A Seamless, Client-Centric Programming Model for Type Safe Web Applications

Anton Ekblad and Koen Claessen
Chalmers University of Technology
{antonek,koen}@chalmers.se

Abstract

We propose a new programming model for web applications which is (1) seamless; one program and one language is used to produce code for both client and server, (2) client-centric; the programmer takes the viewpoint of the client that runs code on the server rather than the other way around, (3) functional and type-safe, and (4) portable; everything is implemented as a Haskell library that implicitly takes care of all networking code. Our aim is to improve the painful and error-prone experience of today's standard development methods, in which clients and servers are coded in different languages and communicate with each other using ad-hoc protocols. We present the design of our library called Haste.App, an example web application that uses it, and discuss the implementation and the compiler technology on which it depends.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Distributed Programming; D.3.2 [*Language Classifications*]: Applicative (functional) languages; H.3.5 [*Online Information Services*]: Web-based services

Keywords web applications; distributed systems; network communication

1. Introduction

Development of web applications is no task for the faint of heart. The conventional method involves splitting your program into two logical parts, writing the one in JavaScript, which is notorious even among its proponents for being wonky and error-prone, and the other in any compiled or server-interpreted language. Then, the two are glued together using whichever home-grown network protocol seems to fit the application. However, most web applications are conceptually single entities, making this forced split an undesirable hindrance which introduces new possibilities for defects, adds development overhead and prevents code reuse.

Several solutions to this problem have been proposed, as discussed in section 5.1, but the perfect one has yet to be found. In this paper,

Haskell '14, September 4–5, 2014, Gothenburg, Sweden.

ACM 978-1-4503-3041-1/14/09…$15.00.
http://dx.doi.org/10.1145/2633357.2633367

we propose a functional programming model in which a web application is written as a single program from which client and server executables are generated during compilation. Type annotations in the source program control which parts are executed on the server and which are executed on the client, and the two communicate using type safe RPC calls. Functions which are not explicitly declared as server side or client side are usable by either side.

Recent advances in compiler technology from functional languages to JavaScript have led to a wealth of compilers targeting the web space, and have enabled the practical development of functional libraries and applications for the browser. This enables us to implement our solution as a simple Haskell library for any compiler capable of producing JavaScript output, requiring no further modification to existing compilers.

As our implementation targets the Haste Haskell to JavaScript compiler [11], this paper also goes into some detail about its design and implementation as well as the alternatives available for compiling functional languages to a browser environment.

Motivation Code written in JavaScript, the only widely supported language for client side web applications, is often confusing and error-prone, much due to the language's lack of modularity, encapsulation facilities and type safety.

Worse, most web applications, being intended to facilitate commu-

nication, data storage and other tasks involving some centralized resource, also require a significant server component. This component is usually implemented as a completely separate program, and communicates with the client code over some network protocol.

This state of things is not a conscious design choice - most web applications are conceptually a single entity, not two programs which just happen to talk to each other over a network - but a consequence of there being a large, distributed network between the client and server parts. However, such implementation details should not be allowed to dictate the way we structure and reason about our applications - clearly, an abstraction is called for.

For a more concrete example, let's say that we want to implement a simple "chatbox" component for a website, to allow visitors to discuss the site's content in real time. Using mainstream development practices and recent technologies such as WebSockets [15], we may come up with something like the program in figure 1 for our client program. In addition, a corresponding server program would need to be written to handle distribution of messages among clients. We will not give such an implementation here, as we do not believe it necessary to state the problem at hand.

Since the "chatbox" application is very simple - users should only be able to send and receive text messages in real time - we opt for a very simple design. Two UI elements, `logbox` and `msgbox`, represent the chat log and the text area where the user inputs their

---

```
function handshake(sock) {sock.send('helo');}
function chat(sock, msg) {sock.send('text' + msg);}
```

```
window.onload = function() {
  var logbox = document.getElementById('log');
  var msgbox = document.getElementById('message');
  var sock = new WebSocket('ws://example.com');

  sock.onmessage = function(e) {
    logbox.value = e.data + LINE + logbox.value;
  };

  sock.onopen = function(e) {
    handshake(sock);
    msgbox.addEventListener('keydown', function(e) {
      if(e.keyCode == 13) {
        var msg = msgbox.value;
        msgbox.value = '';
        chat(sock, msg);
      }
    });
  };
};
```

Figure 1: JavaScript chatbox implementation

messages respectively. When a message arrives, it is prepended to the chat log, making the most recent message appear at the top of the log window, and when the user hits the return key in the input text box the message contained therein is sent and the input text box is cleared.

Messages are transmitted as strings, with the initial four characters indicating the type of the message and the rest being the optional payload. There are only two messages: a handshake indicating that a user wants to join the conversation, and a broadcast message which sends a line of text to all connected users via the server. The only messages received from the server are new chat messages, delivered as simple strings.

This code looks solid enough by web standards, but even this simple piece of code contains no less than three asynchronous callbacks, two of which both read and modify the application's global state. This makes the program flow non-obvious, and introduces unnecessary risk and complexity through the haphazard state modifications.

Moreover, this code is not very extensible. If this simple application is to be enhanced with new features down the road, the network protocol will clearly need to be redesigned. However, if we were developing this application for a client, said client would likely not want to pay the added cost for the design and implementation of features she did not - and perhaps never will - ask for.

Should the protocol need updating in the future, how much time will we need to spend on ensuring that the protocol is used properly across our entire program, and how much extra work will it take to keep the client and server in sync? How much code will need to be written twice, once for the client and once for the server, due to the unfortunate fact that the two parts are implemented as separate programs, possibly in separate languages?

Above all, is it really necessary for such a simple program to involve client/server architectures and network protocol design at all?

2. A seamless programming model

There are many conceivable improvements to the mainstream web development model described in the previous section. We propose an alternative programming model based on Haskell, in which web applications are written as a single program rather than as two independent parts that just so happen to talk to each other.

Our proposed model, dubbed "Haste.App", has the following properties:

- The programming model is synchronous, giving the programmer a simple, linear view of the program flow, eliminating the need to program with callbacks and continuations.
- Side-effecting code is explicitly designated to run on either the client or the server using the type system while pure code can be shared by both. Additionally, general IO computations may be lifted into both client and server code, allowing for safe IO code reuse within the confines of the client or server designated functions.
- Client-server network communication is handled through statically typed RPC function calls, extending the reach of Haskell's type checker over the network and giving the programmer advance warning when she uses network services incorrectly or

forgets to update communication code as the application's internal protocol changes.

