Tail call elimination

Michel Schinz

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**.

This particular tail call also happens to target the function in which it is defined. It is therefore said to be a **recursive tail call**.

Tail calls examples

In the functions below, all calls are underlined. Which ones 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

recursive tail call

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

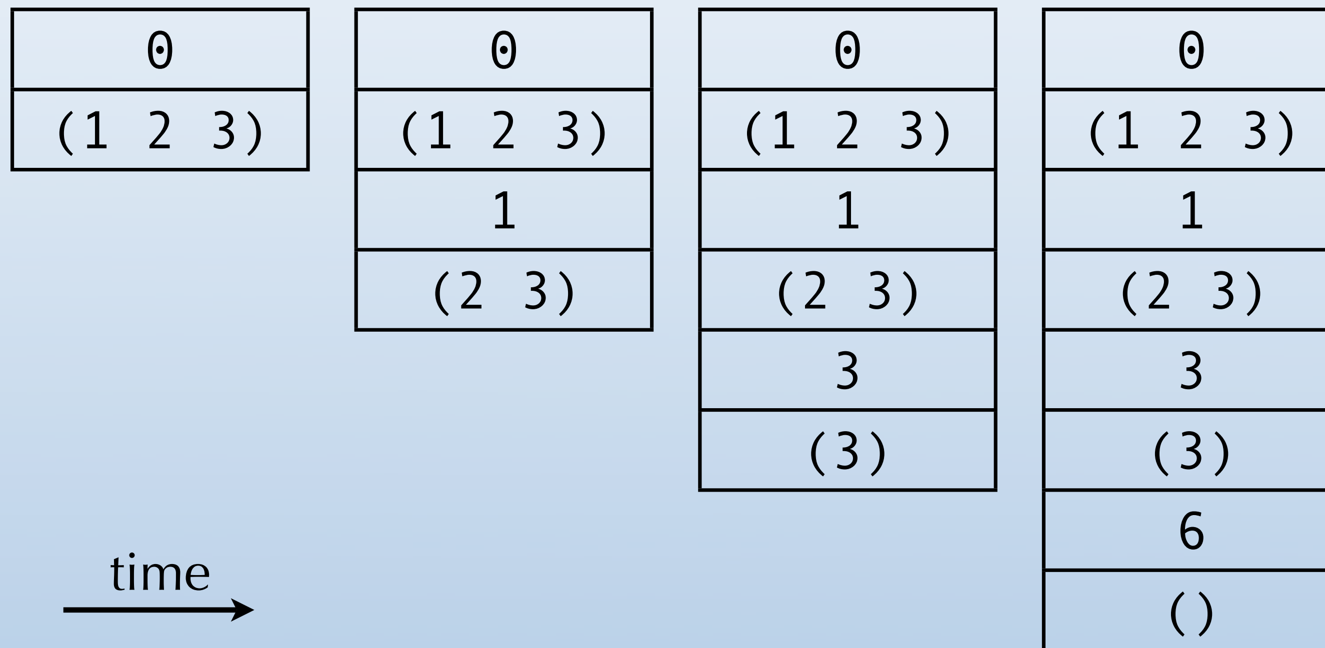
Consider the following function definition and call:

```
(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

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



TCE example

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

time

Tail call *optimisation*?

Tail call elimination is more than just an optimisation! Without it, writing a program that loops endlessly using recursion and does not produce a stack overflow 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.

Tail call elimination for minischeme

TCE example

```
(define succ
  (lambda (x)
    (add 1 x)))
(define add
  ...)
```

Without TCE

```
succ:
  ; ...
  LINT R27 add
  LINT R1 1
  LOAD R2 R30 8
  LINT R29 ret
  JMPZ R27 R0
ret:
  LOAD R29 R30 4
  LOAD R30 R30 0
  JMPZ R29 R0
```

With TCE

```
succ:
  ; ...
  LINT R27 add
  LINT R1 1
  LOAD R2 R30 8
  LOAD R29 R30 4
  LOAD R30 R30 0
  JMPZ R27 R0
```

unlink activation frame

Implementing TCE

Tail call elimination is implemented by:

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

We already know how to compile tail calls, but we did not explain yet how to identify them.

Identifying tail calls

To identify tail calls, we first assume that all calls are marked with a unique number. We then define a function τ that 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))))
```

τ produces the set { 2,4 }.

