GHC/Memory Management

From HaskellWiki

< GHC

Haskell computations produce a lot of memory garbage - much more than conventional imperative languages. It's because data are immutable so the only way to store every next operation's result is to create new values. In particular, every iteration of a recursive computation creates a new value. But GHC is able to efficiently manage garbage collection, so it's not uncommon to produce 1gb of data per second (most part of which will be garbage collected immediately). So, you may be interested to learn how GHC does such a good job.

1 Motivating examples

Through the article, we will use two motivating examples. The first one just computes some value:

```
factorial 0 acc = acc
factorial n acc = factorial (n-1) $! (acc*n)
```

Note that we have used accumulator with strict evaluation in order to suppress the default laziness of Haskell computations - this code really computes new n and acc on every recursion step.

Our second example produces a large list:

```
upto i n | i<=n      = i : upto (i+1) n
         | otherwise = []
```

So, we have two examples - one that produces just one value but leaves a lot of garbage and another producing a lot of live data.

2 Garbage collection

Haskell's computation model is very different from that of conventional mutable languages. Data immutability forces us to produce a lot of temporary data but it also helps to collect this garbage rapidly. The trick is that immutable data NEVER points to younger values. Indeed, younger values don't yet exist at the time when an old value is created, so it cannot be pointed to from scratch. And since values are never modified, neither can it be pointed to later. This is the key property of immutable data.

This greatly simplifies garbage collection (GC). At anytime we can scan the last values created and free those that are not pointed to from the same set (of course, real roots of live values hierarchy are live in the stack). It is how things work: by default, GHC uses generational GC. New data are allocated in 512kb "nursery". Once it's exhausted, "minor GC" occurs - it scans the nursery and frees unused values. Or, to be exact, it copies live values to the main memory area. The fewer values that survive - the less work to do. If you have, for example, a recursive algorithm that quickly filled the nursery with generations of its induction variables - only the last generation of the variables will survive and be copied to main memory, the rest will be not even touched! So it has a counter-intuitive behavior: the larger percent of your values are garbage - the faster it works.

3 Further reading

Parallel Generational-Copying Garbage Collection with a Block-Structured Heap (2008) (http://community.haskell.org/~simonmar/bib/parallel-gc-08_abstract.html)

Retrieved from "https://wiki.haskell.org/index.php?title=GHC/Memory_Management&oldid=58816"

- This page was last modified on 15 September 2014, at 01:26.
- Recent content is available under a simple permissive license.