- Our model takes the view that the client side is the main driver when developing web applications and accordingly assigns the server the role of a computational and/or storage resource, tasked with servicing client requests rather than driving the program. While it is entirely possible to implement a server-to-client communication channel on top of our model, we believe that choosing one side of the heterogenous client-server relation as the master helps keeping the program flow linear and predictable.
- The implementation is built as a library on top of the GHC and Haste Haskell compilers, requiring little to no specialized compiler support. Programs are compiled twice; once with Haste and once with GHC, to produce the final client and server side code respectively.

2.1 A first example

While explaining the properties of our solution is all well and good, nothing compares to a good old Hello World example to convey the idea. We begin by implementing a function which prints a greeting to the server's console.

```
import Haste.App

helloServer :: String → Server ()
helloServer name =
```

```
  liftIO $ putStrLn (name ++ " says hello!")
```

Computations exclusive to the server side live in the `Server` monad. This is basically an IO monad, as can be seen from the regular `putStrLn IO` computation being lifted into it, with a few extra operations for session handling; its main purpose is to prevent the programmer from accidentally attempting to perform client-exclusive operations, such as popping up a browser dialog box, on the server.

Next, we need to make the `helloServer` function available as an RPC function and call it from the client.

```
main :: App Done
main = do
  greetings ← remote helloServer

  runClient $ do
    name ← prompt "Hi there, what is your name?"
    onServer (greetings <.> name)
```

The `main` function is, as usual, the entry point of our application. In contrast to traditional applications which live either on the client or on the server and begin in the `IO` monad, Haste.App applications live on both and begin execution in the `App` monad which provides some crucial tools to facilitate typed communication between the two.

The `remote` function takes an arbitrary function, provided that all its arguments as well as its return value are serializable through the `Serialize` type class, and produces a typed identifier which may

be used to refer to the remote function. In this example, the type of `greetings` is `Remote (String → Server ())`, indicating that the identifier refers to a remote function with a single `String` argument and no return value. Remote functions all live in the `Server` monad. The part of the program contained within the `App` monad is executed on both the server and the client, albeit with slightly different side effects, as described in section 3.

After the `remote` call, we enter the domain of client-exclusive code with the application of `runClient`. This function executes computations in the `Client` monad which is essentially an IO monad with cooperative multitasking added on top, to mitigate the fact that JavaScript has no native concurrency support. `runClient` does not return, and is the only function with a return type of `App Done`, which ensures that each `App` computation contains exactly one client computation.

In order to make an RPC call using an identifier obtained from `remote`, we must supply it with an argument. This is done using the `<.>` operator. It might be interesting to note that its type, `Serialize a ⇒ Remote (a → b) → a → Remote b`, is very similar to the type of the `<*>` operator over applicative functors. This is not a coincidence; `<.>` performs the same role for the `Remote` type as `<*>` performs for applicative functors. The reason for using a separate operator for this instead of making `Remote` an instance of `Applicative` is that since functions embedded in the `Remote` type exist only to be called over a network, such functions must only be applied to arguments which can be serialized and sent over a network connection. When a `Remote` function is applied to an argument using `<.>`, the argument is serialized and stored inside the resulting `Remote` object, awaiting dispatch. `Remote` computations

can thus be seen as explicit representations of closures.

After applying the value obtained from the user to the remote function, we apply the `onServer` function to the result, which dispatches the RPC call to the server. `onServer` will then block until the RPC call returns.

To run this example, an address and a port must be provided so that the client knows which server to contact. There are several ways of doing this: using the GHC plugin system, through Template Haskell or by slightly altering how program entry points are treated in a compiler or wrapper script, to name a few. A non-intrusive method when using the GHC/Haste compiler pair would be to add `-main-is setup` to both compilers' command line and add the `setup` function to the source code.


```
main = do
  remoteref ← liftServerIO $ newIORef 0

  count ← remote $ do
    r ← remoteref
    liftIO $ atomicModifyIORef r (λv → (v+1, v+1))

  runClient $ do
    visitors ← onServer count
```

```
    alert ("Your are visitor #" ++ show visitors)
```

Figure 2: server side state: doing it properly

```
setup :: IO ()
setup =
  runApp (mkConfig "ws://localhost:1111" 1111) main
```

This will instruct the server binary to listen on the port 1111 when started, and the client to attempt contact with that port on the local machine. The exact mechanism chosen to provide the host and port are implementation specific, and will in the interest of brevity not be discussed further.

2.2 Using server side state

While the Hello Server example illustrates how client-server communication is handled, most web applications need to keep some server side state as well. How can we create state holding elements for the server which are not accessible to the client?

To accomplish this, we need to introduce a way to lift arbitrary IO computations, but ensure that said computations are executed on the server and nowhere else. This is accomplished using a more restricted version of `liftIO`:

```
liftServerIO :: IO a → App (Server a)
```

`liftServerIO` performs its argument computation once on the server, in the `App` monad, and then returns the result of said computation inside the `Server` monad so that it is only reachable by server side code. Any client side code is thus free to completely ignore executing computations lifted using `liftServerIO`; since the result of a server lifted computation is never observable on the client, the client has no obligation to even produce such a value. Figure 2 shows how to make proper use of server side state.

2.3 The chatbox, revisited

Now that we have seen how to implement both network communication, we are ready to revisit the chatbox program from section 1, this time using our improved programming model. Since we are now writing the entire application, both client and server, as opposed to the client part from our motivating example, our program has three new responsibilities.

- We need to add connecting users to a list of message recipients;
- users leaving the site need to be removed from the recipient list; and
- chat messages need to be distributed to all users in the list.

