ToiletPaper #93

Tail Call Elimination

Author: Hannes Lerchl / Senior Software Architect / Business Division New Business

Problem

Functional programming loves functions and recursions in all varieties.

<fud>But these many small function calls are very expensive, aren’t they? And what if my stack overflows during a recursion?</fud>

Solution

Various compilers/interpreters or languages (Scheme requires this from the interpreter) provide tail call elimination, e.g. Lisp, Scheme, Lua, Erlang, Elixir, C/C ++ and to a certain extent also Scala and Kotlin. However, you must adapt your code to trigger this feature. What is it and how does it work?

Example

Let's have a closer look at a trivial example of a recursive function:

a) "Naïve" recursion

```
int factorial(int n) {
    if (n == 0) return 1;
    return n * factorial(n - 1);
}
```

b) Recursion with tail calls

```
int f_hlp(int n, int acc) {
    if (n == 0) return acc;
    return f_hlp(n-1, acc*n);
}
 int factorial_2(int n) {
    return f_hlp(n, 1);
}
```

Version a) is quite close to the mathematical definition and would probably occur spontaneously to most people. Version b) seems to be unnecessarily complicated and needs a helper function. So why using it?
The small difference is that the functions in b) end with a function call (the “tail call”). Its result is not processed any further but simply forwarded to the caller (in (a) it is used for a subsequent multiplication). This ultimately means that no new stack frame needs to be created, but the existing one can simply be overwritten. The call degenerates to a *goto*; the return address is already on the stack.
The compiler transforms the recursion into a loop, so that we don't have to misshape our code with ugly *for loops*. At runtime, the code acts like a loop (in terms of speed and stack usage) but remains slim and readable.

Further Aspects

- Although recursion is the standard example here, it is called "tail *call* elimination" and not "tail *recursion* elimination" -- several different functions can be involved
- When debugging, you should keep in mind that the stack trace is very shortened and can seem strange or useless.
- If your "language of choice" doesn't support this feature, you can
    - ignore it and use recursion anyway
    - convert the recursion into a loop (which is usually possible … except for Ackermann function and similar)
    - use a "trampoline" (spoiler: heavy going, sloooow)
- I leave it as a practice up to the reader to implement b) with a fold
- By the way, this topic was already covered in 1977 by Guy Steele in one of his “Lambda Papers“ ("The Ultimate goto"): http://dspace.mit.edu/bitstream/handle/1721.1/5753/AIM-443.pdf
- Outtakes: When I wanted to test b) with gcc, I wrote a main, which only consisted of printf("5! = %i\n", factorial_2(5)). I was quite surprised when the call was completely gone in the assembler code and the “stupid compiler” simply pushed 120 to edx and called printf with it.
- Outtakes 2: So I just threw out the main and compiled both functions: gcc turns both versions a) and b) into exactly the same loop (three opcodes long). My whole example is kind of pointless. These damn low-level languages …