Blank Canvas and the remote-monad design pattern

A Foreign Function Interface to the JavaScript Canvas API

Extended Abstract

Andrew Gill Aleksander Eskilson Ryan Scott James Stanton

Information and Telecommunication Technology Center
The University of Kansas
{andygill,aeskilson,ryanscott,jstanton}@ittc.ku.edu

Abstract

JavaScript is the de-facto assembly language of the internet. Browsers offer an array of powerful rendering and event processing services, including a simple 2D canvas. Blank Canvas is Haskell DSL that provides a Foreign Function Interface to the JavaScript canvas API and the JavaScript event API. With this capability, Haskell programmers can draw pictures on the browsers, and access input from the keyboard and mouse. At the University of Kansas, we use the blank-canvas package for teaching Haskell, where it provides a more interesting I/O experience than stdio.

We investigate the use of the remote-monad design pattern, using Blank Canvas as our driving example. After explaining the design pattern, and constructing the basic remote capability, we critically assess the feasibility of our straightforward approach, and explore improvements.

1. Introduction

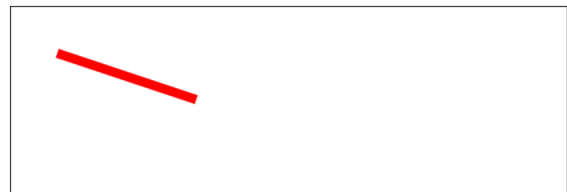
Blank Canvas is a Haskell binding to the complete HTML5 Canvas API. Blank Canvas allows Haskell users to write, in Haskell, interactive images onto their web browsers. Blank Canvas gives the user a single full-window canvas, and provides many well-documented functions for rendering images.

As a first example and in order to give a feel for the library, consider drawing a single red line onto the canvas. In Haskell, using Blank Canvas we can write the following.

```
send context $ do                      -- ❶
        moveTo(50,50)                  -- ❷
        lineTo(200,100)
        lineWidth 10
        strokeStyle "red"
        stroke()                       -- ❸
```

Copyright held by author(s). This is an unrefereed extended abstract, distributed for the purpose of feedback toward submitting a complete paper on the same topic to IFL'14.

First, the `send` command (❶) sends a monadic list of commands to a (graphics) context. Second, the list of commands (❷) operates on this context in an imperative manner. Finally, the `stroke()` commands (❸) actually draws the red line. At this point, the screen looks like



In JavaScript, the same actions can be performed using an almost identical code fragment.

```
-- JavaScript
context.moveTo(50,50);
context.lineTo(200,100);
context.lineWidth = 10;
context.strokeStyle = "red";
context.stroke();
```

Blank Canvas has packaged the JavaScript API as a small Domain Specific Language in Haskell, and allows Haskell users to access the canvas. At the University of Kansas, we make extensive use of this API. Students find it easy to understand, and complete medium-sized projects, usually games, using Blank Canvas as the primary IO mechanism. In the graduate FP class, we also undertake an FRP exercise [1, 8], which uses Blank Canvas to render shapes onto the canvas. Using the Blank Canvas API, we also have developed slide presentation software, and an internal animation framework. Finally, the popular `diagrams` package [11, 12] has been ported to use blank Canvas as a back end [7].

The central issue, and the subject of the full paper, is quantifying the costs associated with having code execute outside the Haskell runtime system, and remotely running monadic code. The browser, running JavaScript, is typically a separately executing process from a Haskell program. Thus, we have two extreme solutions to our API implementation, sending each command over a network connection piecemeal, or compiling the entire Haskell program and runtime system into JavaScript. We investigate a middle ground between sending commands piecemeal, and compiling wholesale to JavaScript, using a design pattern.

2. Remote-monad DSL Pattern

Haskell has no standard graphics library. Instead, a rich Foreign Function Interface (FFI) capability is used to tunnel to C, and onwards to established libraries, such as OpenGL. There are three conceptual problems to be solved in crossing to non-native C (and C++) libraries, such as OpenGL:

- First, control flow needs to flow to the correct C function. Given the lowest level of the GHC runtime system is written in C, this is straightforward. Callbacks, from C to Haskell can also be arranged.
- Second, the data structures that are arguments and results of calls to (and from) C need to be coerced into the correct format. C strings are not the same as Haskell strings.
- Third, the abstractions of OpenGL may not be idiomatic Haskell abstractions. For example, many APIs assume OO-style class inheritance. This can be simulated in Haskell, but raises an obfuscation barrier.

