PURDUE
UNIVERSITY

Tail Calls

Compilers: Principles And Practice

Tiark Rompf

Functional Loops

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 a functionnal way might look like this:

```
def webServerLoop() = {
  waitForConnecion();
  fork(handleConnection);
  webServerLoop()
}
```

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 webServerLoop 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 webServerLoop could be replaced by a jump is that it is the last action taken by the function :

```
def webServerLoop() = {
  waitForConnecion();
  fork(handleConnection);
  webServerLoop()
}
```

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**.

Exercise

In the MiniScala functions below, which calls are tail calls?

```
def listMap[T,U](f: T => U, l: List[T]): List[U] =
  if (l.isEmpty) Nil
  else f(l.head)::listMap[T,U](f, l.tail);

def listFoldLeft[T,U](f: (T,U) => T, z: T, l: List[U]): T =
  if (l.isEmpty) z
  else listFoldLeft[T,U](f, f(z, l.head), l.tail);
```

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 **optimization**), here abbreviated **TCE**.

TCE Example

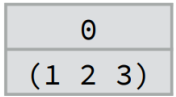
Consider the following function definition and call:

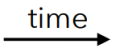
```
def sum(z: Int, l: List[Int]) =
  if (l.isEmpty) z
  else sum(z + l.head, l.tail);

sum(0, 1::2::3::Nil)
```

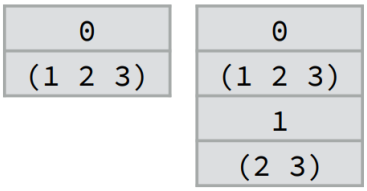
How does the stack evolve, with and without tail call elimination?

TCE Example

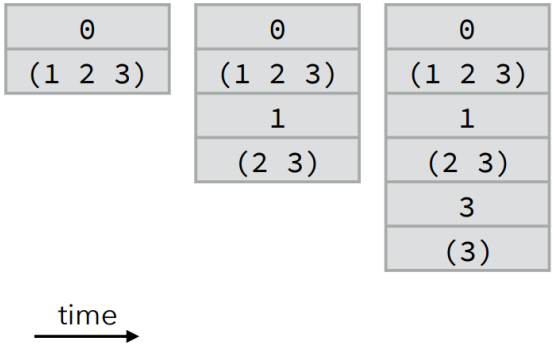




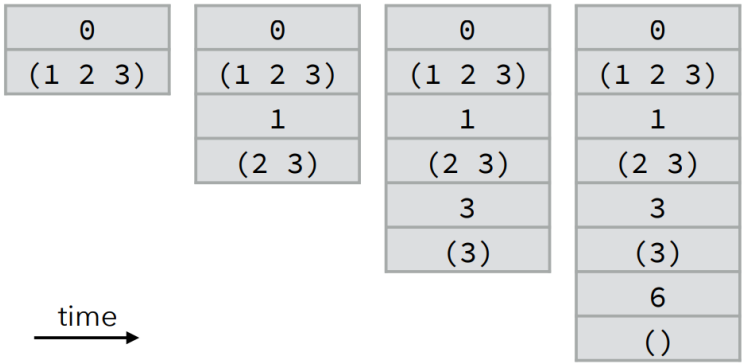
TCE Example



TCE Example

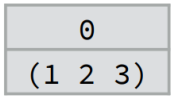


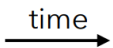
TCE Example



TCE Example

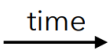
With tail call elimination, the dead activation frames are freed before the tail call, resulting in a stack of constant size.





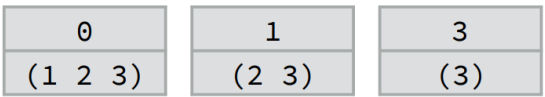
TCE Example

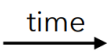
With tail call elimination, the dead activation frames are freed before the tail call, resulting in a stack of constant size.



TCE Example

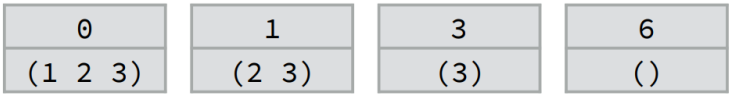
With tail call elimination, the dead activation frames are freed before the tail call, resulting in a stack of constant size.





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 optimization?

Tail call elimination is more than just an optimization!

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 optimization performed by some compilers in some or all cases.

Translation Of MiniScala Tail Calls

The "simple" translation from CMScala to CPS/MiniScala does not handle tail calls specially, and their translation is therefore suboptimal.

