

## Tail calls

Advanced Compiler Construction  
Michel Schinz – 2025-04-10

## Tail calls (and their elimination)

## Functional loops

Often, functional languages do not offer loops.

So, programmers resort to recursion.

E.g., the central loop of an L<sub>3</sub> Web server might be:

```
(defrec web-server-loop
  (fun ()
    (wait-for-connection)
    (fork handle-connection)
    (web-server-loop)))
```

## Recursion problem

Problem:

- recursive calls consume stack,
- the web server will eventually crash (stack overflow).

But:

- the call to `web-server-loop` could be a jump!

So, the compiler should:

- detect such calls,
- replace them by jumps.

## Tail calls

Why can the recursive call of `web-server-loop` be replaced by a jump?

Because it is the last action taken by the function:

```
(defrec web-server-loop
  (fun ()
    (wait-for-connection)
    (fork handle-connection)
    (web-server-loop)))
```

Such a call in terminal position is a **tail call** (this one is also recursive, but not all are).

## Exercise

In the  $L_3$  functions below, which calls are tail calls?

```
(defrec list-map
  (fun (f l)
    (if (list-empty? l)
        l
        (list-prepend
         (f (list-head l))
         (list-map f (list-tail l))))))

(defrec list-fold-left
  (fun (f z l)
    (if (list-empty? l)
        z
        (list-fold-left f
                        (f z (list-head l))
                        (list-tail l)))))
```

## Tail call elimination

When a function performs a tail call, its own activation frame is dead: it won't be used anymore, as there is nothing to do after the call returns.

Therefore tail calls can be compiled as:

1. load the arguments for the callee,
2. free the activation frame of the caller,
3. *jump* (!) to the callee.

This is called **tail call elimination** (or **optimization**).

## TCE example

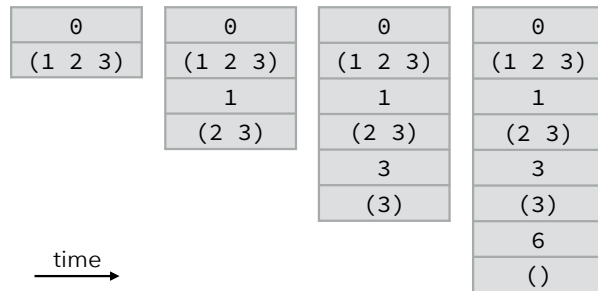
Consider the following function definition and call:

```
(defrec sum
  (fun (z l)
    (if (list-empty? l)
        z
        (sum (+ z (list-head l))
              (list-tail l)))))

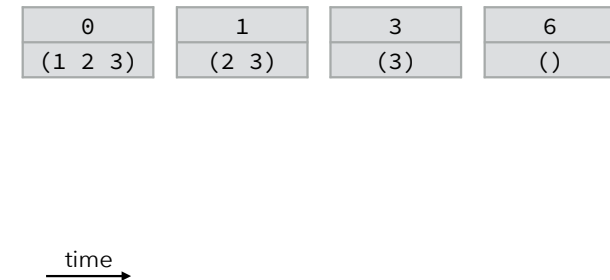
(sum 0 (list-make 1 2 3))
```

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

## Stack evolution (no TCE)



## Stack evolution (TCE)



## Tail call optimization?

Tail call elimination is more than just an optimization: one cannot write endless recursive loops without it.

Therefore:

- some language specifications (e.g. Scheme's) *require* that conforming implementations do TCE,
- other language specification (e.g. C's) don't, so compiler authors choose whether to do TCE or not.

## Tail calls in L<sub>3</sub>

## Translation of L<sub>3</sub> tail calls

Reminder: the basic translation from CL<sub>3</sub> to CPS/L<sub>3</sub> doesn't handle tail calls specially, and translates them sub-optimally.

E.g., the CL<sub>3</sub> term:

```
(letrec ((f (fun (g) (g)))) f)
```

gets translated to the CPS/L<sub>3</sub> term:

```
(letf ((f (fun (r1 g)
              (letc ((r2 (cnt (v)
                               (appc r1 v))))
                (appf g r2))))))
  f)
```

in which the tail call from *f* to *g* returns to *f* – since its return continuation is *r<sub>2</sub>* – instead of directly returning to its caller.

## Translation of L<sub>3</sub> tail calls

The improved translation from CL<sub>3</sub> to CPS/L<sub>3</sub> does handle tail calls specially, and optimizes them correctly.

