

# Functional languages

## Part III – continuations

Michel Schinz

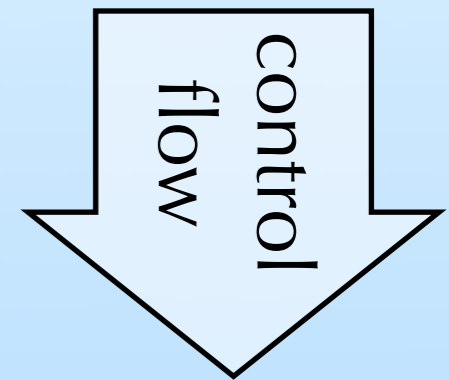
Advanced Compiler Construction / 2006-05-05

# Control flow of web applications

# The adder application

The following Scheme program asks for two numbers, and display 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 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.

# The adder Web application

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?

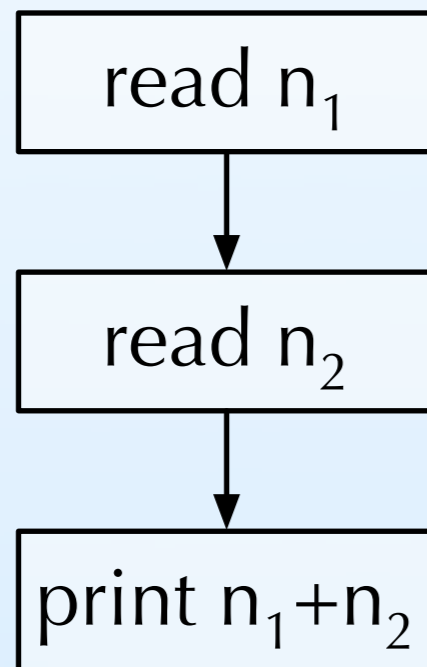
# Browser power

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

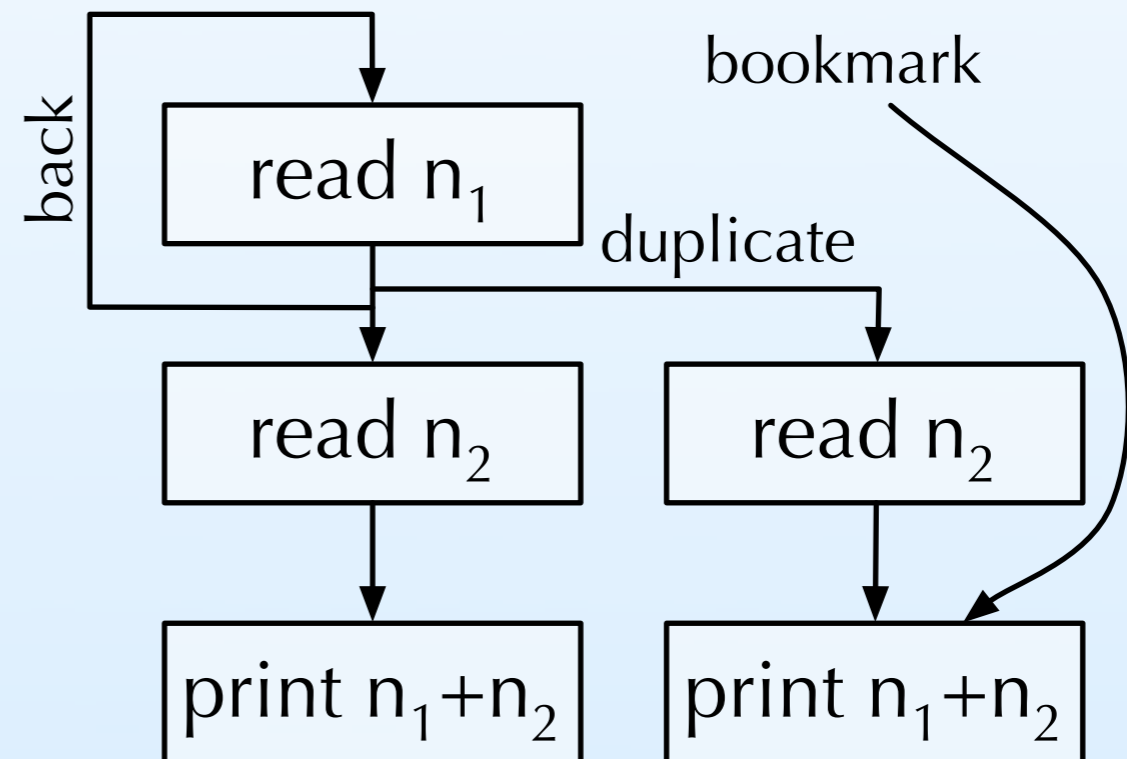
- the back button can be used to revert to a previous state,
- bookmarks can be used to take a snapshot of the execution state,
- URL copying can be used to duplicate state.