With this in mind, we begin by importing a few modules we are going to need and define the type for our recipient list.

```
import Haste.App
import Haste.App.Concurrent
import qualified Control.Concurrent as CC

type Recipient = (SessionID, CC.Chan String)
type RcptList = CC.MVar [Recipient]
```

We use an `MVar` from `Control.Concurrent` to store the list of recipients. A recipient will be represented by a `SessionID`, an identifier used by Haste.App to identify user sessions, and an `MVar` into which new chat messages sent to the recipient will be written as they arrive. Next, we define our handshake RPC function.

```
srvHello :: Server RcptList → Server ()
srvHello remoteRcpts = do
  recipients ← remoteRcpts
  sid ← getSessionID
  liftIO $ do
    rcptChan ← CC.newChan
    CC.modifyMVar recipients $ λcs →
      return ((sid, rcptChan):cs, ())
```

An `MVar` is associated with the connecting client's session identifier, and the pair is prepended to the recipient list. Notice how the application's server state is passed in as the function's argument, wrapped in the `Server` monad in order to prevent client-side inspection.

```
srvSend :: Server RcptList → String → Server ()
srvSend remoteRcpts message = do
```

```
    rcpts ← remoteRcpts
    liftIO $ do
      recipients ← CC.readMVar rcpts
      mapM_ (flip CC.writeChan message) recipients
```

The send function is slightly more complex. The incoming message is written to the `Chan` corresponding to each active session.

```
srvAwait :: Server RcptList → Server String
srvAwait remoteRcpts = do
  rcpts ← remoteRcpts
  sid ← getSessionID
  liftIO $ do
    recipients ← CC.readMVar rcpts
    case lookup sid recipients of
      Just mv → CC.readChan mv
      _       → fail "Unregistered session!"
```

The final server operation, notifying users of pending messages, finds the appropriate `Chan` to wait on by searching the recipient list for the session identifier of the calling user, and then blocks until a message arrives in said `MVar`. This is a little different from the other two operations, which perform their work as quickly as possible and then return immediately.

If the caller's session identifier could not be found in the recipient list, it has for some reason not completed its handshake with the server. If this is the case, we simply drop the session by throwing an error; an exception will be thrown to the client. No server side state needs to be cleaned up as the very lack of such state was our reason for dropping the session.

Having implemented our three server operations, all that's left is to tie them to the client. In this tying, we see our main advantage over the JavaScript version in section 1 in action: the `remote` function builds a strongly typed bridge between the client and the server, ensuring that any future enhancements to our chatbox program are made safely, in one place, instead of being spread about throughout two disjoint code bases.

```
main :: App Done
main = do
  recipients ← liftServerIO $ CC.newMVar []

  hello ← remote $ srvHello recipients
  awaitMsg ← remote $ srvAwait recipients
  sendMsg ← remote $ srvSend recipients

  runClient $ do
    withElems ["log","message"] $ λ[log,msgbox] → do
      onServer hello
```

Notice that the `recipients` list is passed to our three server operations *before* they are imported; since `recipients` is a mutable reference created on the server and inaccessible to client code, it is not possible to pass it over the network as an RPC argument. Even if it were possible, passing server-private state back and forth

over the network would be quite inappropriate due to privacy and security concerns.

The `withElems` function is part of the Haste compiler's bundled DOM manipulation library; it locates references to the DOM nodes with the given identifiers and passes said references to a function. In this case the variable `log` will be bound to the node with the identifier "log", and `msgbox` will be bound to the node identified by "message". These are the same DOM nodes that were referenced in our original example, and refer to the chat log window and the text input field respectively. After locating all the needed UI elements, the client proceeds to register itself with the server's recipient list using the `hello` remote computation.

```
let recvLoop chatlines = do
      setProp log "value" $ unlines chatlines
      message ← onServer awaitMsg
      recvLoop (message : chatlines)
fork $ recvLoop []
```

The `recvLoop` function perpetually asks the server for new messages and updates the chat log whenever one arrives. Note that unlike the `onmessage` callback of the JavaScript version of this example, `recvLoop` is acting as a completely self-contained process with linear program flow, keeping track of its own state and only reaching out to the outside world to write its state to the chat log whenever necessary. As the `awaitMsg` function blocks until a message arrives, `recvLoop` will make exactly one iteration per received message.

```
msgbox `onEvent` OnKeyPress $ λ13 → do
  msg ← getProp msgbox "value"
  setProp msgbox "value" ""
  onServer (sendMsg <.> msg)
```

This is the final part of our program; we set up an event handler to clear the input box and send its contents off to the server whenever the user hits return (character code 13) while the input box has focus.

```
runClient    :: Client () → App Done
liftServerIO :: IO a → App (Server a)
remote       :: Remotable a
             ⇒ a → App (Remote a)

onServer     :: Remote (Server a) → Client a
(<.>)        :: Serialize a
             ⇒ Remote (a → b) → a → Remote b

getSessionID :: Server SessionID
```

Figure 3: Types of the Haste.App core functions

Function	Purpose
`runClient`	Lift a single `Client` computation into the `App` monad. Must be at the very end of an `App` computation, which is enforced by the type system.
`liftServerIO`	Lift an IO computation into the `App` monad. The computation and its result are exclusive to the server, as enforced by the type system, and are not observable on the client.
`remote`	Make a server side function available to be called remotely by the client.
`onServer`	Dispatch a remote call to the server and wait for its completion. The result of the remote computation is returned on the client after it completes.
`<.>`	Apply an `remote` function to a serializable argument.
`getSessionID`	Get the unique identifier for the current session. This is a pure convenience function, to relieve programmers of the burden of session bookkeeping.

Table 1. Core functions of Haste.App

The discerning reader may be slightly annoyed at the need to extract the contents from `Remote` values at each point of use. Indeed, in a simple example such as this, the source clutter caused by this

becomes a disproportionate irritant. Fortunately, most web applications tend to have more complex client-server interactions, reducing this overhead significantly.

A complete listing of the core functions in Haste.App is given in table 1, and their types are given in figure 3.

3. Implementation

Our implementation is built in three layers: the compiler layer, the concurrency layer and the communication layer. The concurrency and communication layers are simple Haskell libraries, portable to any other pair of standard Haskell compilers with minimal effort.

To pass data back and forth over the network, messages are serialized using JSON, a fairly lightweight format used by many web applications, and sent using the HTML5 WebSockets API. This choice is completely arbitrary, guided purely by implementation

convenience. It is certainly not the most performant choice, but can be trivially replaced with something more suitable as needed.

The implementation described here is a slight simplification of our implementation, removing some performance enhancements and error handling clutter in the interest of clarity. The complete implementation is available for download, together with the Haste compiler, from Hackage as well as from our website at

`http://haste-lang.org`.

