Yhc: The York Haskell Compiler

By
Tom Shackell
What?

- Yhc is a rewrite of the back end of the nhc98 system.
- The back-end of the compiler is replaced.
- The runtime system is replaced.
- The instruction set is different.
- The Prelude is heavily modified.
Why?

• It was written to address some issues with the nhc98 back end.

• In particular: The high bit problem.

• Also as an experiment: Can we make nhc98 more portable?
The High Bit Problem
Graph Reduction

• Lazy functional languages are usually implemented using graph reduction.

• Haskell expressions are represented by graphs.

```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

• The expression 'sum [1,2]' might be represented by the graph:
Reduction

```
sum
:
1
:
2
[
]
Reduction

1

2

[ ]
Reduction

\[ \text{sum} : 1 \rightarrow 2 \rightarrow [ ] \rightarrow 3 \]
Reduction
Heap Node

We can see there are 4 types of graph node

Constructor

Thunk

Blackholed Thunk

Indirection

sum

In nhc and Yhc these graph nodes are represented with 4 types of heap node
Heap Nodes in nhc

- **Constructor**: Constructor Information
  - 10
- **Thunk**: Function Information Pointer
  - 0
- **Blackholed Thunk**: Function Information Pointer
  - 1
- **Indirection**: Redirection Pointer
  - 00

```
<table>
<thead>
<tr>
<th>Constructor</th>
<th>Constructor Information</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thunk</td>
<td>Function Information Pointer</td>
<td>1</td>
</tr>
<tr>
<td>Blackholed Thunk</td>
<td>Function Information Pointer</td>
<td>1</td>
</tr>
<tr>
<td>Indirection</td>
<td>Redirection Pointer</td>
<td>00</td>
</tr>
<tr>
<td>sum</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
The “High Bit” problem

- nhc assumes that it can use the topmost bit of a pointer to store information.
- This is not always the case: many modern Linux-x86 kernels allocate memory in addresses too high to fit in 31 bits.
Yhc makes sure that all FInfo structures are 4 byte aligned. Freeing up a bit at the bottom for Thunk nodes.

It also represents constructors by using a pointer to the information about the constructor, rather than encoding the information into the heap word.

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Information Pointer</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Constructor Information Pointer</td>
<td>01</td>
</tr>
<tr>
<td>Thunk</td>
<td>Function Information Pointer</td>
<td>01</td>
</tr>
<tr>
<td>Blackholed Thunk</td>
<td>Function Information Pointer</td>
<td>11</td>
</tr>
<tr>
<td>Indirection</td>
<td>Redirection Pointer</td>
<td>00</td>
</tr>
</tbody>
</table>
Instruction Sets

• The instruction set for Yhc is much simpler than for nhc.
• Both are based on stack machines.
• However, nhc has instructions for directly manipulating both the heap and the stack.
• Where as Yhc only directly manipulates the stack.
Instructions

main :: IO ()
main = putStrLn (show 42)

nhc instructions

main():
    HEAP_CVAL show
    HEAP_INT 42
    PUSH_HEAP
    HEAP_CVAL putStrLn
    HEAP_OFF -3
    RETURN_EVAL

Yhc instructions

main():
    PUSH_INT 42
    MK_AP show
    MK_AP putStrLn
    RETURN_EVAL
nhc instructions

main():
    HEAP_CVAL show
    HEAP_INT 42
    PUSH_HEAP
    HEAP_CVAL putStrLn
    HEAP_OFF -3
    RETURN_EVAL
nhc instructions

main():
  HEAP_CVAL show
  HEAP_INT 42
  PUSH_HEAP
  HEAP_CVAL putStrLn
  HEAP_OFF -3
  RETURN_EVAL
nhc instructions

main():
    HEAP_CVAL show
    HEAP_INT 42
    PUSH_HEAP
    HEAP_CVAL putStrLn
    HEAP_OFF -3
    RETURN_EVAL

Stack

Heap

show

42

Constants
main():
  HEAP_CVAL show
  HEAP_INT 42
  PUSH_HEAP
  HEAP_CVAL putStrLn
  HEAP_OFF -3
  RETURN_EVAL

Heap

Stack

Constants

42
main():
  HEAP_CVAL show
  HEAP_INT 42
  PUSH_HEAP
  HEAP_CVAL putStrLn
  HEAP_OFF -3
  RETURN_EVAL

Stack

Heap

Constants

42

d绚.error

show
pputStrLn
nhc instructions

main():
  HEAP_CVAL show
  HEAP_INT 42
  PUSH_HEAP
  HEAP_CVAL putStrLn
  HEAP_OFF -3
  RETURN_EVAL

Heap

Stack

Constants

42

show

putStrLn
main():
   HEAP_CVAL show
   HEAP_INT 42
   PUSH_HEAP
   HEAP_CVAL putStrLn
   HEAP_OFF -3
   RETURN_EVAL

Stack

Heap

Constants

42

show

putStrLn
Yhc instructions

main():
    PUSH_INT 42
    MK_AP show
    MK_AP putStrLn
    RETURN_EVAL
main():
  PUSH_INT 42
  MK_AP show
  MK_AP putStrLn
  RETURN_EVAL

Stack

Heap

42
main()::
  PUSH_INT 42
  MK_AP show
  MK_AP putStrLn
  RETURN_EVAL
main():
  PUSH_INT 42
  MK_AP show
  MK_AP putStrLn
  RETURN_EVAL
Yhc instructions

main():
  PUSH_INT 42
  MK_AP show
  MK_AP putStrLn
  RETURN_EVAL
Comparison

• Yhc uses less instructions to do the same thing.
• Because it doesn't have to have explicit movements between heap and stack.
• ... and because it can reference other nodes implicitly rather than using explicit heap offsets.
• Yhc instructions are also smaller
• Because it has more 'specializations'
• ... and again, because heap references are implicit
• These two factors make Yhc about 20% faster than nhc
Improving Portability
Bytecode in nhc

• nhc compiles Haskell functions into a bytecode for an abstract machine that manipulates graphs: The G-Machine.

• The bytecode is placed in a C source file, using an array of bytes. The C source file is then compiled and linked with the nhc interpreter to form an executable.