With it, the same CL<sub>3</sub> term as before:

```
(letrec ((f (fun (g) (g)))) f)
```

gets translated to the CPS/L<sub>3</sub> term:

```
(letf ((f (fun (r1 g) (appf g r1))))
  f)
```

in which the tail call to *g* is optimized, in that it gets the same return continuation *r<sub>1</sub>* as *f* itself.

## Translation of L<sub>3</sub> tail calls

Non-tail calls are handled by  $\llbracket \cdot \rrbracket_N$ , as follows:

```
 $\llbracket (e\ e_1\ e_2\ \dots) \rrbracket_N\ C =$ 
 $\llbracket e \rrbracket_N(\lambda v\ \llbracket e_1 \rrbracket_N(\lambda v_1\ \llbracket e_2 \rrbracket_N(\lambda v_2\ \dots$ 
  (letc ((c (cnt (r) C[r])))
    (appf v c v1 v2 ...))))
```

while tail calls are handled by  $\llbracket \cdot \rrbracket_T$ , as follows:

```
 $\llbracket (e\ e_1\ e_2\ \dots) \rrbracket_T\ C =$ 
 $\llbracket e \rrbracket_N(\lambda v\ \llbracket e_1 \rrbracket_N(\lambda v_1\ \llbracket e_2 \rrbracket_N(\lambda v_2\ \dots$ 
  (appf v c v1 v2 ...))))
```

## Translation of CPS/L<sub>3</sub> tail calls

In the L<sub>3</sub> compiler, CPS/L<sub>3</sub> is just an IR, not the target language.

So, when generating target code, tail calls must be identified and translated appropriately.

This is trivial:

- a call where the callee gets the caller's return continuation is a tail call,
- all other calls are non tail calls.

# TCE in uncooperative environments

## TCE in various environments

Doing TCE requires support from the target language, to deallocate the stack frame and do the jump:

- no problem when generating machine code,
- much harder when generating C code, or JVM bytecode.

Several techniques exist to do TCE in these so-called "uncooperative environments".

## Benchmark program

The techniques will be illustrated using the simple C program below. If the C compiler does not do TCE, it crashes with a stack overflow.

```
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

**Single function** approach:

- compile the whole program to a single target function,
- tail calls become local jumps,
- other calls become recursive calls to that function.

Often difficult to apply in practice, due to limitations in the size of functions of the target language.

## Single function in C

```
typedef enum { fun_even, fun_odd } fun_id;
int wholeprog(fun_id fun, int x) {
    switch (fun) {
        case fun_even: goto even;
        case fun_odd:  goto odd;
    }

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

## Trampolines

Trampoline technique:

- functions never perform tail calls directly,
- rather, they return a special value to their caller – freeing their stack frame in the process,
- the caller does the call on their behalf.

This requires checking the return value of all function, 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 trampoline** technique:

- similar to trampolines, but trade some space for speed,
- do not return to trampoline on every tail call,
- rather, wait until a given number of successive ones happened, then return (non locally).

## Non-local returns in C

Extended trampolines require non-local returns.

In C, they can be performed using `setjmp` and `longjmp`, a kind 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 as if the call to `setjmp` had returned `v` instead of 0.

=

## 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:

- transform the whole program to continuation passing style (CPS),
- consequence: all calls are tail calls,
- so the *whole* stack can be shrunk periodically 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); }
```

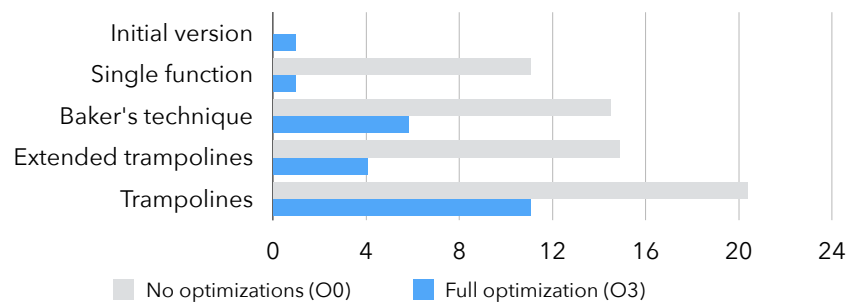
=

## Benchmark results

Processor: 2.3 GHz Intel Core i9

Compiler: clang 11.0.3

Optimization settings: -O0 and -O3



## Techniques summary

