

# Functional languages

## Part II – tail calls

Michel Schinz

Advanced Compiler Construction / 2006-05-05

# Tail calls and their elimination

# Loops in functional languages

Several functional programming languages do not have an explicit looping statement. Instead, programmers resort to *recursion* to loop.

For example, the central loop of a Web server written in Scheme might look like this:

```
(define web-server-loop
  (lambda ()
    (wait-for-connection)
    (fork handle-connection)
    (web-server-loop)))
```

# The problem

Unfortunately, recursion is not equivalent to the looping statements usually found in imperative languages: recursive function calls, like all calls, consume stack space while loops do not...

In our example, this means that the Web sever will eventually crash because of a stack overflow – this is clearly unacceptable!

A solution to this problem must be found...

# The solution

In our example, it is obvious that the recursive call to `web-server-loop` could be replaced by a jump to the beginning of the function. If the compiler could detect this case and replace the call by a jump, our problem would be solved!

This is the idea behind tail call elimination.

# Tail calls

The reason why the recursive call of `web-server-loop` could be replaced by a jump is that it is the *last* action taken by the function :

```
(define web-server-loop
  (lambda ()
    (wait-for-connection)
    (fork handle-connection)
    (web-server-loop)))
```

Calls in terminal position – like this one – are called **tail calls**.

# Recursive tail calls


Tail calls which refer to the function which defines them are called (directly) **recursive tail calls**.

The tail call of our example is of that kind.


# Tail calls examples

In the functions below, which calls are tail calls?

```
(define map
  (lambda (f l)
    (if (null? l)
        l
        (cons (f (car l))
              (map f (cdr l))))))
```



```
(define fold
  (lambda (f z l)
    (if (null? l)
        z
        (fold f (f z (car l)) (cdr l)))))
```





# Tail call elimination

When a function performs a tail call, its own activation frame is dead, as by definition nothing follows the tail call.

Therefore, it is possible to first free the activation frame of a function about to perform such a call, then load the parameters for the call, and finally *jump* to the function's code.

This technique is called **tail call elimination** (or optimisation), abbreviated **TCE**.

# TCE example (1)

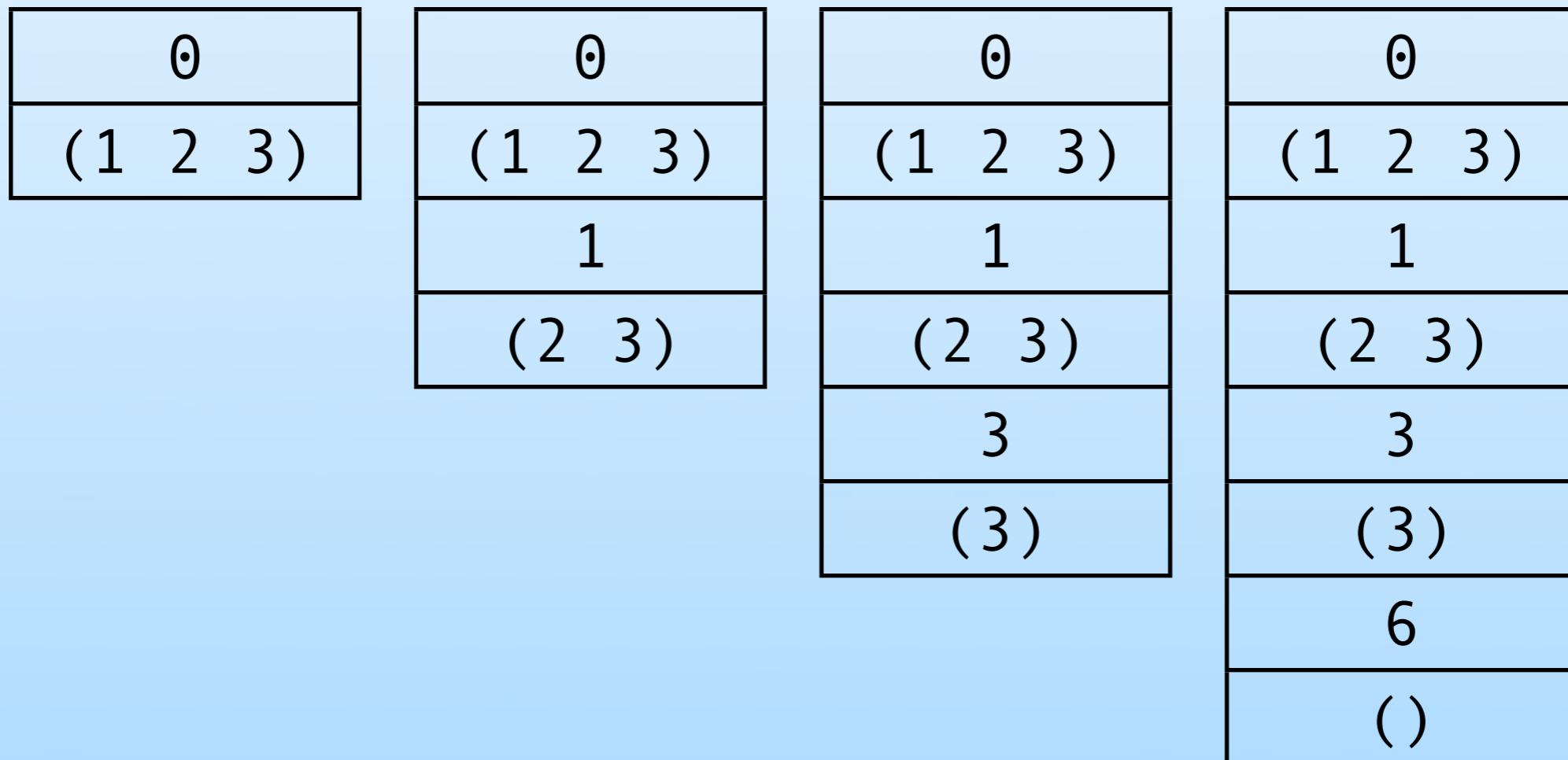
Consider the following function definition and call:

```
(define sum
  (lambda (z l)
    (if (null? l)
        z
        (sum (+ z (car l)) (cdr l)))))
(sum 0 (list3 1 2 3))
```