Two compilers The principal trick to our solution is compiling the same program twice; once with a compiler that generates the server binary, and once with one that generates JavaScript. Conditional compilation is used for a select few functions, to enable slightly different behavior on the client and on the server as necessary. Using Haskell as the base language of our solution leads us to choose GHC as our server side compiler by default. We chose the Haste compiler to provide the client side code, mainly owing to our great familiarity with it and its handy ability to make use of vanilla Haskell packages from Hackage.

The `App` monad The `App` monad is where remote functions are declared, server state is initialized and program flow is handed over to the `Client` monad. Its definition is as follows.

```
type CallID = Int
type Method = [JSON] → IO JSON
type AppState = (CallID, [(CallID, Method)])
newtype App a = App (StateT AppState IO a)
  deriving (Functor, Applicative, Monad)
```

As we can see, `App` is a simple state monad, with underlying IO capabilities to allow server side computations to be forked from within it. Its `CallID` state element contains the identifier to be given to the next remote function, and its other state element contains a mapping from identifiers to remote functions.

What makes `App` interesting is that computations in this monad are executed on both the client and the server; once on server startup, and once in the startup phase of each client. Its operations behave slightly differently depending on whether they are executed on the client or on the server. Execution is deterministic, ensuring that the same sequence of `CallID`s are generated during every run, both on the server and on all clients. This is necessary to ensure that any particular call identifier always refers to the same server side function on all clients.

After all common code has been executed, the program flow diverges between the client and the server; client side, `runClient` launches the application's `Client` computation whereas on the server, this computation is discarded, and the server instead goes into an event loop, waiting for calls from the client.

The workings of the `App` monad basically hinges on the `Server` and `Remote` abstract data types. `Server` is the monad wherein any server side code is contained, and `Remote` denotes functions which live on the server but can be invoked remotely by the client. The implementation of these types and the functions that operate on them differ between the client and the server.

Client side implementations We begin by looking at the client side implementation for those two types.

```
data Server a = ServerDummy
data Remote a = Remote CallID [JSON]
```

The `Server` monad is quite uninteresting to the client; since operations performed within it can not be observed by the client in any

way, such computations are simply represented by a dummy value. The `Remote` type contains the identifier of a remote function and a list of the serialized arguments to be passed when invoking it. In essence, it is an explicit representation of a remote closure. Such closures can be applied to values using the `<.>` operator.

```
(<.>) :: Serialize a
      ⇒ Remote (a → b) → a → Remote b
(Remote identifier args) <.> arg =
  Remote identifier (toJSON arg : args)
```

The `remote` function is used to bring server side functions into scope on the client as `Remote` functions. It is implemented using a simple counter which keeps track of how many functions have been imported so far and thus which identifier to assign to the next remote function.

```
remote :: Remotable a ⇒ a → App (Remote a)
remote _ = App $ do
  (next_id, remotes) ← get
  put (next_id+1, remotes)
  return (Remote next_id [])
```

As the remote function lives on the server, the client only needs an identifier to be able to call on it. The remote function is thus ignored, so that it can be optimized out of existence in the client executable. Looking at its type, we can see that `remote` accepts any argument instantiating the `Remotable` class. `Remotable` is defined as follows.

```
class Remotable a where
```

```
  mkRemote :: a → ([JSON] → Server JSON)

instance Serialize a ⇒ Remotable (Server a) where
  mkRemote m = λ_ → fmap toJSON m

instance (Serialize a, Remotable b) ⇒
         Remotable (a → b) where
  mkRemote f =
    λ(x:xs) → mkRemote (f $ fromJSON x) xs
```

In essence, any function, over any number of arguments, which returns a serializable value in the `Server` monad can be imported. The `mkRemote` function makes use of a well-known type class trick for creating statically typed variadic functions, and works very much like the `printf` function of Haskell's standard library. [25]

The final function operating on these types is `liftServerIO`, used to initialize state holding elements and perform other setup functionality on the server.

```
liftServerIO :: IO a → App (Server a)
liftServerIO _ = App $ return ServerDummy
```

As we can see, the implementation is as simple as can be. Since `Server` is represented by a dummy value on the client, we just return said value.

Server side implementations The server side representation of the `Server` and `Remote` types are in a sense the opposites of their client side counterparts.

```
newtype Server a = Server (ReaderT SessionInfo IO a)
  deriving (Functor, Applicative, Monad, MonadIO)
data Remote a = RemoteDummy
```

Where the client is able to do something useful with the `Remote` type but can't touch `Server` values, the server has no way to inspect `Remote` functions, and thus only has a no-op implementation of the `<.>` operator. On the other hand, it does have full access to the values and side effects of the `Server` monad, which is an IO monad with some additional session data for the convenience of server side code.

`Server` values are produced by the `liftServerIO` and `remote` functions. `liftServerIO` is quite simple: the function executes its argument immediately and the result is returned, tucked away within the `Server` monad.

```
liftServerIO :: IO a → App (Server a)
liftServerIO m = App $ do
  x ← liftIO m
  return (return x)
```

The server version of `remote` is a little more complex than its client side counterpart. In addition to keeping track of the identifier of

the next remote function, the server side `remote` pairs up remote functions with these identifiers in an identifier-function mapping.

```
remote f = App $ do
  (next_id, remotes) ← get
  put (next_id+1, (next_id, mkRemote f) : remotes)
  return RemoteDummy
```

This concept of client side identifiers being sent to the server and used as indices into a table mapping identifiers to remotely accessible functions is an extension of the concept of "static values" introduced by Epstein et al with Cloud Haskell [12], which is discussed further in section 5.1.

The server side dispatcher After the `App` computation finishes, the identifier-function mapping accumulated in its state is handed over to the server's event loop, where it is used to dispatch the proper functions for incoming calls from the client.

```
onEvent :: [(CallID, Method)] → JSON → IO ()
onEvent mapping incoming = do
  let (nonce, identifier, args) = fromJSON incoming
      Just f = lookup identifier mapping
  result ← f args
  webSocketSend $ toJSON (nonce, result)
```

The function corresponding to the RPC call's identifier is looked up in the identifier-function mapping and applied to the received

list of arguments. The return value is paired with a nonce provided by the client to tie it to its corresponding RPC call, since there may be several such calls in progress at the same time. The pair is then sent back to the client.

Note that during normal operation, it is not possible for the client to submit an RPC call with a non-existent call identifier, hence the irrefutable pattern match on `Just f`. Should this pattern match fail, this is a sure sign of malicious tampering; the resulting exception is caught and the session is dropped as it is no longer meaningful to continue.

