CyberTiggyr submission to ICFP annual contest 2006

Gene Michael Stover

created Friday, 2006 July 21
updated Friday, 2006 July 21

Contents

Chapter 1

What is this?

Chapter 2

Implementation thoughts

Chapter 3

Team Info

```
From:  ICFP Contest <icfpcontest-organizers@lists.andrew.cmu.edu>
To: gene@acm.org
Cc:
Date: Friday, July 21, 2006 07:58 am
Subject: Registration Successful

You have been successfully registered for the 2006 ICFP Programming
Contest.

- Contest Team (icfpcontest-organizers@lists.andrew.cmu.edu)

Team name: cybertiggyr
Estimated size: 1 person
Contact e-mail: gene@acm.org

From:  Organizers of the 2006 ICFP Programming Contest <icfpcontest-organizers@lists.andrew.cmu.e
To: cybertiggyr <gene@acm.org>
Cc:
Date: Friday, July 21, 2006 08:39 am
Subject: ICFP Contest: team information

Greetings, cybertiggyr! The following pieces of data will be needed by
your team to participate in the competition.

This decryption key will be needed when you run the codex:
(\b.bb)(\v.vv)06AAaxamA0TcPWjfdV

This URL will allow you to track your progress and submit publications:
http://www.icfpcontest.org/team/2be280637c7e9c94bb83568f2a25e7c2/

Both the key and the URL are unique to your team. You should keep them
```

```
secret.

The 2006 ICFP Contest Organizers
```

Chapter 4

Understanding the input

Here are the first twenty platters[1] & my manually decoded interpretation of them as instructions. I did this to write some of my test programs; you can't tell if your program works unless you know at least some specific cases of correct inputs & outputs.

N	platter (hex)	decoded instruction
0	#x080000D0	conditional move, A = 3, B = 2, C = 0
1	#x300000C0	add, A = 3, B = 0, C = 0
2	#xD2000014	orthography, r = 1, literal = #x0000014
3	#xD400005B	orthography, r = 2, literal = #x000005B
4	#xD6000035	orthography, r = 3, literal = #x0000035
5	#xD000000D	orthography, r = 0, literal = #x000000D
6	#xC45DD8F0	load program, A = 3, B = 6, C = 0
7	#x0F354000	
8	#x5F0000D0	
9	#x300000C0	
10	#xC0767DB1	
11	#xC5084871	
12	#xC92E8971	
13	#xCC4F73F2	
14	#xC8A7C9B3	
15	#xC5117F33	
16	#xC84C31B3	
17	#x0A901E00	
18	#x000FB800	
19	#x04EC8200	

[1] A platter in the contest's programming problem is an unsigned 32-bit value.

Chapter 5

Files

1. The input scroll, `codex.umz`, second version. official copy. local copy.
2. The original input scroll, `codex.old.umz`. local copy.
3. My team's decryption key in a file so it's easy to feed to the program. local copy
4. icfp2006.lisp
5. loadall.lisp
6. The SANDmark correctness test, `sandmark.umz`. official copy. local copy.
7. Example SANDmark output file, `sandmark-output.txt`. official copy. local copy.
8. test.lisp

Chapter 6

Optimization

When I first got the program running correctly, it executed about ten "universal machine" instructions per second. This was about 100 times slower than I expect was possible.

Here's the expression I'm currently using to enable profiling in Lisp:

```
(sb-profile:profile
  um-conditional-move um-array-index um-array-amendment um-allocation
  um-abandonment um-load-program)

(spin)

(sb-profile:report)

(sb-profile:unprofile
  um-conditional-move um-array-index um-array-amendment um-allocation
  um-abandonment um-load-program)
```

Here are the results of the very first profile run.

```
* (sb-profile:report)

measuring PROFILE overhead..done

  seconds   |  consed    |  calls  |  sec/call  |  name
-----------------------------------------------------
    36.050 | 824,303,672 |      79 |   0.456323 | SET-UM-PLATTER
     0.117 |   188,320 |     783 |   0.000150 | SET-UM-REG
     0.009 |    28,392 |      51 |   0.000185 | UM-LOAD-PROGRAM
     0.000 |    85,960 |   2,002 |   0.000000 | DECODE
     0.000 |    65,512 |   1,919 |   0.000000 | COPY-UNIVERSAL-MACHINE
     0.000 |         0 |   6,674 |   0.000000 | UM-TAPE-N-PAIR
     0.000 |   115,304 |   2,170 |   0.000000 | UM-TAPE-N
     0.000 |   110,768 |   2,086 |   0.000000 | UM-PLATTER
```

```
     0.000 |    20,584 |   9,685 |   0.000000 | UM-REG
     0.000 |    24,576 |   2,002 |   0.000000 | FETCH
     0.000 |    16,384 |   1,001 |   0.000000 | TICK
     0.000 |         0 |      84 |   0.000000 | UM-ARRAY-INDEX
     0.000 |    16,384 |      79 |   0.000000 | UM-ARRAY-AMENDMENT
     0.000 | 2,337,792 |       6 |   0.000000 | UM-ALLOCATION
     0.000 |         0 |       5 |   0.000000 | UM-ABANDONMENT
-----------------------------------------------------
    36.176 | 827,313,648 |  28,626 |     | Total

estimated total profiling overhead: 0.10 seconds
overhead estimation parameters:
  5.9999996e-8s/call, 3.66e-6s total profiling,
  1.2200001e-6s internal profiling
*
```

This very first profile run reveals that $\frac{36.050}{36.176} = 99$ percent of the run-time is spent in SET-UM-PLATTER. That's because I used functional persistence in the program, but each tape is an array. So to write to a platter, I must duplicate the entire tape.[1]

There are two ways to make it faster:

1. Use a represenation for a tape which does not require duplication of an entire tape when I write to a platter.

2. Do not use functional persistence for tapes.

If I remove functional persistence for tapes, it pretty much reoves persistence for everything else. While that might have been a good decision earlier in the project, I'm not looking forward to all the code changes it would require now.

On the other hand, since SET-UM-PLATTER isn't called very often, & the main cost is all the memory is allocates, it might be worthwhile to use a tree. That would allow me to stick with functional persistence (which means fewer code changes), but it would allocate less memory, & hopefully reduce the cost.

Maybe one of those tree systems from Okasaki's supremely cool *Purely Functional Data Structures* [1] would fit the bill.

Hours later... I decided to make a global variable which selects between functional persistence & modifying the object in place. I only had to change the program in two (or was it three?) places. It took about 10 minutes. Then I profiled it again. Where the first profiler run required more than 36 seconds, this one required less than a second.

So one well-placed performance improvement changed the program from too slow to very probably fast enough. I love profilers.

[1] I rarely use debuggers, & I don't like them even when I must use them. I sure love profilers, though!

Appendix A

Other File Formats

- This document is available in multi-file HTML format at http://cybertiggyr.com/gene/icfp2006/.
- This document is available in Pointless Document Format (PDF) at http://cybertiggyr.com/gene/icfp2006/icfp2006.pdf.

Bibliography

[1] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1999. ISBN 0-521-66350-4.