A seamless, client-centric programming model for type safe web applications

Koen Claessen and Anton Ekblad
Chalmers University of Technology
{koen, antonek}@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, and (3) functional and type-safe; 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, 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 attempts have been made at tackling this problem, but none is, in our view, satisfactory. In this paper, we propose a functional programming model in which a web application is written as a single program from which client and server binaries 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 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, 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 communication, 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 [12], we may come up with something like the program in figure 1.

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 messages respectively. When a message arrives, it is prepended to the chat log, making the most recent message appear at the 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.
function handshake(sock) {sock.send('helol.');}
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) {
                msg = msgbox.value;
                chat(sock);
            }
        });
    }
};

Figure 1: Javascript chatbox implementation

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 call-backs, 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 delegates the server to 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.
clientMain :: App Done
clientMain = do
  greetings ← export helloServer
  runClient $ do
    name ← prompt "Hi there, what is your name?"
    onServer (greetings <$> name)

This is the entry point of our application. Recall that Haste.App applications are written from the client’s perspective, making this the starting point of the client-side code.

The export 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 exported function. In this example, the type of greetings is Export (String → Server ()), indicating that the identifier refers to an exported function with a single String argument and no return value. Exported functions all live in the Server monad. This part of the program is executed on both the server and the client, albeit with slightly different side effects, as described in section 3.

After the export 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 export, we must supply it with an argument. This is done using the <$> operator. It might be interesting to note that its type, Serialize a ⇒ Export (a → b) → a → Export b, is very similar to the type of the <$> operator over applicative functors. This is not a coincidence; <$> performs the same role for the Export type as <*> performs for applicative functors. The reason for using a separate operator for this instead of making Export an instance of Applicative is that since functions embedded in the Export type exist only to be called over a network, such functions must only be applied to arguments which can be serialized and sent over a network connection. When an Export function is applied to an argument using <$>, the argument is serialized and stored inside the resulting Export object, awaiting dispatch. Export values can thus be seen as explicit representations of closures.

After applying the value obtained from the user to the exported 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.

Finally, we need to launch our application and inform the client part of the whereabouts of its server counterpart. In this example the server will listen for connecting clients on port 1111, and since we will only run this application locally the client should attempt to contact the server over the loopback interface.

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

The URL of the server is provided for the benefit of the client and the port to listen on is provided for the benefit of the server. The final argument is the main function of our application. Figure 2 gives the code of this example in its entirety.

Like runClient, runApp does not return, and is intended to be the only computation executed by main. In fact, performing non-deterministic IO and allowing runApp or its arguments to capture the result may lead to unpredictable behavior. The clientMain function should for all intents and purposes be regarded as the program’s entry point. A non-intrusive method of enforcing this using the type system is described in section 5.2, but is not used in the examples given here as it relies on the use of the GHC plugin system.

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. At first glance, this would seem to be a trivial problem; can we not just create our state holding elements in main as usual? Figure 3 describes an attempt at a solution using this idea.

While this may at first seem to solve the problem of server-side state, this approach raises a number of questions. Does visitorref live on the client or the server, or perhaps on both? What happens if our state holding element is initialized in a non-deterministic way, and what if we assign it a value on the server and then attempt to read that value on the client? Recall that, as explained in section 2.1,
Therein lies the secret to restricting opaque and thus useless unless its contained value can somehow be extracted, something which is only possible using a single function:

\[
\text{mkUseful :: Useless a → Server a}
\]

Therein lies the secret to restricting liftServerIO to the server: the only function capable of turning a Useless value into a useful one lives in the server monad. 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 4 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.

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

-- Recipient type
type Recipient = (SessionID, CC.MVar String)

-- List of recipients
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.

```haskell
srvHello :: Useless RcptList → Server ()
srvHello uRcpts = do
    recipients ← mkUseful uRcpts
    sid ← getSessionID
    liftIO $ do
        rcptMVar ← CC.newEmptyMVar
        CC.modifyMVar recipients $ \cs → return ((sid, rcptMVar):cs ,())

The handshake is a simple operation; 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 first Useless RcptList argument.