The `Client` monad and the `onServer` function As synchronous network communication is one of our stated goals, it is clear that we will need some kind of blocking primitive. Since JavaScript does not support any kind of blocking, we will have to implement this ourselves.

A solution is given in the *poor man's concurrency monad* [4]. Making use of a continuation monad with primitive operations for forking a computation and atomically lifting an IO computation into the monad, it is possible to implement cooperative multitasking on top of the non-concurrent JavaScript runtime. This monad allows us to implement `MVar`s as our blocking primitive, with the same semantics as their regular Haskell counterpart. [21] This concurrency-enhanced IO monad is used as the basis of the `Client` monad.

```
type Nonce = Int
type ClientState = (Nonce, Map Nonce (MVar JSON))
type Client = StateT ClientState Conc
```

Aside from the added concurrency capabilities, the `Client` monad

only has a single particularly interesting operation: `onServer`.

```
newResult :: Client (Nonce, MVar JSON)
newResult = do
  (nonce, m) ← get
  mv ← liftIO newEmptyMVar
  put (nonce+1, insert nonce var m)
  return (nonce, mv)

onServer :: Serialize a
         ⇒ Remote (Server a) → Client a
onServer (Remote identifier args) = do
  (nonce, mv) ← newResult
  webSocketSend $
    toJSON (nonce, identifier, reverse args)
  fromJSON <$> takeMVar mv
```

The `createResultMVar` function creates a new `MVar`, paired with its corresponding nonce in the

After a call is dispatched, `onServer` blocks, waiting for its *result variable* to be filled with the result of the call. Filling this variable is the responsibility of the *receive callback*, which is executed every time a message arrives from the server.

```
onMessage :: JSON → Client ()
onMessage response = do
  let (nonce, result) = fromJSON response
  (n, m) ← get
  put (n, delete nonce m)
  putMVar (m ! nonce) result
```

As we can see, the implementation of our programming model is rather simple and requires no bothersome compiler modifications or language extensions, and is thus easily portable to other Haskell compilers.

4. The Haste compiler

In order to allow the same language to be used on both client and server, we need some way to compile that language into JavaScript. To this end, we make use of the Haste compiler [11], started as an MSc thesis and continued as part of this work. Haste builds on the

GHC compiler to provide the full Haskell language, including most GHC-specific extensions, in the browser.

As Haste has not been published elsewhere, we describe here some key elements of its design and implementation which are pertinent to this work.

4.1 Choosing a compiler

Haste is by no means the only JavaScript-targeting compiler for a purely functional language. In particular, the GHC-based GHCJS [17] and UHC [8] compilers are both capable of compiling standard Haskell into JavaScript; the Fay [10] language was designed from

the ground up to target the web space using a subset of Haskell; and there exist solutions for compiling Erlang [13] and Clean [9] to JavaScript as well. While the aforementioned compilers are the ones most interesting for purely functional programming, there exist a wealth of other JavaScript-targeting compilers, for virtually any language.

Essentially, our approach is portable to any language or compiler with the following properties:

- The language must provide a static type system, since one of our primary concerns is to reduce defect rates through static typing of the client-server communication channel.
- The language must be compilable to both JavaScript and a format suitable for server side execution as we want our web applications to be written and compiled as a single program.
- We want the language to provide decent support for a monadic programming style, as our abstractions for cooperative multitasking and synchronous client-server communication are neatly expressible in this style.

As several of the aforementioned compilers fullfil these criteria, the choice between them becomes almost arbitrary. Indeed, as Haste.App is compiler agnostic, this decision boils down to one's personal preference. We chose to base our solution on Haste as we, by virtue of its authorship, have an intimate knowledge of its internal workings, strengths and weaknesses. Without doubt, others may see many reasons to make a different choice.

4.2 Implementation overview

Haste offloads much of the heavy lifting of compilation - parsing, type checking, intermediate code generation and many optimizations - onto GHC, and takes over code generation after the STG generation step, at the very end of the compilation process. STG [20] is the last intermediate representation used by GHC before the final code generation takes place and has several benefits for use as Haste's source language:

- STG is still a functional intermediate representation, based on the lambda calculus. When generating code for a high level target language such as JavaScript, where functions are first class objects, this allows for a higher level translation than when doing traditional compilation to lower level targets like stack machines or register machines. This in turn allows us to make more efficient use of the target language's runtime, leading to smaller, faster code.

- In contrast to Haskell itself and GHC's intermediate Core language, STG represents 'thunks', the construct used by GHC to implement non-strict evaluation, as closures which are explicitly created and evaluated. Closures are decorated with a wealth of information, such as their set of captured varibles, any type information needed for code generation, and so on. While extracting this information manually is not very hard, having this done for us means we can get away with a simpler compilation pipeline.

- The language is very small, essentially only comprising lambda

abstraction and application, plus primitive operations and facilities for calling out to other languages. Again, this allows the Haste compiler to be a very simple thing indeed.

- Any extensions to the Haskell language implemented by GHC will already have been translated into this very simple intermediate format, allowing us to support basically any extension GHC supports without effort.

- Application of external functions is always saturated, as is application of most other functions. This allows for compiling most function applications into simple JavaScript function calls, limiting the use of the slower dynamic techniques required to handle curried functions in the general case [16] to cases where it is simply not possible to statically determine the arity of a function.

In light of its heavy reliance on STG, it may be more correct to categorize Haste as an STG compiler rather than a Haskell compiler.

4.3 Data representation

The runtime data representation of Haste programs is kept as close to regular JavaScript programs as possible. The numeric types are represented using the JavaScript `Number` type, which is defined as the IEEE754 double precision floating point type. This adds some overhead to operations on integers as overflow and non-integer divisions must be handled. However, this is common practice in hand-written JavaScript as well, and is generally handled efficiently by JavaScript engines.

Values of non-primitive data types in Haskell consist of a data constructor and zero or more arguments. In Haste, these values are represented using arrays, with the first element representing the data constructor and the following values representing its arguments. For instance, the value `42 :: Int` is represented as `[0, 42]`, the leading `0` representing the zeroth constructor of the `Int` type and the `42` representing the "machine" integer. It may seem strange that a limited precision integer is represented using one level of indirection rather than as a simple number, but recall that the `Int` type is defined by GHC as **`data`** `Int = I# Int#` where `Int#` is the primitive type for machine integers.

