete expressions. If the equation were to be

computed in direct style, the program will pop a '+' onto the stack, then a

'\*', then another '\*' then eventually, it will get a full expression, and

evaluate it popping it off the stack to be replaced by another partial

expression. When using continuation-passing style, one can easily see that

because of the way the program is structured, there is no need to be

popping and pushing things onto and off of the stack. It's sort of a

strange, inside-out-way of thinking because order of operations are

programmed in explicitly, and rather than returning values, the program

simply calls another procedure to act upon the recently modified data. Now

it is important to remember that even though it's doing all this with

minimal stack usage, the stack is still being used. It's actually being used

in a fantastic way that allows for a much much simpler way of following the

stack trace, and the state of the program. Since each process has another

process called inside of it rather than returning to the caller, the

entirety of programs are monitored and proceeded through as though it were a

list of processes rather than proceeding through a program as though it were

a tree.

In many cases, continuation-passing style is an optimization for either

memory or time, and often both. To address memory first, if we were to

define a function that computed factorial, normally, we would simply create

a recursive function that iterates through each value up to a number

provided demonstrated in the following code.

(define (factorial n)

(if (= n 1)

1

(\* n (factorial (- n 1)))

)

)

Even though this code is relatively simply, it very quickly throws the program into a complicated recursive loop. The computation might be as follows.

(factorial 4)

(\* 4 (factorial 3))

(\* 4 (\* 3 (factorial 2)))

(\* 4 (\* 3 (\* 2 (factorial 1))))

(\* 4 (\* 3 (\* 2 1)))

(\* 4 (\* 3 2))

(\* 4 6)

24

We simply want to find the factorial of 4 and it takes us 8 computations

using multiple stack frames! If we were to convert that function into

continuation-passing style, we would get this new bit of code.

(define (factorial2 n)

(local

([define (factIter counter result)

(cond

((> counter n) result)

(else (factIter

(+ 1 counter)

(\* result counter))))])

(factIter 1 1)))

Continuation-passing style allows us to create a recursive function that

behaves almost as though it were a loop. If we looked at the computation

that was taking place, we would see a very minimal dependence on that stack

which is similar to looping behavior.

(factorial 4)

(fact-iter 1 1 4)

(fact-iter 1 2 4)

(fact-iter 2 3 4)

(fact-iter 6 4 4)

(fact-iter 24 5 4)

24

By using continuation passing style, we free up the stack. The process does

not have the same behavior of growing and shrinking that our first recursive

example of factorial did, and actually behaves iteratively because

computation state is captured in a fixed number of variables (rather than

the process creating new variables to place on the stack at each recursive

call)As great as continuations sound though, they're not infallible. One

downside to supporting continuations is that it means you can't use the

stack to hold both control information and data. So things like C's auto

variables can't be put on the stack, which slows things down as you need to

allocate a frame for them somewhere out of heap memory. While not slow, it's

definitely much slower than just incrementing/decrementing (depending on

which way the stack goes) the stack pointer by a few dozen words.[squaks]

However, if creating auto variables and incrementing/decrementing registers

isn't something that you plan on doing very much, then using continuations

and continuation-passing style will definitely optimize your code.

Continuations are a hard concept to grasp initially. It doesn't help that it

makes code harder to read, and isn’t widely used. So if you’re on a consumer

software team with programmers that graduated from a tech-school, you should

use continuations sparingly. It would be a shame though because there is

real power in continuations. As we have talked about already, there are

clear benefits that arise from web development, but there are also

significant advantages that come from using in in simple computation--

especially when dealing with algorithms that involve a lot of recursion. It

has the power to use recursion with little more expense (if any) than

iteration. All in all, it is an important principle to understand and know.

[squaks] http://www.sidhe.org/~dan/blog/archives/000185.html

[wiki-cont] http://en.wikipedia.org/wiki/Continuation

[stg] http://www.st.informatik.tu-darmstadt.

de/pages/lectures/pl/ss07/slides/V8-continuations.pdf