The State of the GHCJS

Luite Stegeaman
Currently...
Works out of the box with GHC 7.10.2!

After Hackage release (soon...):

$ cabal install ghcjs
$ ghcjs-boot

Before it’s been actually uploaded, use a snapshot:

$ cabal install http://ghcjs.luite.com/improved-base.tar.gz
$ ghcjs-boot
Where we are

Since last ICFP:

- Cabal support merged (version 1.22)
- Stack support
- Time profiling (node.js only)
- Base library rewrite (improved-base)
- GHCJSi REPL (experimental)
- Major performance improvements (linker, Template Haskell)
- FFI more flexible: More types allowed, return (unboxed) tuples
- Many bugs fixed!
**IMPROVED-BASE**

- Rewrite of the `ghcjs-base` package
- *Batteries included* for standard JavaScript:
  - `Data.JSString` library with full `Data.Text` API and stream fusion
  - `JavaScript.Array`
  - `JavaScript.TypedArray`
  - `JavaScript.Number`
  - `JavaScript.Object`

- Standard web API’s:
  - `JavaScript.Web.Canvas` (used to be in `ghcjs-canvas`)
  - `JavaScript.Web.Storage`
  - `JavaScript.Web.WebSocket`
  - `JavaScript.Web/XMLHttpRequest`

- Todo:
  - JSON support unfinished (ideally: integration with `aeson`
PROFILING

Based on Cost Centre Stacks, like GHC

- Heap Profiling
  - Last year GSoC
  - GUI still incomplete

- Time Profiling (node.js)
  - Uses the built-in statistical profiler
  - record Cost Centre Stacks in samples
  - Requires installation of support library with npm
  - see /utils/ghcjs-node-profiling
<table>
<thead>
<tr>
<th>Percentage</th>
<th>Time</th>
<th>Function/Module</th>
<th>File/Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>98.8%</td>
<td>00.0%</td>
<td>LazyCompile: ~Module._extensions..js</td>
<td>module.js:476:37</td>
</tr>
<tr>
<td>98.8%</td>
<td>00.0%</td>
<td>LazyCompile: ~Module._compile</td>
<td>module.js:378:37</td>
</tr>
<tr>
<td>98.5%</td>
<td>00.0%</td>
<td>Function: ~&lt;anonymous&gt;</td>
<td>/home/luite/ghcjs/testc.js:1:11</td>
</tr>
<tr>
<td>98.8%</td>
<td>00.0%</td>
<td>LazyCompile: ~h$cpuProfiler.runCC</td>
<td>/home/luite/ghcjs/testc.js:27:17</td>
</tr>
<tr>
<td>59.1%</td>
<td>00.0%</td>
<td>CostCentre: cost centre A</td>
<td>main.hs:10:10</td>
</tr>
<tr>
<td>39.7%</td>
<td>00.0%</td>
<td>CostCentre: cost centre B</td>
<td>main.hs:14:3</td>
</tr>
<tr>
<td>39.7%</td>
<td>39.6%</td>
<td>CostCentre: cost centre C</td>
<td>main.hs:21:9</td>
</tr>
<tr>
<td>19.4%</td>
<td>19.4%</td>
<td>CostCentre: cost centre C</td>
<td>main.hs:21:9</td>
</tr>
<tr>
<td>39.4%</td>
<td>00.0%</td>
<td>CostCentre: cost centre B</td>
<td>main.hs:14:3</td>
</tr>
<tr>
<td>20.0%</td>
<td>19.8%</td>
<td>CostCentre: cost centre A</td>
<td>main.hs:10:10</td>
</tr>
<tr>
<td>00.2%</td>
<td>00.1%</td>
<td>LazyCompile: *pow native</td>
<td>math.js:89:17</td>
</tr>
<tr>
<td>19.4%</td>
<td>19.4%</td>
<td>CostCentre: cost centre C</td>
<td>main.hs:21:9</td>
</tr>
</tbody>
</table>
GHCJSi

- Finally, a REPL! (experimental!)
- See ghcjsi branch on Github
- Works like GHCi with full DOM access and JavaScript FFI
- Code runs on node.js until a browser connects
- Uses incremental linking:
  1. compile expression
  2. collect JS code for dependencies not yet loaded
  3. send code to JS engine and run

Limitation: *Stepping and tracing not yet supported*

![GHCJSi components](image)
Currently...

Figure 2: GHCJSi
Currently... So...
Currently...