```haskell
srvSend :: Useless RcptList → String → Server ()
srvSend uRcpts message = do
    rcpts ← mkUseful uRcpts
    liftIO $ do
        recipients ← CC.readMVar rcpts
        mapM_ (CC.forkI0 . deliver message) recipients
        where
            deliver msg (_,rcptMVar) = CC.putMVar rcptMVar msg
```

The send function is slightly more complex. The incoming message is written to the MVar corresponding to each active session. Note the use of forkI0. Some recipients may be slow to empty their MVar to make room for a new message. In this case, we don’t want to force all recipients to wait for the slowest one, so we spawn a new lightweight thread to handle delivery for each recipient.

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

The final server operation, notifying users of pending messages, finds the appropriate MVar 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; the session will be automatically cleaned up on the server side and an exception is thrown to the client.

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 export 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.

```haskell
clientMain :: App Done
clientMain = do
  recipients ← liftServerIO $ CC.newMVar []
  hello ← export $ srvHello recipients
  awaitMsg ← export $ srvAwait recipients
  sendMsg ← export $ srvSend recipients

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

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

Notice that the recipients list is passed to our three server operations before they are exported; 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 export.

```haskell
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.

```haskell
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. Finally, all that is left is to actually launch our application.

```haskell
clientMain
```

The discerning reader may be slightly annoyed at the need to extract the contents from `Useless` 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 5.

### 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 our source repository at http://ekblad.cc/haste.

#### 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. As we believe that a monadic programming style is a good fit for this application, we have chosen to use Haskell as the base of our

---

**Figure 5:** Types of the Haste.App core functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>runApp</td>
<td>:: AppCfg → App Done → IO ()</td>
</tr>
<tr>
<td>runClient</td>
<td>:: Client () → App Done</td>
</tr>
<tr>
<td>liftServerIO</td>
<td>:: IO a → App (Useless a)</td>
</tr>
<tr>
<td>export</td>
<td>:: Exportable a ⇒ a → App (Export a)</td>
</tr>
<tr>
<td>onServer</td>
<td>:: Export (Server a) → Client a</td>
</tr>
<tr>
<td>(.&lt;.)</td>
<td>:: Serialize a ⇒ Export (a → b) → a → Export b</td>
</tr>
<tr>
<td>mkUseful</td>
<td>:: Useless a → Server a</td>
</tr>
<tr>
<td>getSessionID</td>
<td>:: Server SessionID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Types of the Haste.App core functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client a</td>
</tr>
<tr>
<td>Server a</td>
</tr>
<tr>
<td>App Done</td>
</tr>
<tr>
<td>App (Useless a)</td>
</tr>
<tr>
<td>App (Export a)</td>
</tr>
<tr>
<td>App (Server a)</td>
</tr>
<tr>
<td>App (Useless a) → Server a</td>
</tr>
<tr>
<td>App (Export a) → Server a</td>
</tr>
<tr>
<td>App (Server a) → Client a</td>
</tr>
<tr>
<td>App (Useless a) → Server a → Export b</td>
</tr>
<tr>
<td>Server SessionID</td>
</tr>
<tr>
<td>IO a</td>
</tr>
<tr>
<td>AppCfg</td>
</tr>
<tr>
<td>mkConfig &quot;ws://localhost:1111&quot; 1111</td>
</tr>
</tbody>
</table>

