Continuations

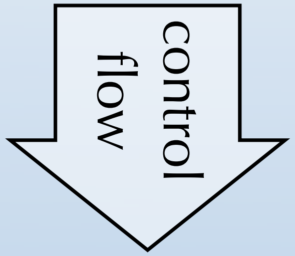
Michel Schinz
2007–05–04

Control flow of Web applications

The adder application

The following Scheme program asks for two numbers, and displays their sum – assuming the obvious definitions for `prompt-int` and `display-int`:

```
(let ((n1 (prompt-int "n1=")))
  (let ((n2 (prompt-int "n2=")))
    (display-int "n1+n2=" (+ n1 n2))))
```



Its control flow is completely obvious...

The adder Web application

Let’s assume that we want to take our adder application and turn it into a Web application, with the requirement that every interaction happens on a separate page.

That is, we want to use a first Web page to ask for the first number, a second page to ask for the second number, and a third one to display their sum.

If we suppose that we have the proper primitives at our disposal, this should be trivial:

```
(let ((n1 (web-prompt-int "n1=")))
  (let ((n2 (web-prompt-int "n2=")))
    (web-display-int "n1+n2=" (+ n1 n2))))
```

What about control flow?

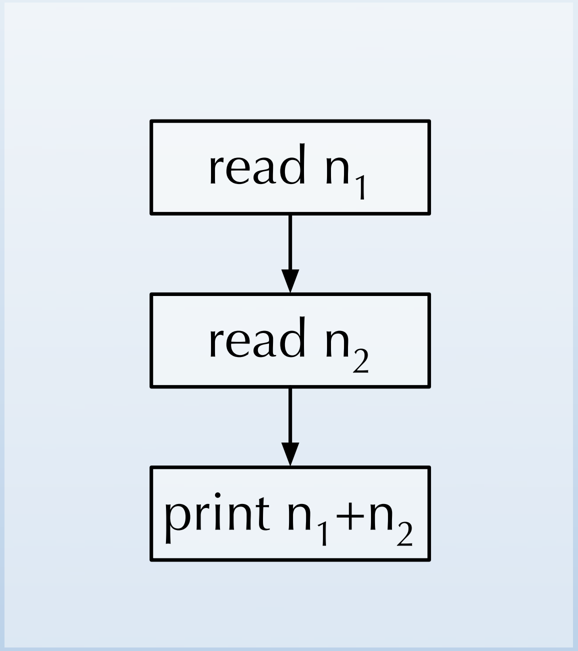
Browser power

When interacting with a Web application, the user has some very powerful means to alter its flow of control:

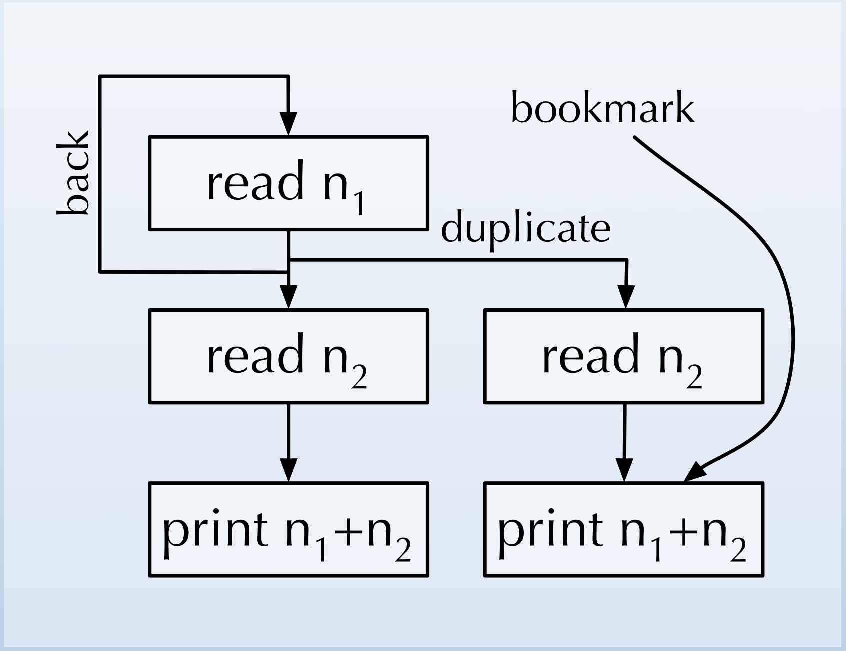
- the “back” button can be used to go back to a previous state,
- bookmarks can be used to take a snapshot of the execution state,
- URL copying can be used to duplicate state.

