The State of the GHCJS

Luite Stegeman

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`

Time Profiling report

```
98.8% 00.0% LazyCompile: ~Module._extensions..js module.js:
98.8% 00.0%  LazyCompile: ~Module._compile module.js:378:3
98.8% 00.0%   Function: ~<anonymous> /home/luite/ghcjs/tes
98.5% 00.0%    LazyCompile: ~h$cpuProfiler.runCC /home/lu
59.1% 00.0%     CostCentre: cost centre A main.hs:10:10
39.7% 00.0%      CostCentre: cost centre B main.hs:14:3
39.7% 39.6%       CostCentre: cost centre C main.hs:21:9
19.4% 19.4%      CostCentre: cost centre C main.hs:21:9
39.4% 00.0%     CostCentre: cost centre B main.hs:14:3
20.0% 19.8%      CostCentre: cost centre A main.hs:10:10
00.2% 00.1%       LazyCompile: *pow native math.js:89:17
19.4% 19.4%      CostCentre: cost centre C main.hs:21:9
```

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*

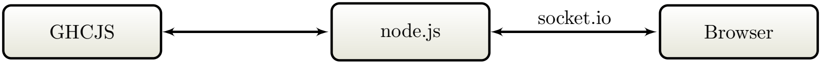

Figure 1: GHCJSi components

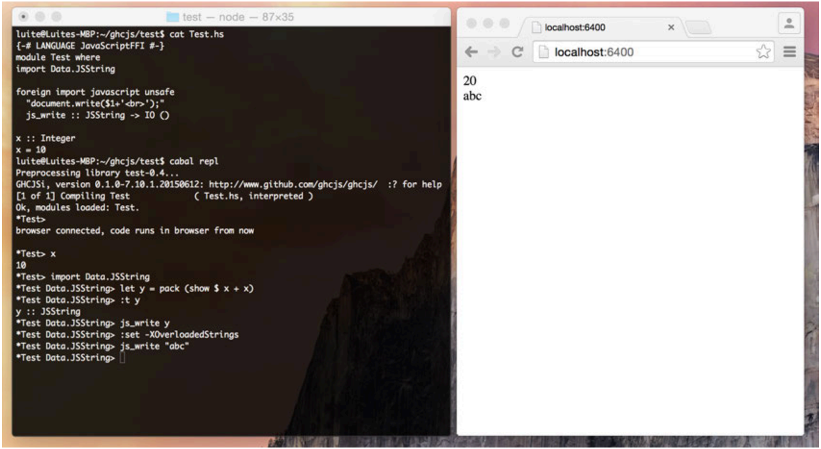

Figure 2: GHCJSi

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: function, m: meta, d1: x, d2: y }
```

- Some data values represented directly as a *Number*, can be distinguished from thunks using JavaScript's `typeof` operator:
  - `Bool`
  - `Int`
  - `Char`
  - `Double`
  - `Word16`
  - enumerations

Threads

```
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

```
f :: Maybe a -> (a -> Bool)
  -> Bool
f x p = case x of
  Nothing -> False
  Just y  -> p y
```

```
function f(x, p) {
  var _x = reduce(x);
  if(constrTag(_x) === 1) {
    return false;
  } else {
    return apply1(p, _x.d1);
  }
}
```

```
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 TAIL CALLS

```
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)naming 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