```
unsigned char[] FN_Prelude_46sum = {
    NEEDHEAP_I32, HEAP_CVAL_I3, HEAP_ARG, 1, HEAP_CVAL_I4,
    HEAP_ARG, 1, HEAP_CVAL_I5, HEAP_OFF_N1, 3, HEAP_CADR_N1, 1,
    PUSH_HEAP, HEAP_CVAL_P1, 6, HEAP_OFF_N1, 8, HEAP_OFF_N1, 5,
    RETURN, ENDCODE
};
```
Portable?

• The C code is portable, isn't it?
• Yes, but:
  • It creates a dependency on a C compiler.
  • There are issues with the nuances of various C compilers.
• The bytecode can't be loaded dynamically.
Improved Portability.

- Yhc also compiles Haskell functions into bytecode instructions for a G-Machine.
- However, Yhc places the bytecodes in a separate file which is then loaded by the interpreter at runtime. Similar to Java's classfile system.
- More portable, but it means Yhc has to do its own linking.
More Portable Still?

• Can we extend portability to include portability over a network?

• Then we could take a closure on one machine and have it run on another machine.

• Not implemented yet, but some interesting ideas.
Computer A

calc → data

Computer B
Computer A

Computer B

calc

data

Need calc
Computer A

Computer B

Need calc

calc → data
Computer A

Need calc

Computer B

calc

data
Computer A

Need calc

calc

calc(x):
    PUSH_ARG x
    PUSH_CONST subcalc
    MK_AP iter
    RETURN_EVAL

Computer B

calc → data
calc(x):
PUSH_ARG x
PUSH_CONST subcalc
MK_AP iter
RETURN_EVAL
calc(x):
PUSH_ARG x
PUSH_CONST subcalc
MK_AP iter
RETURN_EVAL
calc(x):
  PUSH_ARG x
  PUSH_CONST subcalc
  MK_AP iter
  RETURN_EVAL
calc(x):
  PUSH_ARG x
  PUSH_CONST subcalc
  MK_AP iter
  RETURN_EVAL
Computer A

Computer B

IND
data

iter

Need iter

subcalc
And so on ...
Computer A

Result

Computer B

42
Computer A

Result

42

Computer B
Computer A

Calc

Data

42

Result

Computer B
Computer A

IND

42

Result

Computer B
Challenges

• Needs concurrency to be useful.
• Complicates Garbage collection.
• Level of granularity versus laziness.
• Possible architecture differences.
Other Things!

• Other people have written various interpretters and backends for Yhc bytecode: Java, Python, .NET

• ... and various related tools such as interactive interpretters.

• I'm also using Yhc to do my Hat G-Machine work.
Questions?