How does the stack evolve, with and without tail call elimination?

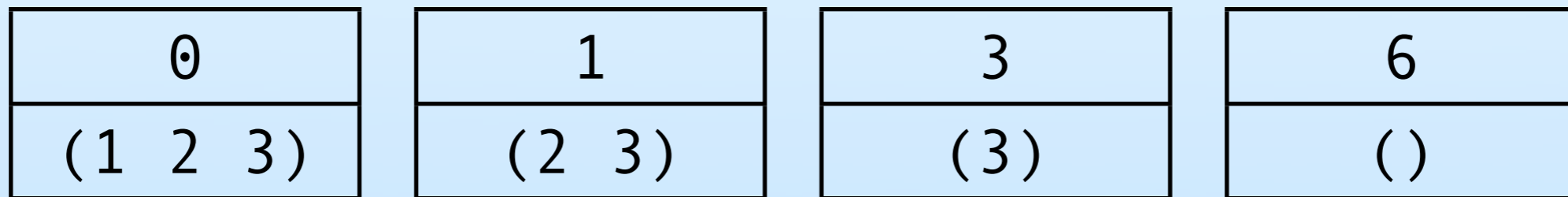
# TCE example (2)

Without tail call elimination, each recursive call to `sum` makes the stack grow, to accommodate activation frames.



# TCE example (3)

With tail call elimination, the dead activation frames are freed before the tail call, resulting in a stack of constant size.



# Tail call “optimisation”?

Tail call elimination is more than just an optimisation! Without it, writing an endless loop using recursion is simply impossible.

For that reason, full tail call elimination is actually *required* in some languages, e.g. Scheme.

In other languages, like C, it is simply an optimisation performed by some compilers in some cases.

# TCE in uncooperative environments

# TCE in uncooperative environments

When generating assembly language, it is easy to perform TCE, as the target language is sufficiently low-level to express the deallocation of the activation frame and the following jump.

When targeting higher-level languages, like C or the JVM, this becomes difficult – although recent VMs like .NET's support tail calls. We explore several techniques which have been developed to perform TCE in such contexts.

# Single function approach

The “single function” approach consists in compiling the whole program to a single function of the target language.