For example, the CMiniScala term:

```
def f(g: () => Int) = g(); f
```

gets translated to the CPS/MiniScala term:

```
deff f(c, g) = {
  defc j(r) = { c(r) };
  g(j)
};
f
```

in which the tail call from `f` to `g` returns to `f` - since its return continuation is `j` - instead of directly returning to its caller.

Translation Of MiniScala Tail Calls

The improved translation from CMScala to CPS/MiniScala does handle tail calls specially, and optimizes them correctly. With it, the same CMScala term as before:

```
def f(g: () => Int) = g(); f
```

gets translated to the CPS/MiniScala term:

```
deff f(c, g) = { g(c) };
f
```

in which the tail call to `g` is optimized, in that it gets the same return continuation `c` as `f` itself.

Translation Of MiniScala Tail Calls

The improved translation uses a different translation function for terms that are in tail position ($[\![.]\!]_T$), and uses it to translate function application efficiently.

Non-tail calls are handled by $[\![.]\!]_N$, as follows:

$$
\begin{aligned}
&[\![\ \texttt{e(e}_1\texttt{, e}_2\texttt{, ...)}\ ]\!]_N\ \texttt{C} = \\
&\quad [\![\texttt{e}]\!]_N(\lambda \texttt{v}([\![\texttt{e}_1]\!]_N(\lambda \texttt{v}_1([\![\texttt{e}_2]\!]_N(\lambda \texttt{v}_2\ \texttt{...})))));\\
&\quad \texttt{def}_c\ \underline{\texttt{c}}(\underline{\texttt{r}})\ \texttt{= \{ C[r] \};}\\
&\quad \texttt{v(c, v}_1\texttt{, v}_2\texttt{ ...)}
\end{aligned}
$$

while tail calls are handled by $[\![.]\!]_T$, as follows:

$$
\begin{aligned}
&[\![\ \texttt{e(e}_1\texttt{, e}_2\texttt{, ...)}\ ]\!]_T\ \texttt{c} = \\
&\quad [\![\texttt{e}]\!]_N(\lambda \texttt{v}([\![\texttt{e}_1]\!]_N(\lambda \texttt{v}_1([\![\texttt{e}_2]\!]_N(\lambda \texttt{v}_2\ \texttt{...})))));\\
&\quad \texttt{v(c, v}_1\texttt{, v}_2\texttt{ ...)}
\end{aligned}
$$

Translation Of CPS/MiniScala Tail Calls

In the MiniScala compiler, CPS/MiniScala is just an intermediate language, not the final target language.

Therefore, when translating CPS/L3 to virtual machine code, tail calls must be identified and translated appropriately.

Their identification is trivial: a CPS/MiniScala function call is a tail call iff it gets the return continuation of its enclosing function.

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. Let's 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 optimization — 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 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

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 (1)

```
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;
}
```

Trampolines in C (2)

```
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 passed to every function.

When that number reaches a predefined limit, a non-local return is performed to transfer control to a trampoline "waiting" at the bottom of the chain, thereby reclaiming several activation frames in one go.

Non-Local Returns in C

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

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

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 (1)

```
typedef void (*cont)(int);
typedef void (*fun_ptr)(cont, int);
int tcc = 0;
struct { fun_ptr fun; int arg; cont k; } resume;
jmp_buf jmp_env;
void even_cps(cont k, int x) {
  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(k, x - 1);
}
```

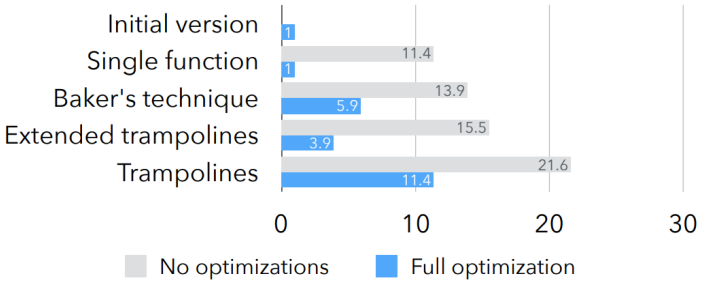
Baker's Technique In C (2)

```
void odd_cps(cont k, int x) { /* similar to even_cps */ }
void main_1(int res) { printf("%d\n", res); exit(0); }
int main(int argc, char* argv[]) {
  if (_setjmp(jmp_env) == 0) even_cps(main_1, 300000000);
  else (resume.fun)(resume.k, resume.arg);
}
```

Benchmark Results

The programs presented earlier were compiled with clang v503.0.38 and two different optimization settings (-O0 and -O3). The normalized running times observed on an Intel Core i5 are presented below.

Notice that the initial version compiled without optimization produces a stack overflow, hence the absence of timing.



Techniques Summary