Control flow comparison

Normal application



Web application



Solutions for Web applications

Several solutions have been developed to deal with the unusual control flow of Web programs:

- do nothing and let the programmer deal with the complexity – *e.g.* PHP,
- tame the browser by disabling both the “back” button and cloning – *e.g.* JWIG,
- use continuations to please the user and the programmer – *e.g.* Seaside.

Continuations

Suspended computations

In our adder application, each time some data has to be obtained from the user, the execution of the program is suspended. It is then resumed as soon as the user submits the data.

The power of the Web version of our application comes from the fact that those suspended computations are given a name: the URL associated with them! The user can therefore manipulate those suspended computations at will. She can for example resume the same suspended computation several times, something that is not possible with the non-Web version of the application.

Continuations

A **continuation** is a data structure representing a suspended computation.

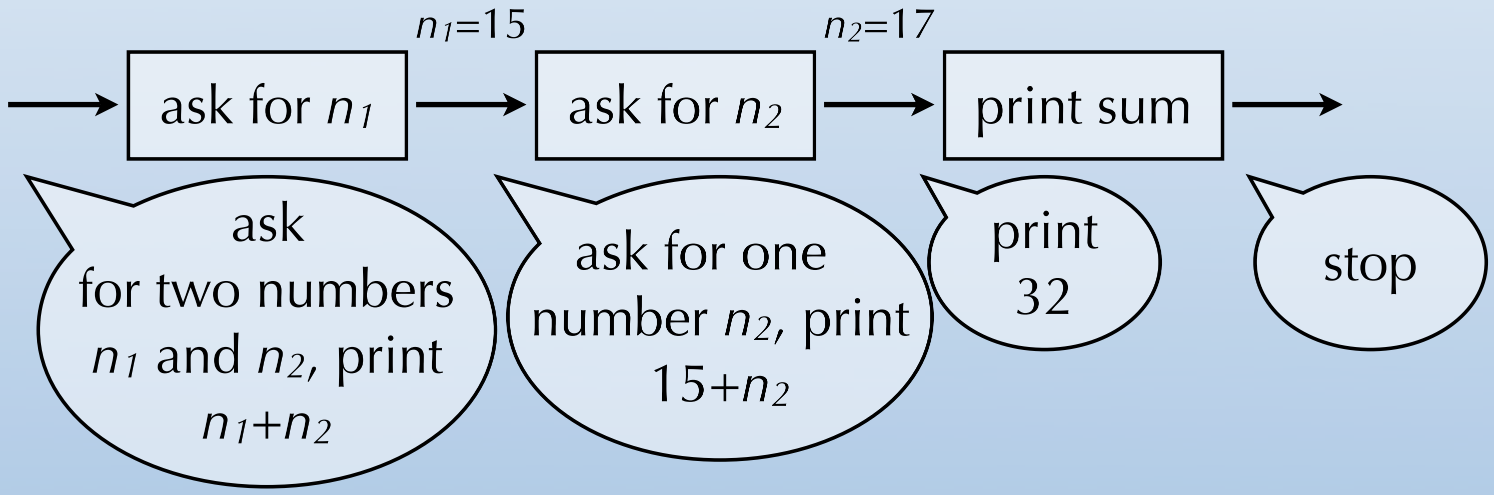
The main operation that can be performed on a continuation is **resuming** – or **throwing** – it. When a continuation *k* is resumed, the current execution of the program is *replaced* by the execution of *k*'s computation.

