ToiletPaper #93

Tail Call Elimination

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

✕ Problem

Funktionale Programmierung liebt Funktionen und Rekursionen in allen Spielarten.

<fud>Sind aber nicht diese vielen kleinen Funktionsaufrufe sehr teuer? Und was, wenn bei einer Rekursion mein Stack überläuft? </fud>

✓ Lösung

Diverse Compiler/Interpreter oder Sprachen (Scheme fordert das vom Interpreter) bieten tail call elimination (z.B. Lisp, Scheme, Lua, Erlang, Elixir, C/C++ und etwas eingeschränkt auch Scala und Kotlin). Allerdings muss man seinen Code ein wenig kneten, um dieses Feature "zu triggern". Was ist das und wie funktioniert das?

➔ Beispiel

Schauen wir uns mal ein banales Beispiel einer rekursiven Funktion an:

a) "Naive" Rekursion

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

b) Rekursion mit 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);
}
```

Variante a) ist recht nah an der mathematischen Definition und würde wohl den meisten spontan einfallen. Variante b) wirkt unnötig kompliziert und braucht eine Hilfs-Funktion. Wozu?
Der kleine feine Unterschied ist, dass die Funktionen rechts jeweils mit einem Funktionsaufruf enden (dem "tail call"). Dessen Ergebnis wird nicht weiterverarbeitet sondern einfach zum Aufrufer weitergeleitet (bei (a) wird hingegen noch multipliziert). Das bedeutet letztendlich, dass kein neuer Stackframe aufgebaut werden muss, sondern der bestehende einfach überschrieben werden kann. Der Call degeneriert quasi zu einem GOTO; die Rücksprung-Adresse liegt eh schon auf dem Stack. Der Compilier formt die Rekursion also zu einer Schleife um, so dass wir unseren Code nicht mit hässlichen for-Schleifen verunstalten müssen. Der Code ähnelt zur Laufzeit einer Schleife (von Geschwindigkeit und Stacknutzung her), bleibt aber schön schlank und lesbar.

✚ Weiterführende Aspekte

- Obwohl Rekursion hier das Standard-Beispiel ist, heißt es "tail *call* elimination" und nicht "tail *recursion* elimination" -- das geht also auch mit verschiedenen beteiligten Funktionen
- Beim Debugging sollte man das übrigens im Hinterkopf behalten: Der Stacktrace ist stark verkürzt und kann einem befremdlich oder nutzlos erscheinen.
- Wenn die "Sprache der Wahl" dieses Feature nicht unterstützt, kann man
  - das ignorieren und trotzdem Rekursion einsetzen
  - die Rekursion in eine Schleife umwandeln (was meistens geht … außer bei Ackermann-Funktion und Konsorten)
  - ein "Trampolin" nutzen (Spoiler: schwere Kost, laaang-saaam)
- Es sei dem Leser zur Übung überlassen, Beispiel b) mit einem Fold zu implementieren
- Dieses Thema wurde übrigens schon 1977 von Guy Steele in einem seiner "Lambda Papers" behandelt ("The Ultimate GOTO"): http://dspace.mit.edu/bitstream/handle/1721.1/5753/AIM-443.pdf
- Outtakes: Als ich b) mit dem gcc durchtesten wollte, hab ich eine Main geschrieben, die nur aus printf("5! = %i\n", factorial_2(5)); besteht. Groß waren die Augen, als im Assembler-Code der Call ganz weg war. Stattdessen hat "der Drecks-Compiler" einfach 120 nach edx geschoben und damit printf gerufen.
- Outtakes 2: Also einfach die Main raugeworfen und beides kompiliert: Der gcc macht aus beiden Varianten exakt die gleiche (drei Opcodes lange) Schleife. Mein ganzes Beispiel ist zum Teufel. Diese verdammten Low-Level-Sprachen …