---

2014/3/3
The purpose of the Haste.App program is to begin execution of a Haste.App program. Lift an IO computation into the App monad. Must be at the very end of an App computation, which is enforced by the type system. 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. Make a server-side function available to be called remotely by the client. 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 exported function to a serializable argument. Extract the contents of a Useless value produced by liftServerIO on the server. Get the unique identifier for the current session. This is a pure convenience function, to relieve programmers of the burden of session bookkeeping.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>runApp</td>
<td>Begin execution of a Haste.App program.</td>
</tr>
<tr>
<td>runClient</td>
<td>Lift a single Client computation into the App monad. Must be at the very</td>
</tr>
<tr>
<td></td>
<td>end of an App computation, which is enforced by the type system.</td>
</tr>
<tr>
<td>liftServerIO</td>
<td>Lift an IO computation into the App monad. The computation and its result</td>
</tr>
<tr>
<td></td>
<td>are exclusive to the server, as enforced by the type system, and are not</td>
</tr>
<tr>
<td></td>
<td>observable on the client.</td>
</tr>
<tr>
<td>export</td>
<td>Make a server-side function available to be called remotely by the client.</td>
</tr>
<tr>
<td>onServer</td>
<td>Dispatch a remote call to the server and wait for its completion. The</td>
</tr>
<tr>
<td></td>
<td>result of the remote computation is returned on the client after it</td>
</tr>
<tr>
<td></td>
<td>completes.</td>
</tr>
<tr>
<td>&lt;.&gt;</td>
<td>Apply an exported function to a serializable argument.</td>
</tr>
<tr>
<td>mkUseful</td>
<td>Extract the contents of a Useless value produced by liftServerIO on the</td>
</tr>
<tr>
<td></td>
<td>server.</td>
</tr>
<tr>
<td>getSessionID</td>
<td>Get the unique identifier for the current session. This is a pure</td>
</tr>
<tr>
<td></td>
<td>convenience function, to relieve programmers of the burden of session</td>
</tr>
<tr>
<td></td>
<td>bookkeeping.</td>
</tr>
</tbody>
</table>

Table 1. Core functions of Haste.App

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.

```
instance Exportable a where
    mkExported :: a -> (JSON -> Server JSON)
```

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 exported function, and its other state element contains a mapping from identifiers to exported 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 CallIDs 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.

Aside from runClient, the App monad has two primary operations, both of which have slightly different implementations on the client and server respectively.

```
data Useless a = Useful a | Useless
data Export a = Export CallID [JSON]

liftServerIO :: IO a -> App (Useless a)
export :: Exportable a => a -> App (Export a)
```

The client-side definition of liftServerIO is as simple as can be.

```
liftServerIO _ = App $ return Useless
```

Its argument is simply ignored, and an opaque dummy value is returned; recall that server-lifted IO computations are not observable on the client. The server-side implementation is only marginally more complex.

```
liftServerIO m = App $ fmap Useful m
```

Its argument IO computation is executed, and the result is wrapped in the Useful constructor. The counterpart function, mkUseful simply unwraps the Useful constructor and returns the contained value in the Server monad.

```
mkUseful :: Useless a -> Server a
mkUseful (Useful x) = return x
```

export is a more interesting function. Looking at its type, we see that it accepts any Exportable value for its argument. Exportable is defined as follows.

```
class Exportable a where
    mkExported :: a -> ([JSON] -> Server JSON)
```

In essence, any function, over any number of arguments, which returns a serializable value in the Server monad can be exported. The mkExported 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. [19]

The client implementation of export is not very complex.

```
export _ = App $ do
    (next_id, exports) <- get
    put (next_id+1, exports)
    return (Export next_id [])
```

As the exported function lives on the server, the client only needs an identifier to be able to call on it. The exported function is thus ignored, so that it can be optimized out of existence in the client version of the code. The server version is only marginally more
complex in that it updates the identifier-function mapping table as well.

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 [9], which is discussed further in section 5.1.

The second argument of the Export constructor is the list of arguments to be sent to the export, to be updated as the exported function is applied using the \(<.>\) operator.

\[
\text{do}
\begin{align*}
\text{next\_id} & = \text{App} \quad \text{\$ do} \\
& (\text{next\_id}, \text{exports}) \leftarrow \text{get} \\
& \text{put} \quad (\text{next\_id} + 1, (\text{next\_id}, \text{mkExported f}) : \text{exports}) \\
& \text{return} \quad (\text{Export} \quad \text{next\_id} \quad [])
\end{align*}
\]

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 bundled up with the \(\text{a}\) nonce, provided by the client to match each return value to the RPC call that produced it since there may be several RPC calls in progress at the same time, and sent back to the client.