A continuation describes how to continue a suspended computation, hence the name.

Current continuation

At any given point during the execution of a program, it is possible to talk about the **current continuation**. This continuation describes what still needs to be done in order to complete the running program.

For example, imagine that our adder application is used to sum 15 and 17. How can the current continuation be described at various points of the execution?


n_1=15
n_2=17
ask for n_1
ask for n_2
print sum
ask for two numbers n_1 and n_2, print n_1+n_2
ask for one number n_2, print 15+n_2
print 32
stop

Continuations and the Web

In a Web application, execution is suspended each time a page is presented to the user. When the user proceeds – by clicking on a link or by submitting a form – execution is resumed.

In terms of continuations, this means that the current continuation is saved on the server whenever a page is displayed, and associated with a (unique) URL. That saved continuation is resumed later when the user requests its URL.

Functions and continuations

In any programming language, when a function *f* calls a function *g*, the execution of *f* is suspended while *g* is running, and resumed as soon as *g* is finished.

In terms of continuations, calling a function therefore consists in saving the current continuation, and then proceed with the execution of the called function. Returning from a function consists in restoring the most recently saved continuation.

In most languages, continuations can only be manipulated in that indirect fashion, through function calls and returns. However, some languages like Scheme offer **first-class continuations**, that is the ability to manipulate continuations like all other values.

Continuations in Scheme

Exposing continuations

How should (first-class) continuations be exposed to the programmer?

In an object-oriented language, continuations could be represented as a class, with methods to obtain the current continuation, or resume an existing continuation.

In a functional language, continuations can be represented as *functions*. Invoking such “continuation functions” resumes the associated continuation. This is how Scheme and several other languages expose continuations.

Continuations in Scheme

Scheme provides the primitive `call-with-current-continuation` – often abbreviated to `call/cc` – to obtain the current continuation.

This primitive expects a function as argument, and calls that function with the current continuation as argument, reified as a standard Scheme function.

If that function is invoked later, its continuation will be resumed, and replace the current continuation of the program. Example:

```
(call/cc (lambda (k) (k 10) 20))
  ⇒ 10
```

(reified) continuation

Understanding `call/cc`

To understand `call/cc`, it is useful to distinguish two cases:

1. If the continuation is not invoked, then execution proceeds like if `call/cc` was not present. Example:
   ```
   (call/cc (lambda (k) (+ 5 6)))
     ⇒ 11
   ```
2. If the continuation is invoked with some value `v`, then execution proceeds like if the call to `call/cc` returned immediately the value `v`. Example:
   ```
   (call/cc (lambda (k) (+ (k 5) 6)))
     ⇒ 5
   ```

`call/cc` examples

```
(call/cc (lambda (k) 10))
  ⇒ 10

(call/cc (lambda (k) (k 10)))
  ⇒ 10

(+ 1 (call/cc (lambda (k) (k 10) 20)))
  ⇒ 11

(call/cc (lambda (k) (k (k (k 20)))))
  ⇒ 20

(call/cc (lambda (k1)
           (+ (call/cc (lambda (k2) 5))
              (k1 6))))
  ⇒ 6
```

Example use: local return

Continuations can be used to return immediately from a function, like the `return` statement in Java.

This is achieved by obtaining the current continuation at the beginning of the function, and invoking it to return.

```
(define contains-negative?
  (lambda (l)
    (call/cc
     (lambda (return)
       (for-each (lambda (e)
                   (if (< e 0)
                     (return #t)))
                 l)
       #f))))
```

Example use: non-local return

Continuations can also be used to perform "non-local returns", similar to exceptions:

```
(define average
  (lambda (l throw)
    (if (null? l)
        (throw "empty list")
        (/ (fold + 0 l) (length l)))))

(define averages
  (lambda (ls)
    (let ((res (call/cc
                 (lambda (throw)
                   (map (lambda (l) (average l throw))
                        ls)))))
      (if (string? res)
          (error res)
          res))))
```

More advanced uses

Continuations are probably the most powerful control operator available in any language.