Identifying tail calls

τ[(`lambda` (args) $body_1$ … $body_n$)] = τ'[$body_n$]

where the auxiliary function τ' is defined as follows:

τ'[(`let` (defs) $body_1$ … $body_n$)] = τ'[$body_n$]

τ'[(`if` e_1 e_2 e_3)] = τ'[e_2] ∪ τ'[e_3]

τ'[^{m}(e_1 e_2 … e_n)] when e_1 is not a primitive = { m }

τ'[*anything else*] = ∅

Tail call elimination in uncooperative environments

TCE in various 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 that have been developed to perform TCE in such contexts.

Benchmark program

To illustrate how the various techniques work, we will use a benchmark program in C that tests whether a number is even, using two mutually tail-recursive functions.

When no technique is used to manually eliminate tail calls, it looks as follows. And unless the C compiler performs tail call elimination – like `gcc` does with full optimisation – it crashes with a stack overflow at run time.

```
int even(int x) {
  return x == 0 ? 1 : odd(x - 1);
}
int odd(int x) {
  return x == 0 ? 0 : even(x - 1);
}
int main(int argc, char* argv[]) {
  printf("%d\n", even(300000000));
}
```

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.

Single function approach in C

```
typedef enum { fun_even, fun_odd } fun_id;

int wholeprog(fun_id fun, int x) {
 start:
  switch (fun) {
  case fun_even:
    if (x == 0) return 1;
    fun = fun_odd;
    x = x - 1;
    goto start;
  case fun_odd:
    if (x == 0) return 0;
    fun = fun_even;
    x = x - 1;
    goto start;
  }
}
int main(int argc, char* argv[]) {
  printf("%d\n", wholeprog(fun_even, 300000000));
}
```

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**.

Trampolines in C

```
typedef void* (*fun_ptr)(int);
struct { fun_ptr fun; int arg; } resume;
void* even(int x) {
  if (x == 0) return (void*)1;
  resume.fun = odd;
  resume.arg = x - 1;
  return &resume;
}
void* odd(int x) {
  if (x == 0) return (void*)0;
  resume.fun = even;
  resume.arg = x - 1;
  return &resume;
}
int main(int argc, char* argv[]) {
  void* res = even(300000000);
  while (res == &resume)
    res = (resume.fun)(resume.arg);
  printf("%d\n",(int)res);
}
```

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, using a tail call counter (`tcc`) passed to every function. 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.

C's `setjmp` / `longjmp`

Extended trampolines are more efficient when a non-local return is used to free dead stack frames.

In C, non-local returns can be performed using the standard functions `setjmp` and `longjmp`, which can be seen as a form of `goto` that works across functions:

- `setjmp(b)` saves its calling environment in `b`, and returns `0`,
- `longjmp(b,v)` restores the environment stored in `b`, and proceeds like if the call to `setjmp` had returned `v` instead of `0`.

In the following slides, we use `_setjmp` and `_longjmp`, which do not save and restore the signal mask and are therefore much more efficient.

Extended trampolines in C

```
typedef int (*fun_ptr)(int, int);
struct { fun_ptr fun; int arg; } resume;
jmp_buf jmp_env;

int even(int tcc, int x) {
  if (tcc > TC_LIMIT) {
    resume.fun = even;
    resume.arg = x;
    _longjmp(jmp_env, -1);
  }
  return (x == 0) ? 1 : odd(tcc + 1, x - 1);
}
int odd(int tcc, int x) { /* similar to even */ }

int main(int argc, char* argv[]) {
  int res = (_setjmp(jmp_env) == 0)
    ? even(0, 300000000)
    : (resume.fun)(0, resume.arg);
  printf("%d\n",res);
}
```