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.

\[
\text{do}
\begin{align*}
\text{onEvent} & \quad :: \quad (\text{\[(\text{CallID}, \text{Method})]\} \rightarrow \text{JSON} \rightarrow \text{IO} \quad ()) \\
\text{onEvent} & \quad \text{mapping} \quad \text{incoming} = \text{do} \\
\text{let} \quad & \quad : \quad (\text{nonce}, \text{identifier}, \text{args}) = \text{fromJSON} \quad \text{incoming} \\
\text{Just f} & = \text{lookup} \quad \text{identifier} \quad \text{mapping} \\
\text{result} & \leftarrow \quad \text{f} \quad \text{args} \\
\text{webSocketSend} & \quad \text{\$ JSON} \quad (\text{nonce}, \text{result})
\end{align*}
\]

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 bundled up with the \(\text{a}\) nonce, provided by the client to match each return value to the RPC call that produced it since there may be several RPC calls in progress at the same time, and 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 \(\text{Just f}\). 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.

\textbf{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 [2]. 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 MVars as our blocking primitive, with the same semantics as their regular Haskell counterpart. [16] This concurrency-enhanced monad is used as the basis of the Client monad.

\[
\text{type} \quad \text{Nonce} = \text{Int} \\
\text{type} \quad \text{Client} = \text{StateT} \quad \text{Nonce} \quad \text{Conc}
\]

Aside from the added concurrency capabilities, the Client monad only has a single particularly interesting operation: \text{onServer}.

\[
\text{onServer} \quad :: \quad \text{Serialize a} \\
\quad \rightarrow \quad \text{Export (Server a)} \\
\text{onServer} \quad (\text{Export identifier args}) \quad \text{do} \\
\text{nonce} \quad \leftarrow \quad \text{get} \\
\text{put} \quad (\text{nonce} + 1) \\
\text{mv} \quad \leftarrow \quad \text{createResultMVar nonce} \\
\text{webSocketSend} \quad \text{\$ toJSON} \quad (\text{nonce}, \text{identifier}, \text{reverse args}) \\
\text{takeMVar mv}
\]

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

\[
\text{onMessage} \quad :: \quad \text{JSON} \rightarrow \text{Client ()} \\
\text{onMessage} \quad \text{response} = \text{do} \\
\text{let} \quad & \quad : \quad (\text{nonce}, \text{result}) = \text{fromJSON response} \\
\text{mv} \quad \leftarrow \text{findResultMVar nonce} \\
\text{putMVar} \quad \text{mv} \quad \text{result}
\]

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 [8], 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 [14] and UHC [5] compilers are both capable of compiling standard Haskell into Javascript; the Fay [7] language was designed from the ground up to target the web space using a subset of Haskell; and there exist solutions for compiling Erlang [10] and Clean [6] to Javascript as well.

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 we our abstractions for cooperative
multitasking and synchronous client-server communication are
neatly expressible in this style.

While several of the aforementioned compilers fulfill these criteria,
the choice between them becomes almost arbitrary. 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
different choice.

4.2 Design goals
Haste is first and foremost intended to be a practical compiler;
the ability to produce code which is lean and fast enough to be
used in the real world is critical. To this end, we make several
design choices and tradeoffs, some of which would be viewed as
detrimental in a more purist context.

One such tradeoff is the decision to make use of the facilities
provided by the web browser - most prominently the Javascript
engine’s native garbage collector - as far as possible rather than
implementing parts of the GHC runtime in Javascript. In the same
vein, a conscious effort is made to generate as “regular-looking”
code as possible. This results in large gains in code compactness
and execution speed - in our experiences, compiler Haste programs
are often an order of magnitude smaller and several times faster
than an equivalent program compiled with some other Haskell to
Javascript compiler.

In some rare cases, however, this means having to sacrifice com-
patibility with low-level GHC features. Most notably, Haste does
not support weak references or pre-emptive multitasking, as these
would require modifications to the runtime system and generated
code which confound the structure of the resulting Javascript and
have a detrimental impact on code size and execution speed.