Apart from exceptions and returns, they can also be used to implement:

- threads,
- coroutines,
- C#-like iterators,
- etc.

Implementing `call/cc`

To implement `call/cc`, it must be possible to save the current continuation at some point, and restore it later. This can be achieved using two different techniques:

1. a low-level technique, which consists in saving and restoring the continuations that are maintained at run time during function calls and returns, and
2. a more high-level technique, which consists in transforming the source program to ensure that the current continuation is always explicitly represented as a function, and therefore easy to manipulate.

We will explore both techniques in turn.

Technique #1: machine continuations

Machine continuations

As explained earlier, all languages have continuations, as they are used to implement function calls:

- before a function call, the current continuation is saved,
- when a function returns, the most recently saved continuation is resumed.

However, these continuations – that we will call **machine continuations** – are usually not first-class values that can be manipulated by the programmer. The aim of `call/cc` is precisely to turn those continuations into first-class values!

call/cc

Assuming that our language is augmented with two new primitives to save and restore machine continuations, `call/cc` is easy to implement:

```
(define call/cc
  (lambda (f)
    (let ((cc (get-machine-continuation)))
      (f (lambda (r)
           (set-machine-continuation! cc)
           r)))))
```

It remains to be seen how those two primitives can be implemented.

Continuation representation

Where are machine continuations stored? In other words, where is the information necessary to return from a function call – return address, register contents – stored?

The answer depends on the machine being used, and on calling conventions.

In the simplest case, all that information is stored on the stack before a function call. Therefore, the frame pointer represents the continuation of a function!

In more complex cases, the information is stored both in the stack and in callee-saved registers. Some more work is necessary to save and restore continuations, but the basic idea is the same.

The stack

In a language without first-class continuations, the following two properties are true:

- continuations are saved and resumed in LIFO order,
- continuations can only be resumed once, which implies that they can be freed after having been resumed.

These properties make it possible to use a stack to store continuations. Unfortunately, they do not hold for languages with first-class continuations!

Implementations of such languages either abandon the stack completely and allocate all activation frames on the heap, or lazily copy those frames from the stack to the heap when continuations are saved.

Technique #2: continuation-passing style

Continuations “by hand”

What can we do if we want to use continuations but the language we use doesn’t offer them?

One idea is to transform the program to explicitly represent continuations using functions.

A program is said to be in **continuation-passing style** (**CPS**) if:

- all functions receive a continuation as an additional argument, and
- they invoke that continuation with their result instead of returning that result to the caller – *i.e.* no function ever returns.