Baker's technique

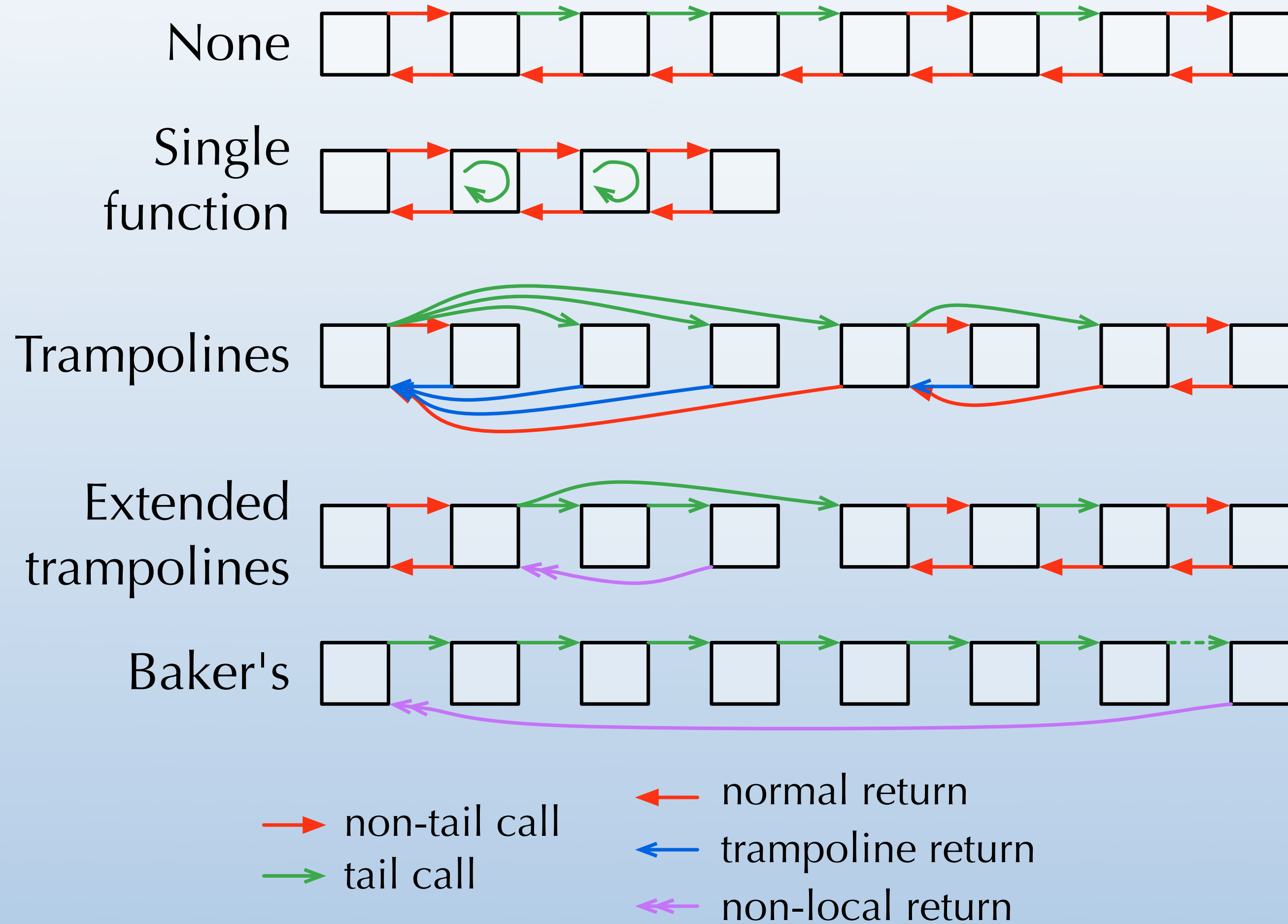
Baker's technique consists in first transforming the whole program to continuation-passing style.

One important property of CPS is that *all* calls are tail calls. Consequently, it is possible to periodically shrink the *whole* stack using a non-local return.

Baker's technique in C

```
typedef void (*cont)(int);
typedef void (*fun_ptr)(int, cont);
int tcc = 0;
struct { fun_ptr fun; int arg; cont k; } resume;
jmp_buf jmp_env;
void even_cps(int x, cont k) {
  if (++tcc > TC_LIMIT) {
    tcc = 0;
    resume.fun = even_cps;
    resume.arg = x;
    resume.k = k;
    _longjmp(jmp_env, -1);
  }
  if (x == 0) (*k)(1); else odd_cps(x - 1, k);
}
void odd_cps(int x, cont k) { /* similar to even_cps */ }
int main(int argc, char* argv[]) {
  if (_setjmp(jmp_env) == 0) even_cps(300000000, main_1);
  else (resume.fun)(resume.arg, resume.k);
}
void main_1(int val) { printf("%d\n", val); exit(0); }
```