Any time control flow leaves the eco-system, all three of these concerns come to into play. All three are well handled in the Haskell FFI for C. There is a way of directly promoting a C function into Haskell-land, there is a good support for marshalling data structures, in C structures, as well an automatic memory management support, and Haskell abstraction capabilities are used to build more Haskell-centric APIs on top of the FFI capability. Calling C functions directly from Haskell is cheap. However, we want to investigate another FFI with a different tradeoff, where the call is remote and expensive, and understand what abstractions can be used.

The **remote-monad pattern** is our name for the transmission of a (fixed) set of commands to a remote site, for execution. In its most basic form, we have a `send` command, a remote location identifier, and a single command.

```
send :: Name -> RemoteCommand a -> IO a
remote :: Name
readRemoteFile :: String -> RemoteCommand String
example :: IO ()
example = do
     txt <- send remote (readFile "foo")
     print txt
```

The idea is that the `send` command reifies the `RemoteCommand`, sends it to the remote location, runs it, accepts the response, transports it back to the original `send`, and returns the remotely generated value. This pattern can implemented using the first two of the three requirements above of a an FFI interface, as described above. First, a remote function is called (control is moved to the remote site), and back. Second, the pattern takes care of the necessary data conversation conventions, both in transport, and on the remote site.

The remote-monad command has many manifestations. At KU, we have used it for Blank Canvas, but also for Sunroof (sending whole JavaScript programs), and using Kansas Lava [2] to talk to remote peripherals. Furthermore, The pattern appears in many different places. If we interpret "remote" to mean different environment, the run function for many well know monads can be considered a `send`. Software transactional memories (`atomically` [3]), the ST monad (`runST` [5]) and IO (`forkIO` [4]) can also ben considered close instances of the remote-monad pattern, where a monad is executed in a different context.

The remote-monad patterns has two laws:

```
send (return a) = return a              (1)
send m1 >>= send m2 = send (m1 >>= m2) (2) [*]
```

The first law states that a `send` has no effect except the remote commands. The second law, which has a pre-condition of non-interference[*], states that remote commands preserve ordering, and can be split and joined into different sized packets. The pre-condition is interesting: it is possible to have the result of two `send`'s be the same as a single `send`, yet the observable effects be different, for example a screen update is done between the two send commands.

3. Blank Canvas

Blank Canvas is a small library at around 1500 lines of Haskell. At the heart of the library is the remote-monad, and the `send` command. There is quite a bit of careful construction, however, to make everything work.

The packet principles are:

- Where possible, everything in a `send`-packet should be sent to be executed together.
- The breaks between packets should be deterministic and statically/syntactically determinable.
- Packets are not combined between different calls to `send`.

The command principles are:

- Anything that returns `()` is asynchronous, and may be combined with the next monadic command, or send instantly.
- Anything that does not return `()` is synchronous, and requires a round-trip to the server.

The `Canvas` data type has a small number of constructors. The four main constructor are:

```
data Canvas :: * -> * where
  Method :: Method                          -> Canvas ()
  Query  :: Query a                         -> Canvas a
  Bind   :: Canvas a -> (a -> Canvas b) -> Canvas b
  Return :: a                               -> Canvas a
```

The choice of constructors follow the principles carefully. `Method` is used for asynchronous drawing commands, while `Query` is used for commands that need a round trip. `Bind` and `Return` form the monad for Canvas, allowing monadic reification [9, 10].

4. Benchmarking Blank Canvas

A key question is the cost of using the remote-monad design pattern. At first glance, it would seem prohibitive. The current version of Blank Canvas (0.5) uses Haskell Strings internally, transliterating each command to a String, and combines intra-`send` commands, where possible. Absolutely every command needs translated then sent over a (typically local) network.

We have measured Blank Canvas on a small number of benchmarks, and compared to native JavaScript. We have two classes of benchmarks: "display" benchmarks, that simply render to the HTML5 canvas, and "query" benchmarks, that the inner loop of the benchmark invokes some from of query that requires a round-trip from server, to client, back to the server.

Blank Canvas works on almost any modern, HTML5 compliant browser. Figure 1 gives our initial results. We have tested each benchmark on recent versions of Firefox and Chrome, on both Linux and OSX, to gain a crude overall benchmark for how much using the remote-monad design pattern costs. The Haskell tests were run 100 times using criterion [6], the JavaScript tests were averaged over 100 runs.