This makes it possible to compile tail calls to simple jumps within that function, and other calls to recursive calls to it.

This technique is rarely applicable in practice, due to limitations in the size of functions of the target language.



# Trampolines

With trampolines, functions never perform tail calls directly. Rather, they return a special value to their caller, informing it that a tail call should be performed. The caller performs the call itself.

For this scheme to work, it is necessary to check the return value of all functions, to see whether a tail call must be performed. The code which performs this check is called a **trampoline**.

# Baker's technique

**Baker's technique** consists in first transforming the whole program to a form called continuation-passing style (CPS). In CPS, *all* calls are tail calls.

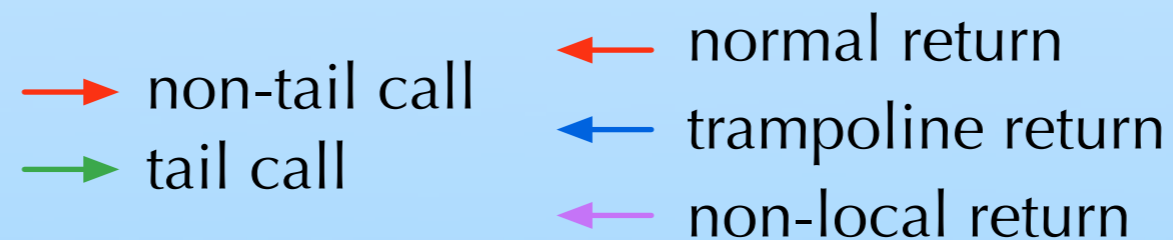
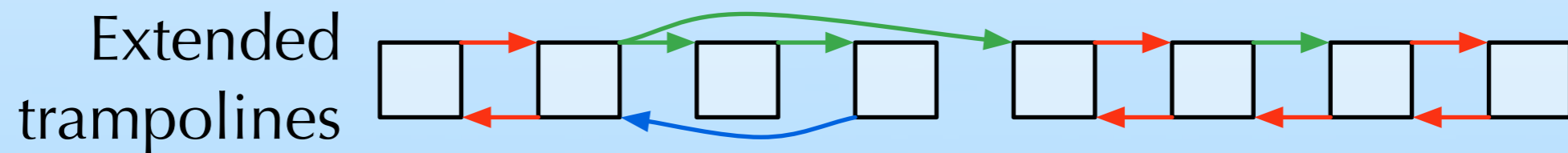
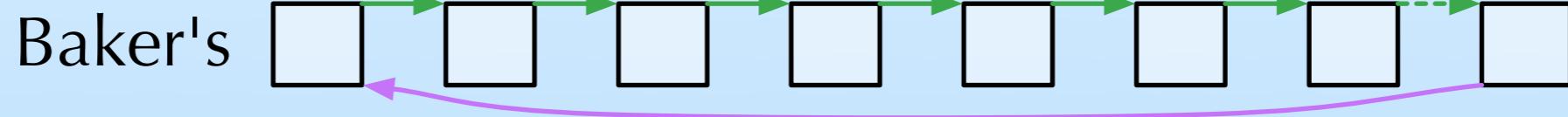
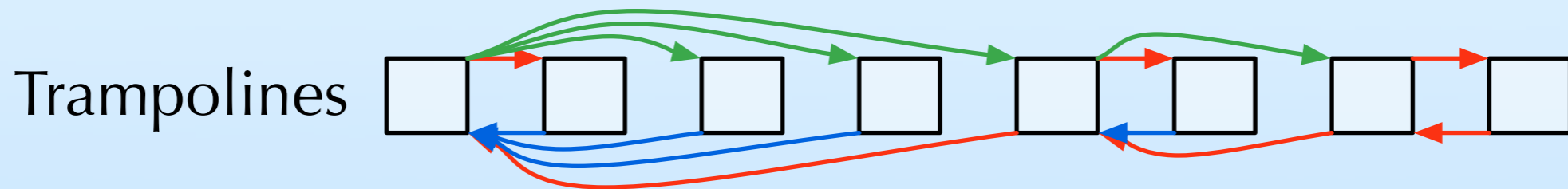
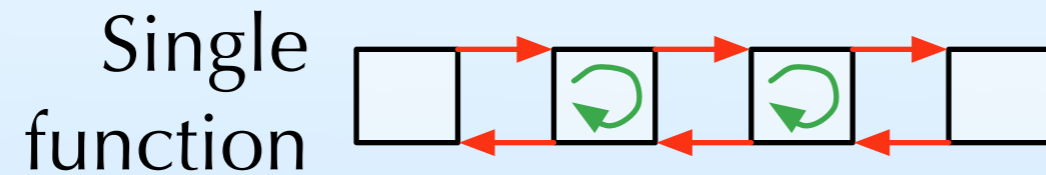
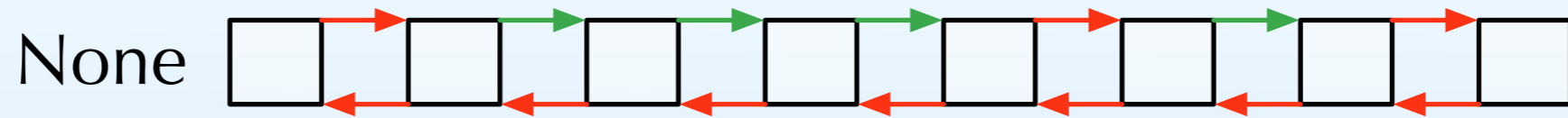
Consequently, it is possible to periodically shrink the *whole* stack. In C, this can be done using `setjmp/longjmp`, in Java by throwing an exception, etc.