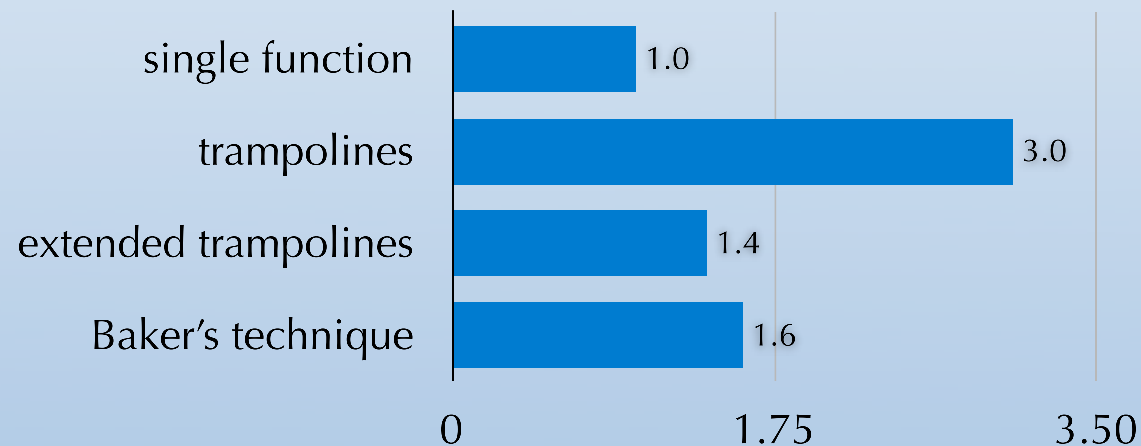
Techniques summary



Benchmark 1

To get an idea of the relative performance of the techniques just presented, we first compiled all benchmarks without optimisation, using `gcc` 4.01 on a PowerPC G4.

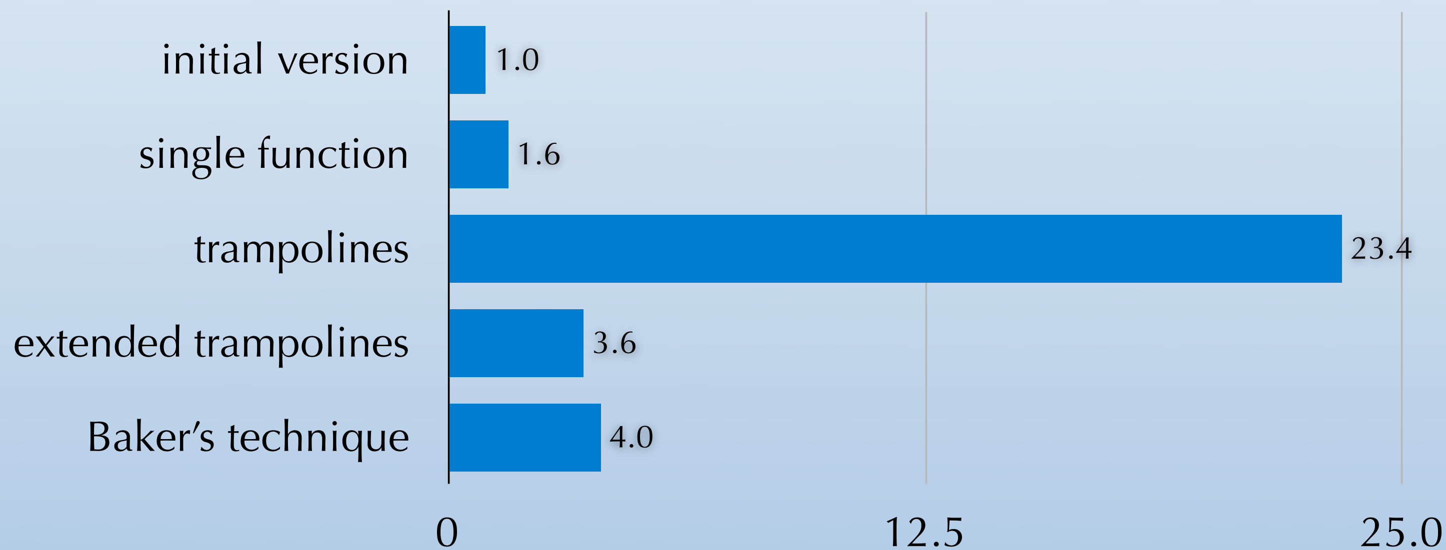
The version of the program that doesn't perform tail call elimination results in a stack overflow, and is therefore not included below. The normalised running times of the other versions are presented below.



Benchmark 2

As a second experiment, we compiled all programs with full optimisation. This enables the first version of the program to complete successfully, as `gcc` 4.01 performs at least some amount of tail call elimination.

The normalised running times are presented below.



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 makes it possible to write loops using recursion without overflowing the stack.

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