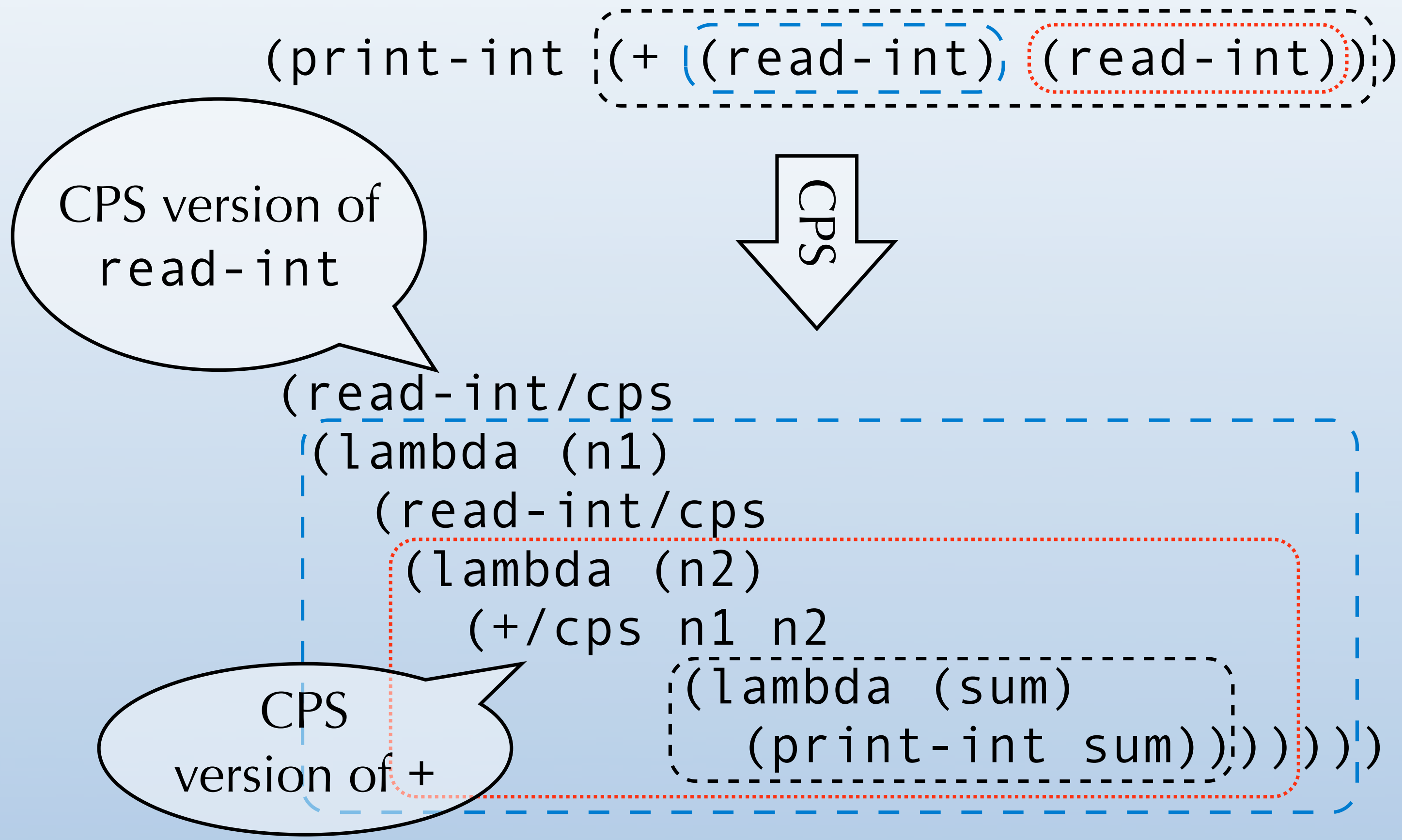
CPS example

To illustrate CPS, we will use the following simplified variant of our adder program:

```
(print-int (+ (read-int) (read-int)))
```

To transform this program to CPS, we need to use functions to represent the current continuation at all possible points of its execution: just after reading the first integer, after reading the second, etc.

CPS example



```
(print-int (+ (read-int) (read-int)))
```

```
(read-int/cps
 (lambda (n1)
   (read-int/cps
    (lambda (n2)
      (+/cps n1 n2
             (lambda (sum)
               (print-int sum)))))))
```

Primordial continuation

In the CPS version of our example, we cheated by using the normal version of `print-int`. Rigourously, we should have used the CPS version. But what continuation should it get?

More generally, what is the **primordial continuation**, *i.e.* the continuation of a complete program? A function halting execution is a good choice, which could be defined as follows given a `halt` primitive:

```
(lambda (res) (halt))
```

Defining `call/cc/cps`

As long as the program is not in CPS, it is not possible to define `call/cc`, which must be a primitive.