# Extended trampolines

Extended trampolines trade some of the space savings of standard trampolines for speed.

Instead of returning to the trampoline on *every* tail call, the number of *successive* tail calls is counted at run time. When that number reaches a predefined limit  $l$ , a non-local return is performed to transfer control to a trampoline “waiting” at the bottom of the chain, thereby reclaiming  $l$  activation frames in one go.

# Techniques comparison



# Tail call elimination for minischeme

# TCE example

## Without TCE

```
loop:
  LINT R1 8
  ALOC R1 R1
  STOR R29 R1 0
  CMOV R29 R1 R0
  STOR R28 R29 4
  LINT R27 loop
  LINT R28 ret
  CMOV R31 R27 R0
ret:
  LOAD R28 R29 4
  LOAD R29 R29 0
  CMOV R31 R28 R0
```

```
(define loop
  (lambda ()
    (loop)))
```

unlink  
activation  
frame

## With TCE

```
loop:
  LINT R1 8
  ALOC R1 R1
  STOR R29 R1 0
  CMOV R29 R1 R0
  STOR R28 R29 4
  LINT R27 loop
  LOAD R28 R29 4
  LOAD R29 R29 0
  CMOV R31 R27 R0
```

# Implementing tail call elimination

Tail call elimination is implemented by:

- identifying the tail calls,
- compiling those tail calls specially, by deallocating the activation frame of the caller before jumping to the called function.

# Identifying tail calls

To identify tail calls, we first assume that all calls are marked with a unique number. We then define a function  $\tau$  which returns the marks corresponding to the tail calls.

For example, given the following expression:

```
(lambda (x)
  (if 1(even? x) 2(g 3(h x)) 4(h 5(g x))))
```

$\tau$  produces the set { 2,4 }.



# Identifying tail calls

$$\tau[(\text{lambda } (\text{args}) \text{ body}_1 \dots \text{ body}_n)] = \tau[\text{body}_n]$$

$$\tau[(\text{let } (\text{defs}) \text{ body}_1 \dots \text{ body}_n)] = \tau[\text{body}_n]$$

$$\tau[(\text{if } \text{cond} \text{ then } \text{else})] = \tau[\text{then}] \cup \tau[\text{else}]$$

$$\tau^m(e_1 \ e_2 \ \dots \ e_n) = \{ m \}$$

$$\tau[v] = \tau[n] = \emptyset$$

Note:  $\tau$  is only defined for functions, not for top-level expressions or definitions.

# Summary

Tail call elimination consists in compiling tail calls specially, so that the activation frame of the caller is freed before the called function is invoked.

This technique reduces memory usage and, more importantly, makes it possible to write loops using recursion without overflowing the stack.

Tail call elimination can be hard to implement when the target platform is uncooperative.