# Control flow comparison

## Normal application



## Web application



# Dealing with complex control flow

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,
- remove the power from the user by disabling both the back button and cloning – e.g. JWIG,
- use continuations to please the user *and* the programmer – e.g. recent Web frameworks.



# 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, and resumed as soon as the user has submitted 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 them at will.

# Continuations

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

The main operation which 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.

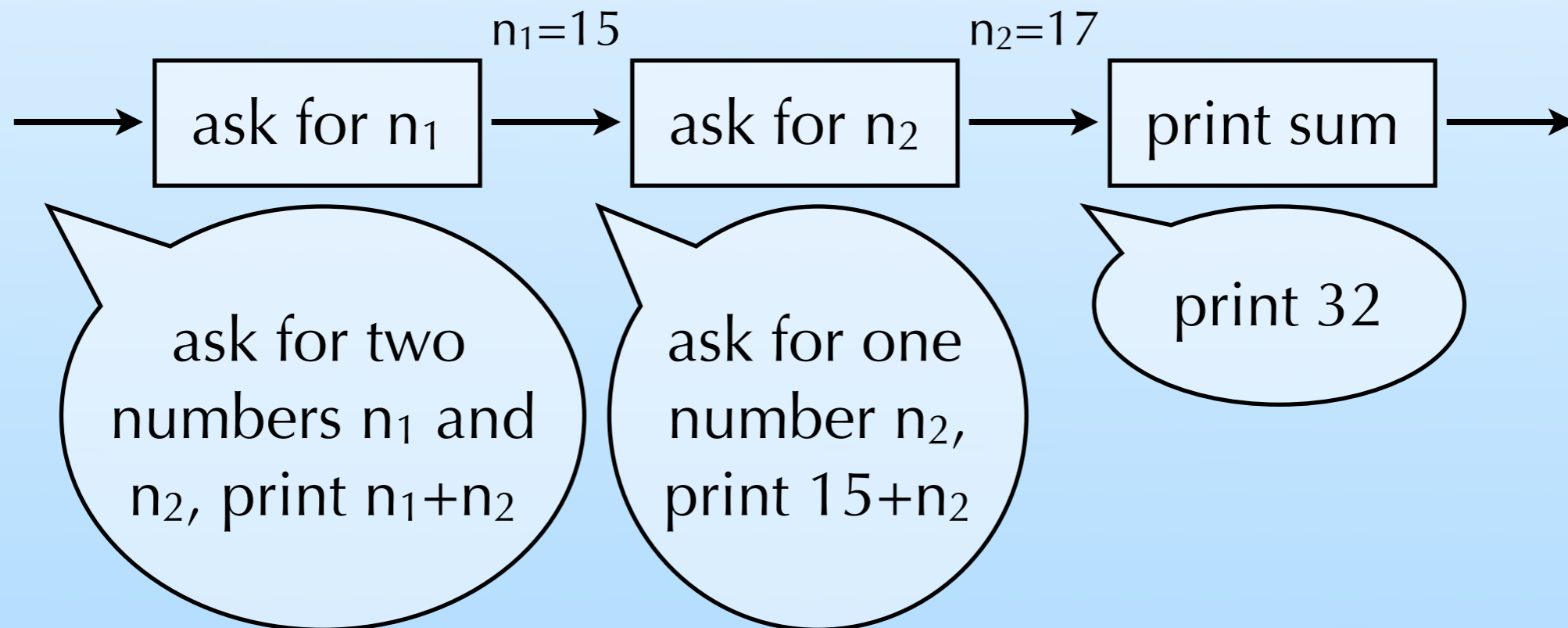
A continuation describes how to continue a suspended computation, hence the name.

# 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.

# Current continuation examples

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?



# Continuations and Web applications

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.

# A low-level view of continuations

What information should a continuation contain?

All the information which belongs to an execution state: the contents of registers – including the program counter – and the whole stack.

# Exposing continuations

How should continuations be exposed to the programmer?

In a functional language, it makes sense to represent continuations as functions. Invoking a continuation function resumes it.

This is how Scheme exposes continuations.



# Continuations in Scheme

# Continuations in Scheme

Scheme provides a primitive called `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.

# Continuation examples

current continuation

```
(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 (k) (k (k 3) (k 4) (k 5))))  
⇒ 3
```

# Continuation example

## Exiting from a loop

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

# Continuation example

## 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 of continuations

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

They can be – and are – used to implement exceptions and non-local returns as we have seen, but also threads, coroutines, C#-like iterators, etc.

# 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 transformed so that continuations are represented as functions is said to be in continuation-passing style.