Functions are represented as plain JavaScript functions, one of the blessings of targeting a high level language, and application can therefore be implemented as its JavaScript counterpart in most cases. In the general case, however, functions may be curried. For such cases where the arity of an applied function can not be determined statically, application is implemented using the eval/apply method described in [16] instead.

4.4 Interfacing with JavaScript

While Haste supports the Foreign Function Interface inherited from GHC, with its usual features and limitations [21], it is often impractical to work within the confines of an interface designed for communication on a very low level. For this reason Haste sports its own method for interacting with JavaScript as well, which allows the programmer to pass any value back and forth between Haskell

```
import Haste.Foreign

-- A MutableVar is completely opaque to Haskell code
-- and is only ever manipulated in JavaScript. Thus,
-- we use the Unpacked type to represent it,
-- indicating a completely opaque value.
newtype MutableVar a = MV Unpacked

instance Marshal (MutableVar a) where
  pack          = MV
  unpack (MV x) = x

newMutable :: Marshal a ⇒ a → IO (MutableVar a)
newMutable = ffi "(function(x) {return {val: x};})"

setMutable :: Marshal a ⇒ MutableVar a → a → IO ()
setMutable = ffi "(function(m, x) {m.val = x;})"

getMutable :: Marshal a ⇒ MutableVar a → IO a
getMutable = ffi "(function(m) {return m.val;})"
```

Figure 4: Mutable variables with `Haste.Foreign`

and JavaScript, as long as she can come up with a way to translate this value between its Haskell and JavaScript representations. Not performing any translation at all is also a valid "translation", which allows Haskell code to store any JavaScript value for later retrieval without inspecting it and vice versa. The example given in figure 4 implements mutable variables using this custom JavaScript interface.

The core of this interface consists of the `ffi` function, which allows the programmer to create a Haskell function from arbitrary JavaScript code. This function exploits JavaScript's ability to parse and execute arbitrary strings at run time using the `eval` function, coupled with the fact that functions in Haste and in JavaScript share the same representation, to dynamically create a function object at runtime. The `ffi` function is typed using the same method as the `mkRemote` function described in section 3. When applied to one or more arguments instantiating the `Marshal` type class, the `pack` function is applied to each argument, marshalling them into their respective JavaScript representations, before they are passed to the dynamically created function. When that function returns, the inverse `unpack` function is applied to its return value before it is passed back into the Haskell world.

As the marshalling functions chosen for each argument and the foreign function's return value depends on its type, the programmer must explicitly specify the type of each function imported using `ffi`; in this, Haste's custom method is no different from the conventional FFI.

There are several benefits to this method, the most prominent being that new marshallable types can be added by simply instantiating a type class. Thanks to the lazy evaluation employed by Haste, each foreign function object is only created once and then cached; any further calls to the same (Haskell) function will reuse the cached function object. Implementation-wise, this method is also very non-intrusive, requiring only the use of the normal FFI to import JavaScript's `eval` function; no modification of the compiler is needed.

5. Discussion and related work

5.1 Related work

Several other approaches to seamless client-server interaction exist. In general, these proposed solutions tend to be of the "all or nothing" variety, introducing new languages or otherwise requiring custom full stack solutions. In contrast, our solution can be implemented entirely as a library and is portable to any pair of compilers supporting typed monadic programming. Moreover, Haste.App has a quite simple and controlled programming model with a clearly defined controller, which stands in contrast to most related work which embraces a more flexible but also more complex programming model.

The more notable approaches to the problem are discussed further in this section.

Conductance and Opa Conductance [6] is an application server built on StratifiedJS, a JavaScript language extension which adds a few niceties such as cooperative multitasking and more concise syntax for many common tasks. Conductance uses an RPC-based model for client-server communication, much like our own, but also adds the possibility for the server to independently transmit data back to the client through the use of shared variables or call back into the client by way of function objects received via RPC call, as well as the possibility for both client and server to seamlessly modify variables located on the opposite end of the network. Conductance is quite new and has no relevant publications. It is, however, used for several large scale web applications.

While Conductance gets rid of the callback-based programming model endemic to regular JavaScript, it still suffers from many of its usual drawbacks. In particular, the weak typing of JavaScript poses a problem in that the programmer is in no way reprimanded by her tools for using server APIs incorrectly or trying to transmit values which can not be sensibly serialized and de-serialized, such as DOM nodes. Wrongly typed programs will thus crash, or even worse, gleefully keep running with erroneous state due to implicit type conversions, rather than give the programmer some advance warning that something is amiss.

We are also not completely convinced that the ability to implicitly pass data back and forth over the network is a unilaterally good thing; while this indeed provides the programmer some extra convenience, it also requires the programmer to exercise extra caution to avoid inadvertently sending large amounts of data over the network or leak sensitive information.

The Opa framework [18], another JavaScript framework, is an improvement over Conductance by introducing non-mandatory type checking to the JavaScript world. Its communication model is based on implicit information flows, allowing the server to read and update mutable state on the client and vice versa. While this is a quite flexible programming model, we believe that this uncontrolled, implicit information flow makes programs harder to follow, debug, secure and optimize.

Google Web Toolkit Google Web Toolkit [26], a Java compiler targeting the browser, provides its own solution to client-server interoperability as well. This solution is based on callbacks, forcing developers to write code in a continuation passing style. It also suffers from excessive boilerplate code and an error prone configuration process. The programming model shares Haste.App's client centricity, relegating the server to serving client requests.

Duetto Duetto [22] is a C++ compiler targeting the web, written from the ground up to produce code for both client and server simultaneously. It utilizes the new attributes mechanism introduced in C++11 [24] to designate functions and data to live on either client or server side. Any calls to a function on the other side of the network and attempts to access remote data are implicit, requiring no extra annotations or scaffolding at the call site. Duetto is still a highly experimental project, its first release being only a few months old, and has not been published in any academic venue.

Like Conductance, Duetto suffers somewhat from its heritage:

while the client side code is not memory-unsafe, as it is not possible to generate memory-unsafe JavaScript code, its server side counterpart unfortunately is. Our reservations expressed about how network communication in Duetto can be initiated implicitly apply to Duetto as well.