However, as soon as the program is in CPS, defining the CPS version of `call/cc` – *i.e.* `call/cc/cps` – is relatively simple.

To define it, it is useful to remember that the goal of `call/cc/cps` is to *reify* the current continuation by making it available as a standard (CPS) function.

That function, when applied to an argument `x`, should invoke the continuation that was current at the time when `call/cc/cps` was invoked, passing it `x`, and ignore the current continuation.

Defining `call/cc/cps`

The definition of `call/cc/cps` is:

reified continuation

```
(define call/cc/cps
  (lambda (f k)
    (f (lambda (res ignored-k) (k res))
       k)))
```

Notice how the reified continuation ignores the current continuation (`ignored-k`) and uses the captured one (`k`) instead.

CPS conversion for minischeme

CPS conversion

As we have seen, we can offer continuations by first transforming the program to CPS, and then providing an implementation of `call/cc/cps`.

Doing this transformation by hand is tiresome and error-prone, the compiler should do it for us!

This is the idea of **CPS conversion**, which will be presented here as a function $\mathcal{K}$ mapping minischeme terms to equivalent terms in CPS.

Simplified minischeme

To simplify the presentation, we will define CPS conversion for a version of minischeme restricted as follows:

- the body of a `lambda` expression is composed of a single expression,
- all functions take exactly one argument,
- there is no `let` expression, as it is only syntactic sugar:
  `(let ((`v_1 e_1`) …) `b_1` …)`
  ⇔ `((lambda (`v_1` …) `b_1` …) `e_1` …)`

Removing those restrictions is relatively easy, and left as an exercise.

Conversion outline

The basic idea of CPS conversion is to translate terms to functions that expect a continuation and invoke that continuation with the value of the term.

Therefore, *all* terms are translated to an expression with the following structure:

`(λ (k)` *some expression using* `k)`

shortcut for `lambda`

CPS for minischeme

```
𝒦[v] =
  (λ (k) (k  v))

𝒦[(if c t e)] =
  (λ (k) (𝒦[c] (λ (cᵥ) (if cᵥ (𝒦[t] k) (𝒦[e] k)))))

𝒦[(λ (x) b)] =
  (λ (k) (k (λ (x k₂) (𝒦[b] k₂))))

𝒦[(f x)] =
  (λ (k)
    (𝒦[f] (λ (fᵥ) (𝒦[x] (λ (xᵥ) (fᵥ  xᵥ  k))))))

𝒦[(p x y)] when p is a primitive =
  (λ (k)
    (𝒦[x] (λ (xᵥ) (𝒦[y] (λ (yᵥ) (p xᵥ yᵥ))))))
```

Example translation

```
(print-int (+ (read-int) (read-int)))
```

$$\Big\Downarrow \overline{\lambda}$$

```
(lambda (k1)
  ((lambda (k2) (k2 print-int/cps))
   (lambda (fv1)
     ((lambda (k3)
        ((lambda (k4)
           ((lambda (k5)
              (k5 read-int/cps))
            (lambda (fv2) (fv2 k4))))
         (lambda (xv1)
           ((lambda (k6)
              ((lambda (k7) (k7 read-int/cps))
               (lambda (fv3) (fv3 k6))))
            (lambda (yv) (k3 (+ xv1 yv)))))))
      (lambda (xv2) (fv1 xv2 k1))))))
```

much more complicated, but equivalent to what we would obtain by hand

Improving the translation

The previous examples make it clear that the translation we defined generates much more complex code than the one we obtained by hand earlier.

Other, more complicated translations to CPS can be defined in order to produce simpler code. We will not cover them here, however.

Summary

Continuations are the “ultimate” control operator. They can be used to implement many powerful concepts like threads, exceptions, etc.

Continuations can either be implemented in the virtual machine – basically by copying the stack – or by a transformation of the program to continuation-passing style (CPS), done by the compiler.

One important characteristic of CPS is that all calls are tail calls.