# Continuation-passing style

More precisely, 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:

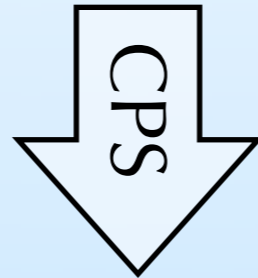
```
(print-int (+ (read-int) (read-int)))
```

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)))
```

CPS version of  
read-int



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

CPS  
version of +

# 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 – we assume a `$halt` primitive:

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

# Defining `call/cc`

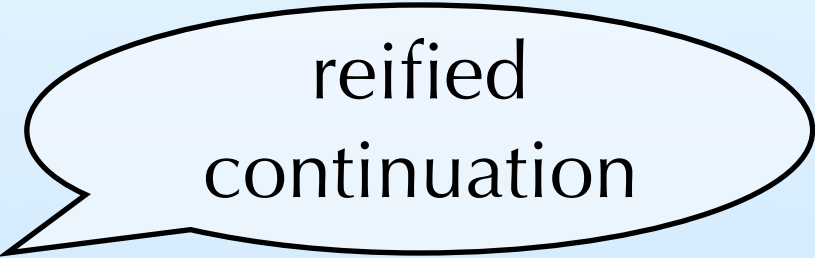
Once the program is in CPS, one important question remains: how can `call/cc` be defined?

The goal of `call/cc` 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 which was current at the time when `call/cc` was invoked – passing it  $x$  – and ignoring the current continuation.

# Defining call/cc

The definition of call/cc is:

```
(define call/cc
  (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 and tail calls

One important property of CPS is that *all* calls are tail calls.

Consequently, if tail calls are “optimised” by the compiler, then a program in CPS uses *no* stack!

There is no miracle, though: instead of existing as a data-structure managed by the run time system, the stack is represented by the heap-allocated closure(s) forming the current continuation.

# CPS conversion for minischeme



# CPS conversion

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

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 restricted version of minischeme:

- the bodies of `let` and `lambda` expressions are composed of a single expression,
- functions always take exactly one argument, and `let` binds exactly one value.

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 which 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:

$(\lambda (k) \text{ some expression using } k)$

lambda

# CPS for minischeme

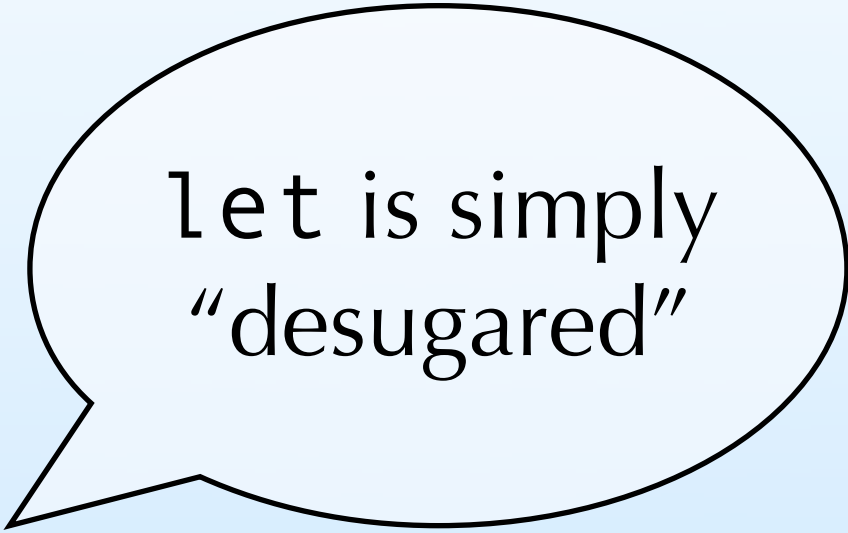
$\mathcal{K}[v] =$   
 $(\lambda (k) (k v))$

$\mathcal{K}[(if\ c\ t\ e)] =$   
 $(\lambda (k) (\mathcal{K}[c] (\lambda (c_v) (if\ c_v\ (\mathcal{K}[t]\ k)\ (\mathcal{K}[e]\ k))))))$

$\mathcal{K}[(\lambda (x)\ b)] =$   
 $(\lambda (k) (k (\lambda (x\ k_2) (\mathcal{K}[b]\ k_2))))$

$\mathcal{K}[(f\ x)] =$   
 $(\lambda (k)$   
 $(\mathcal{K}[f] (\lambda (f_v) (\mathcal{K}[x] (\lambda (x_v) (f_v\ x_v\ k))))))$

# CPS for minischeme



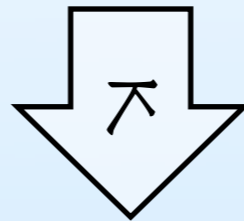
let is simply  
"desugared"

$$\kappa[(\text{let } ((v\ e))\ b)] =$$
$$\kappa[(\lambda (v)\ b)\ e]$$
$$\kappa[(\$+ x\ y)] =$$
$$(\lambda (k)$$
$$(\kappa[x]\ (\lambda (x_v)\ (\kappa[y]\ (\lambda (y_v)\ (\$+ x_v\ y_v))))))$$

Other primitives are translated like \$+

# Example translation

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



```
(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 explore 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, done by the compiler.

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