Sunroof In contrast to Conductance and Duetto, Sunroof [2] is an embedded language. Implemented as a Haskell library, it allows the programmer to use Haskell to write code which is compiled to JavaScript and executed on the client. The language can best be described as having JavaScript semantics with Haskell's type system. Communication between client and server is accomplished through the use of "downlinks" and "uplinks", allowing for data to be sent to and from the client respectively.

Sunroof is completely type-safe, in the DSL itself as well as in the communication with the Haskell host. However, the fact that client and server must be written in two separate languages - any code used to generate JavaScript must be built solely from the primitives of the Sunroof language in order to be compilable into JavaScript, precluding use of general Haskell code - makes code reuse hard. As the JavaScript DSL is executed from a native Haskell host, Sunroof's programming model can be said to be somewhat server centric, but with quite some flexibility due to its back and forth communication model.

Ocsigen Ocsigen [1] enables the development of client-server web applications using O'Caml. Much like Opa, it accomplishes typed, seamless communication by exposing mutable variables across the network, giving it many of the same drawbacks and

benefits. While Ocsigen is a full stack solution, denying the developer some flexibility in choosing their tools, it should be noted that said stack is rather comprehensive and well tested.

AFAX AFAX [19], an F#-based solution, takes an approach quite similar to ours, using monads to allow client and server side to coexist in the same program. Unfortunately, using F# as the base of such a solution raises the issue of side effects. Since any expression in F# may be side effecting, it is quite possible with AFAX to perform a side effect on the client and then attempt to perform some action based on this side effect on the server. To cope with this, AFAX needs to introduce cumbersome extensions to the F# type system, making AFAX exclusive to Microsoft's F# compiler and operating system, whereas our solution is portable to any pair of Haskell compilers.

HOP, Links, Ur/Web and others In addition to solutions which work within existing languages, there are several languages specifically crafted targeting the web domain. These languages target not only the client and server tiers but the database tier as well, and incorporate several interesting new ideas such as more expressive type systems and inclusion of typed inline XML code. [23][5][3] As our solution aims to bring typed, seamless communication into the existing Haskell ecosystem without language modifications, these languages solve a different set of problems.

Advantages of our approach We believe that our approach has a number of distinct advantages to the aforementioned attacks on the problem.

Our approach gives the programmer access to the same strongly

typed, general-purpose functional language on both client and server; any code which may be of use to both client and server is effortlessly shared, leading to less duplication of code and increased possibilities for reusing third party libraries.

Interactive multiplayer games are one type of application where this code sharing may have a large impact. In order to ensure that players are not cheating, a game server must keep track of the entire game state and send updates to clients at regular intervals. However, due to network latency, waiting for server input before rendering each and every frame is completely impractical. Instead, the usual approach is to have each client continuously compute the state of the game to the best of its knowledge, rectifying any divergence from the game's "official" state whenever an update arrives from the server. In this scenario, it is easy to see how reusing much of the same game logic between the client and the server would be very important.

Any and all communication between client and server is both strongly typed and made explicit by the use of the `onServer` function, with the programmer having complete control over the serialization and de-serialization of data using the appropriate type classes. Aside from the obvious advantages of type safety, making the crossing of the network boundary explicit aids the programmer in making an informed decision as to when and where server communication is appropriate, as well as helps prevents accidental transmission of sensitive information intended to stay on either side of the network.

Our programming model is implemented as a library, assuming only two Haskell compilers, one targeting JavaScript and one targeting the programmer's server platform of choice. While we use

Haste as our JavaScript-targeting compiler, modifying our implementation to use GHCJS or even the JavaScript backend of UHC would be trivial. This implementation not only allows for greater flexibility, but also eliminates the need to tangle with complex compiler internals.

Inspiration and alternatives to `remote` One crucial aspect of implementing cross-network function calls is the issue of data representation: the client side of things must be able to obtain some representation of any function it may want to call on the server.

In our solution, this representation is obtained through the use of the `remote` function, which when executed on the server pairs a function with a unique identifier, and when executed on the client returns said identifier so that the client may now refer to the function. While this has the advantage of being simple to implement, one major drawback of this method is that all functions must be explicitly imported in the `App` monad prior to being called over the network.

This approach was inspired by Cloud Haskell [12], which introduces the notion of "static values"; values which are known at compile time. Codifying this concept in the type system, to enable it to be used as a basis for remote procedure calls, unfortunately requires some major changes to the compiler. Cloud Haskell has a stopgap measure for unmodified compilers wherein a remote table, pairing values with unique identifiers, is kept. This explicit bookkeeping relies on the programmer to assign appropriate types to both values themselves and their identifiers, breaking type safety.

The astute reader may notice that this is exactly what the `remote` function does as well, the difference being that `remote` links the identifier to the value it represents on the type level, making it impossible to call non-existent remote functions and break the program's type safety in other ways.

Another approach to this problem is defunctionalization [7], a program transformation wherein functions are translated into algebraic data types. This approach would allow the client and server to use the same actual code; rather than passing an identifier around, the client would instead pass the actual defunctionalized code to the server for execution. This would have the added benefit of allowing functions to be arbitrarily composed before being remotely invoked.

This approach also requires significant changes to the compiler, making it unsuitable for our use case. Moreover, we are not entirely convinced about the wisdom of allowing server side execution of what is essentially arbitrary code sent from the client which, in a web application context, is completely untrustworthy. While analyzing code for improper behavior is certainly possible, designing and enforcing a security policy sufficiently strict to ensure correct behavior while flexible enough to be practically useful would be an unwelcome burden on the programmer.

5.2 Limitations

Client-centricity Unlike most related work, our approach takes a firm stand, regarding the client as the driver in the client-server relationship with the server taking on the role of a passive computational or storage resource. The server may thus not call back into the client at arbitrary points but is instead limited to returning answers to client side queries. This is clearly less flexible than the back-and-forth model of Sunroof and Duetto or the shared variables of Conductance. However, we believe that this restriction makes program flow easier to follow and comprehend. Like the immutability of Haskell, this model gives programmers a not-so-subtle hint as to how they may want to structure their programs. Extending our existing model with an `onClient` counterpart to `onServer` would be a simple task, but we are not quite convinced that there is value in doing so.

Environment consistency As our programming model uses two different compilers to generate client and server code, it is crucial to keep the package environments of the two in sync. A situation where, for instance, a module is visible to one compiler but not to the other will render many programs uncompilable until this inconsistency is fixed.