4.3 Implementation overview
As previously mentioned, Haste offloads much of the heavy lifting
of compilation - parsing, type checking, intermediate code gen-
eration and many optimizations - onto GHC, and takes over code
generation after the STG generation step, at the very end of the
compilation process. STG [15] is the last intermediate representa-
tion 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 lan-
guage, STG represents ‘thunks’, the construct used by GHC to
implement non-strict evaluation, as closures which are explicit-
ly created and evaluated. Closures are decorated with a wealth
of information, such as their set of captured variables, any type
information needed for code generation, and so on. While ex-
tracting 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 facil-
ities 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 inter-
mediate format, allowing us to support basically any extension
GHC supports without effort.

• Application of external functions is always saturated, as is ap-
plication 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 [13] 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.

4.4 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 con-
structor and zero or more arguments. In Haste, these values are rep-
resented using arrays, with the first element representing the data
constructor and the following values representing its arguments.
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
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 6: Mutable variables with Haste.Foreign

Figure 6 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 mkExported 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

Other approaches to this problem exist, notably in the form of the Conductance [3], Duetto [17] and Sunroof [1] projects, all of which implement some synchronous abstraction over the network between the client and the server, much like our solution.

Conductance Conductance 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 convenience; while this indeed provides the programmer some extra freedom, it also requires the programmer to exercise extra caution to avoid inadvertently sending large amounts of data over the network.

Duetto Duetto 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 [18] 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 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.
**Advantages to 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 is 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 export**  
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 `export` function, which when executed on the server pairs a function with a unique identifier and on the client, 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 exported in the App monad prior to being called over the network.

This approach was inspired by Cloud Haskell [9], 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 `export` function does as well, the difference being that `export` 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 [4], 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 Conductance and Duetto, 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. Should this model prove insufficiently expressive, extending our existing model with an `onClient` counterpart to `onServer` is a simple task.

**Preventing divergence**  
While our solution completely safe from the invocation point of the `runApp` function onward, capturing IO-dependent variables introduced before this invocation may lead to the client and server execution paths diverging. Recall that our solution does not modify either of the compilers used. As such, both will produce code to execute all computations called by `main`. Consider the following example:

```haskell
main = do
  exportTwo ← randomRIO (False, True)

runApp config $ do
  when exportTwo $ do
    → export (return () :: Server ())
    return ()
  fun ← export $ return "I'm a little teapot"

runClient $ do
  onServer fun >>= alert
```

Whenever `exportTwo` gets assigned a `True` value, this application will call `export` twice, assigning the identifier to the first exported computation, which is discarded, and 1 to the second, `fun`, which is
kept and invoked. If, on the other hand, `exportTwo` is `False`, only the second computation will be exported, and will thus be assigned the identifier 1.

This is perfectly fine as long as the client and server both assign the same identifier to `fun`. However, since `randomRIO` is not deterministic and since all computations called by `main` will be executed on both the client and the server, there is a possibility that `fun` will be assigned the identifier 0 on the client and 1 on the server. In this scenario, when the client attempts to call the return "I'm a little teapot" computation over the network, the server instead executes the return `()` computation, which returns `()` rather than `String`, breaking the type safety of our program.

This is easily remedied however, by a slight compiler modification. Instead of using `main :: IO ()` as the program entry point, we introduce a function `clientMain :: App` for this purpose, and let the runtime system start the application by invoking `runApp config clientMain`, where `config` contains the URL and port configuration for the server, given by the programmer using a compiler flag.

As this modification only affects the program’s entry point, which is already given “special” treatment in Haskell compilers, and may even be implemented using the plugin mechanism of GHC, requiring no modification of the compiler itself, we believe this to be a practical, non-intrusive way to remove this limitation, and are planning on releasing an implementation which includes this limitation.

The astute reader may notice that the same applies to code using `unsafePerformIO` to declare IO-dependent top-level values, and that this problem is unaffected by changing the type of the program’s entry point. While this is indeed true, we feel that since it is quite possible to break any Haskell program using `unsafePerformIO`, even subverting the type system, “solving” `unsafePerformIO` is not within the scope of this work.

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 solution where, for instance, a module is visible to one compiler but not to the other will render many programs un compilable 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, collection 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 [11].

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 runtime 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 essentially no modification of existing tools or compilers.

References


