

Re: The Lisp CPU

Source: <http://coding.derkeiler.com/Archive/Lisp/comp.lang.lisp/2004-08/1052.html>

From: rem642b_at_Yahoo.Com (*RobertMaas_at_YahooGroups.Com*)

Date: 08/16/04

Date: Sun, 15 Aug 2004 16:45:11 -0700

> *From: Frank Buss <fb@frank-buss.de>*
> > *Not sure I understand; alists will be pretty fast for the typical*
> > *case.*
> *not so fast as direct access to a fixed memory position, because you*
> *have to traverse the list, which is too slow. I didn't mention that I*
> *want to built a real-time Lisp CPU with guaranteed response time.*

It's impossible to guarantee response time if you allow any program whatsoever to be run, because some programs do so many operations it's impossible for them to run as fast as they are supposed to, such as bubble-sort on a very large array. So the only thing you can guarantee is that for any given algorithm there's a way to program it to run quickly on your CPU (as well as *other* ways to program it that don't run quickly at all). There's a simple fix for a-list searches taking too long, which can be applied to most programs that are too slow for that reason, so you shouldn't worry about it right now before you have anything working at all. So don't fall into the trap of PREMATURE OPTIMIZATION, where you worry so much about it, and spend so much time tweaking your CPU to be faster here and faster there, that you never ship a product, never even have a working prototype to test. Wait until you have something working, then if it runs too slow, profile it to see where it's taking too much time.

So what's the simple trick? If profiling shows there's a tight loop where some particular variable is reference millions of times, but that variable was stuck on the a-list a long time, so it's way down the list, with more recent bindings ahead of it, so every time you reference that variable you do a long search again: Re-bind that particular variable around your tight loop so it's now at the head of the a-list! All the other variables, which you seldom reference, will be slightly slower, while that one most-commonly-used variable will be very fast. Because you're using dynamic binding, the following code

```
(let ((x x))  
  ...)
```

won't change which variables are accessible nor their values, merely will make x faster-access.

But what if you have several such variables? Rate them most-crucial second-most-crucial etc., and re-bind them all with the most-crucial innermost.

But what if there are a whole bunch of such variables? So the sixth one still takes a while. In that case you might need to re-write the tight loop to access slots in a single structure rather than separate variables. So instead of variables x1 x2 x3 x4 x5 x6, you'd have (caar xstr) (cdar xstr) (caadr xstr) (caddr xstr) (cdadr xstr) (cdddr xstr), where x1 and x2 are very fast and the rest are slightly slower. If you know how to build a Huffman code, you can use a similar method to optimize the total cost of a set of variables that have known (profiled) frequencies of usage.

(I'm assuming you've already cleaned up any profiled-critical code that is blatantly poorly written such as large bubble sorts etc., to where a-list searching for variable lookups is the remaining slowdown)

Note if most of the time you write your code as if using lexical variables, i.e. the index variable of a loop is bound right there around the loop rather than re-using some binding from way way up the a-list, at least your index variable will never be slow. But the upper-limit of a loop could have been bound long ago, in which case re-binding around the tight loop it might make a good speedup. But even then, typically you pass the upper limit as a parameter, forcing it to be re-bound, rather than referring to it via dynamic scope, so again no problem in the first place usually. For example, a typical usage might be to fetch a distantly-bound variable *once* and pass it as argument to function where it serves as upper bound on loop. So there's *one* very slow access to the original binding and a large number of very fast accesses to the re-binding as function parameter.