This kind of divergence can be worked around using conditional compilation, but is highly problematic even so; using a unified package database between the two compilers, while problematic due to the differing natures of native and JavaScript compilation respectively, would be a significant improvement in this area.

6. Future work

Information flow control Web applications often make use of a wide range of third party code for user tracking, advertising, collecition of statistics and a wide range of other tasks. Any piece of code executing in the context of a particular web session may not only interact with any other piece of code executing in the same context, but may also perform basically limitless communication with third parties and may thus, inadvertently or not, leak information about the application state. This is of course highly undesirable for many applications, which is why there is ongoing work in controlling the information flow within web applications [14].

While this does indeed provide an effective defence towards attackers and programming mistakes alike, there is value in being able to tell the two apart, as well as in catching policy violations resulting from programming mistakes as early as possible. An interesting venue of research would be to investigate whether we can take advantage of our strong typing to generate security policies for such an information flow control scheme, as well as ensure that this policy is not violated at compile time. This could shorten development cycles as well as give a reasonable level of confidence that any run time policy violation is indeed an attempted attack.

Real world applications As Haste.App is quite new and experimental, it has yet to be used in the creation of large scale applications. While we have used it to implement some small applications, such as a spaced repetition vocabulary learning program and a more featureful variant on the chatbox example given in section 2.3, further investigation of its suitability for larger real world applications

through the development of several larger scale examples is an important area of future work.

7. Conclusion

We have presented a programming model which improves on the current state of the art in client-server web application development. In particular, our solution combines type safe communication between the client and the server with functional semantics, clear demarcations as to when data is transmitted and where a particular piece of code is executed, and the ability to effortlessly share code between the client and the server.

Our model is client-centric, in that the client drives the application while the server takes on the role of passively serving client requests, and is based on a simple blocking concurrency model rather than explicit continuations. It is well suited for use with a GUI programming style based on self-contained processes with local state, and requires no modification of existing tools or compilers, being implemented completely as a library.

Acknowledgments

This work has been partially funded by the Swedish Foundation for Strategic Research, under grant RAWFP.

References

[1] V. Balat. ”Ocsigen: typing web interaction with objective Caml.”

Proceedings of the 2006 workshop on ML. ACM, 2006.

[2] J. Bracker and A. Gill. ”Sunroof: A Monadic DSL for Generating JavaScript.” In Practical Aspects of Declarative Languages, pp. 65-80. Springer International Publishing, 2014.

[3] A. Chlipala. ”Ur: statically-typed metaprogramming with type-level record computation.” ACM Sigplan Notices. Vol. 45. No. 6. ACM, 2010.

[4] K. Claessen. ”Functional Pearls: A poor man’s concurrency monad.” Journal of Functional Programming 9 (1999): 313-324.

[5] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In Formal Methods for Components and Objects (pp. 266-296). Springer Berlin Heidelberg, 2007.

[6] The Conductance application server. Retrieved March 1, 2014, from http://conductance.io.

[7] O. Danvy and L. R. Nielsen. ”Defunctionalization at work.” In Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming, pp. 162-174. ACM, 2001.

[8] A. Dijkstra, J. Stutterheim, A. Vermeulen, and S. D. Swierstra. ”Building JavaScript applications with Haskell.” In Implementation and Application of Functional Languages, pp. 37-52. Springer Berlin Heidelberg, 2013.

[9] L. Domoszlai, E. Bruël, and J. M. Jansen. ”Implementing a non-strict purely functional language in JavaScript.” Acta Universitatis Sapientiae 3 (2011): 76-98.

[10] C. Done. (2012, September 15). “Fay, JavaScript, etc.”, Retrieved March 1, 2014, from http://chrisdone.com/posts/fay.

[11] A. Ekblad. "Towards a declarative web." Master of Science Thesis, University of Gothenburg (2012).

[12] J. Epstein, A. P. Black, and S. Peyton-Jones. "Towards Haskell in the cloud." In ACM SIGPLAN Notices, vol. 46, no. 12, pp. 118-129. ACM, 2011.

[13] G. Guthrie. (2014, January 1). "Your transpiler to JavaScript toolbox". Retrieved March 1, 2014, from http://luvv.ie/2014/01/21/your-transpiler-to-javascript-toolbox/.

[14] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. "JSFlow: Tracking information flow in JavaScript and its APIs." In Proc. 29th ACM Symposium on Applied Computing. 2014.

[15] P. Lubbers and F. Greco. "Html5 web sockets: A quantum leap in scalability for the web." SOA World Magazine (2010).

[16] S. Marlow, and S. Peyton Jones. "Making a fast curry: push/enter vs. eval/apply for higher-order languages." In ACM SIGPLAN Notices, vol. 39, no. 9, pp. 4-15. ACM, 2004.

[17] V. Nazarov. "GHCJS Haskell to JavaScript Compiler". Retrieved March 1, 2014, from https://github.com/ghcjs/ghcjs.

[18] The Opa framework for JavaScript. Retrieved May 2, 2014, from http://opalang.org.

[19] T. Petricek, and Don Syme. "AFAX: Rich client/server web applications in F#." (2007).

[20] S. Peyton Jones. "Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine." J. Funct. Program. 2, no. 2 (1992): 127-202.

[21] S. Peyton Jones. "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell." Engineering theories of software construction 180 (2001): 47-96.

[22] A. Pignotti. (2013, October 31). "Duetto: a C++ compiler for the Web

going beyond emscripten and node.js". Retrieved March 1, 2014, from http://leaningtech.com/duetto/blog/2013/10/31/Duetto-Released/.

[23] M. Serrano, E. Gallesio, and F. Loitsch. "Hop: a language for programming the web 2. 0." OOPSLA Companion. 2006.

[24] B. Stroustrup. (2014, January 21). "C++11 - the new ISO C++ standard." Retrieved March 1, 2014, from http://www.stroustrup.com/C++11FAQ.html.

[25] C. Taylor. (2013, March 1). "Polyvariadic Functions and Printf". Retrieved March 1, 2014, from http://chris-taylor.github.io/blog/2013/03/01/how-haskell-printf-works/.

[26] S. Wargolet. "Google Web Toolkit. Technical report 12." University of Wisconsin-Platterville Department of Computer Science and Software Engineering, 2011.