	Benchmark	Linux Firefox Haskell	Linux Firefox JS	Linux Firefox Ratio	Linux Chrome Haskell	Linux Chrome JS	Linux Chrome Ratio	OSX Firefox Haskell	OSX Firefox JS	OSX Firefox Ratio	OSX Chrome Haskell	OSX Chrome JS	OSX Chrome Ratio
Display	Bezier	6.90	4.09	1.69	4.03	1.71	2.36	11.56	3.23	3.58	8.51	0.55	15.47
	CirclesRandSz	138.64	105.45	1.31	71.15	25.07	2.84	68.77	46.26	1.49	66.97	12.84	5.22
	CirclesUniSz	106.90	75.41	1.42	62.43	15.28	4.09	71.19	31.32	2.27	67.52	12.54	5.38
	FillText	57.95	48.33	1.20	4.99	1.80	2.77	7.81	5.22	1.50	5.07	1.29	3.93
	StaticAsteroids	365.10	121.71	3.00	309.59	14.92	20.75	197.92	30.49	6.49	201.21	8.07	24.93
	Image	214.63	21.87	9.81	421.41	57.41	7.34	596.29	209.74	2.84	657.68	75.82	8.67
Query	IsPointInPath	22.31	0.49	45.53	27.73	0.26	106.65	33.72	0.73	46.19	74.71	0.37	201.91
	MeasureText	184.18	50.56	3.64	160.76	2.04	78.80	265.22	5.92	44.80	320.49	1.40	228.92
	Rave	58.30	20.50	2.84	38.66	1.71	22.61	62.18	10.98	5.66	115.43	0.58	199.02

Table 1. Benchmarking Blank Canvas vs. Native JavaScript. (times in milliseconds)

The display benchmarks are:

- Bezier – drawing 1000 bezier curves.
- CirclesRandomSize – 1000 filled in circles of random sizes.
- CirclesUniformSize – 1000 filled in circles of a uniform size.
- FillText – 50 words
- StaticAsteroids – 1000 wire polygons.
- Image – 100 images of a cat, drawn at different sizes.

What can be seen is that the relative performance varies widely, depending on browser and benchmark, but on average, the cost of using Haskell, and the Blank Canvas API is between approximately 2 and 25, and typically less than 5. This is surprising and encouraging! We were expecting a larger overhead. We can also see the importance of a testing with different environments.

The query benchmarks are:

- IsPointInPath – Draw 1000 rectangles and and points; the points' color depends on if the point is inside the rectangle.
- MeasureText – measure the width of 100 words.
- Rave – gradient bars.

Here, as expecting, the cost is much higher. However, again, the result is encouraging. The places where the overhead is especially high are where a specific browser does an especially good job of optimization. Further, as has been pointed out by Jeffrey Rosenbluth, a number of our queries simply allocate a numbered resource, and this unique number generation can be done on the server, allowing a command rather than query to be used.

5. Related Work

The final paper will contain a detailed related work section, including various JavaScript-based Haskell compilers, and other approaches to the FFI problem.

6. Conclusion

This short extended abstract has introduced the remote-monad design pattern, and shown its use in a full scale case-study for accessing the HTML5 Canvas JavaScript API. The cost was not prohibitive, and the API is useful in practice.

Acknowledgments

We would like the thank Jeffrey Rosenbluth, for writing `diagrams-canvas`, and helping with the implementation of Blank Canvas, and Justin Dawson, for working on an earlier version of the asteroid benchmark. This material is based upon work supported by the National Science Foundation under Grant No. 1117569 and Grant No. 1350901.

References

[1] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997. URL `http://conal.net/papers/icfp97/`.

[2] A. Gill, T. Bull, A. Farmer, G. Kimmell, and E. Komp. Types and associated type families for hardware simulation and synthesis: The internals and externals of Kansas Lava. *Higher-Order and Symbolic Computation*, pages 1–20, 2013. ISSN 1388-3690. . URL `http://dx.doi.org/10.1007/s10990-013-9098-7`.

[3] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

[4] S. L. P. Jones, A. D. Gordon, and S. Finne. Concurrent haskell. In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, St. Petersburg Beach, Florida, USA, January 21-24, 1996*, pages 295–308, 1996.

[5] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. *ACM SIGPLAN Notices*, 29(6):24–35, 1994.

[6] B. O'Sullivan. `http://hackage.haskell.org/package/criterion`, 2014.

[7] J. Rosenbluth. `http://hackage.haskell.org/package/diagrams-canvas`, 2014.

[8] N. Sculthorpe and A. Gill. `http://hackage.haskell.org/package/yampa-canvas`, 2014.

[9] N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *In Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 287–298. ACM, 2013. URL `http://dl.acm.org/citation.cfm?doid=2500365.2500602`.

[10] J. Svenningsson and B. J. Svensson. Simple and compositional reification of monadic embedded languages. In *International Conference on Functional Programming*, pages 299–304. ACM, 2013.

[11] B. Yorgey. `http://hackage.haskell.org/package/diagrams`, 2014.

[12] B. A. Yorgey. Monoids: theme and variations (functional pearl). In *Haskell Syposium*. ACM, 2012.