Soon...
Figure 3: West Coast Trail
SHORT-TERM GOALS

- Code size and performance improvements
  - Better optimizer
  - ES2015 support (tail calls)

- Better development tools
  - Source maps
  - Wider support for profiling
  - Assertions
  - make GHCJSi more robust
Limitations of the Gen2 code generator

- Optimizer is slow and complicated
  - Adding rules tricky: rewrite untyped JavaScript (JMacro)
  - Generated code is hard to debug

- No flexibility in output naming, h$ prefixes everywhere

- Impossible to trade features for code size or speed (drop threading for example)

- Cannot make use of new ES2015 features (tail calls!) since they’re not supported everywhere

Reason: JavaScript AST and data stored in the js_o object files too close to the final JavaScript

Solution: Change AST, but a large performance hit when linking is unacceptable!
HEAPS OF THUNKS

- Haskell heap object

  \{ f: \text{function}, m: \text{meta}, d1: x, d2: y \} 

- Some data values represented directly as a \textit{Number}, can be distinguished from thunks using JavaScript’s \texttt{typeof} operator:
  - \texttt{Bool}
  - \texttt{Int}
  - \texttt{Char}
  - \texttt{Double}
  - \texttt{Word16}
  - \texttt{enumerations}
Threads

```javascript
function mainloop() {
    var thread, f;
    while((thread = scheduler()) !== null) {
        f = thread.nextCall;
        while(f !== stop && !endOfQuantum()) {
            f = f();
        }
    }
}
```

Forces us to use global variables for arguments
**FORCING A THUNK**

\[
\begin{align*}
\text{function } f(x, p) &\{ \\
&\quad \text{var } _x = \text{reduce}(x); \\
&\quad \text{if}(\text{constrTag}(\_x) === 1) \{ \\
&\quad\quad \text{return false;} \\
&\quad\} \text{ else } \{ \\
&\quad\quad \text{return } \text{apply1}(p, \_x.d1); \\
&\quad\} \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{function } f() &\{ \\
&\quad \text{var } x = \text{arg1}; \\
&\quad \text{var } p = \text{arg2}; \\
&\quad \text{push}(p, \text{f1}); \\
&\quad \text{return } \text{reduce}(x); \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{function } f1() &\{ \\
&\quad \text{var } _x = \text{arg1}; \\
&\quad \text{pop();} \quad // \text{pop } f1 \\
&\quad \text{var } p = \text{pop()}; \\
&\quad \text{if}(\text{constrTag}(\_x) === 1) \{ \\
&\quad\quad \text{arg1 } = \text{false}; \\
&\quad\quad \text{return } \text{stack}[\text{sp}]; \\
&\quad\} \text{ else } \{ \\
&\quad\quad \text{return } \text{apply1}(p, \_x.d1); \\
&\quad\} \\
\}
\end{align*}
\]
ES2015 TAIL CALLS

```javascript
function f() {
  var x = arg1;
  var p = arg2;
  push(p, f1);
  return reduce(x);
}

function f1() {
  var _x = arg1;
  pop(); // pop f1
  var p = pop();
  if(constrTag(_x) === 1) {
    arg1 = false;
    return stack[sp];
  } else {
    return apply1(p, _x.d1);
  }
}

ES2015:

function f(x, p) {
  push(p, f1);
  return reduce(x);
}

function f1(_x) {
  pop(); // pop f1
  var p = pop();
  if(constrTag(_x) === 1) {
    return stack[sp](false);
  } else {
    return apply1(p, _x.d1);
  }
}
```
**New Code Generator (Tyr)**

- Replaces JMacro based current generator (Gen2)
- Own AST, no more quasiquoter
- JavaScript with some extensions:
  - Source location annotations
  - Haskell calls
  - Heap object construction / matching
  - Tuples
- Two Phase (delay CPS transformation)
  1. Non-preemptive threads
  2. Preemptive threads (after CPS)
- Simple type system for optimizer, AST linter and runtime assertions
  - `int`, `number`, `heap object`, `unknown`
- Flexible (re)namining of Haskell symbols
  - Keep track of origin of all generated names
  - Get rid of fixed `h$` prefixes
  - Module system support?
CONCLUSION

- Integration with build tools is complete
- *improved*-base library is a major step forward in usability
- REPL and profiling support in progress
- Further improvements and ES2015 require some internal changes, addressed by *Tyr*

Other work to do:

- More comprehensive continuous integration testing (including performance)
- Automated DOM testing