An EDSL for distributed, heterogeneous web applications

Anton Ekblad
Chalmers University of Technology
antonek@chalmers.se

Abstract

We present a domain-specific language for constructing and configuring web applications distributed across any number of networked, heterogeneous systems. Our language is embedded in Haskell, provides a common framework for integrating components written in third-party EDSLs, and enables type-safe, access-controlled communication between nodes, as well as effortless sharing and movement of functionality between application components. We give an implementation of our language and demonstrate its applicability by using it to implement several important components of distributed web applications, including RDBMS integration, load balancing, and fine-grained sandboxing of untrusted third party code.

The rising popularity of cloud computing and heterogeneous computer architectures is putting a strain on conventional programming models, which commonly assume that one application executes on one machine, or at best on one out of several identical machines. With our language, we take the first step towards a programming model better suited for a computationally multicultural future.

CCS Concepts **•Computing methodologies → Distributed programming languages; •Software and its engineering → Distributed programming languages; Functional languages; Extensible languages; Multiparadigm languages;**

Keywords domain-specific languages, web applications, distributed applications

ACM Reference format:
Anton Ekblad. 2017. An EDSL for distributed, heterogeneous web applications. In *Proceedings of Haskell Symposium 2017, Oxford, United Kingdom, September 07–08, 2017 (Haskell '17),* 11 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 Introduction

1.1 Background

Conventional programming models often assume that one program corresponds to one application. With the rise of cloud computing and increasingly heterogeneous computing platforms, this assumption is becoming increasingly invalid. Modern applications, and web applications in particular, are more often than not distributed affairs, involving multiple nodes which often have very different programming models.

One example of this would be the so-called *smart home*, where refrigerators, lightbulbs, toasters and a myriad of other household objects are all connected to a central hub, which allows the whole home to be controlled remotely, or even automatically in response to data gathered in real-time from the various objects. Not only does a smart home application need to reliably integrate any number of distributed components, but those components generally have very different capabilities, and may even span several different networks. The central hub might be connected to the Internet for remote control, some sensors may actually be a smarter front-end for an array of "dumb" sensors, and so on.

However, languages and programming models are slow to catch up. Distributed applications are conventionally written, not as a single program, but as a set of independent programs communicating over a set of home-grown, more or less ad hoc protocols. While this model allows developers some flexibility in choosing the appropriate language for each component individually, it has several serious shortcomings:

- Code cannot be easily shared between components. Even when two components are written in the same language, sharing functionality between them involves adapting the build system or taking care to break shared functionality into libraries, even for minor, otherwise unrelated code fragments.
- Applications are not type-safe. Changes in one component without matching changes to another, which in a non-distributed application would have triggered a type error at build time, may easily slip into production as protocol incompatibilities between components. This can be particularly problematic – dangerous, even – for applications which make use of actuators to affect the physical world, as in the case of smart homes.
- Distribution requires significant overhead: network communication needs to be implemented, protocols designed, and networks configured. Even worse, this fragile machinery needs to be properly maintained, adding an extra cost to refactoring and bug fixing, as well as to feature development.

This forces developers into *early design decisions.* Once a piece of functionality is implemented in a particular component, it cannot be easily moved to another. However, deciding how and where any particular functionality should be run up front may not be optimal. Design decisions are best made with as much information available as possible; information which often becomes available only as a project makes gradual progress.

Heterogeneity and domain-specific languages In functional programming circles, *embedded domain-specific languages*, or EDSLs, are a popular design pattern for dealing with heterogeneity with regards to an application's problem domain, letting domain experts, who may not be experts in the EDSL's host language, productively express problems and their solutions in code, without having to take a detour via a host language expert. EDSLs are also being used to let developers program systems with heterogeneous components, such as GPUs or special-purpose many-core processors without having to leave the host language (Karácsony and Claessen 2016; Mainland and Morrisett 2010).

While helpful in managing local heterogeneity, EDSLs usually do not do much to help with *distributed* heterogeneity. Remotely executing code written in an EDSL often boils down to writing an ad hoc, boilerplate server which accepts commands from a client, translates them into an appropriate program in the EDSL, executes the resulting program, and sends the result back to the client. This

Haskell '17, Oxford, United Kingdom
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

mode of development suffers, just like the more general case discussed previously, from excessive development overhead and forced early design decisions.

Web-specific trust challenges While distributed applications in general often assume a model of mutual trust, in which all nodes making up the application are presumed to faithfully represent the intention of the developer, web applications are fundamentally incompatible with this assumption. By nature of executing locally on user-controlled machines, any application node representing the client part of a web application cannot be trusted not to have been tampered with. As a result, any distributed web application must treat any client nodes as adversaries.

Worse, unlike other software, web applications commonly load third-party dependencies from online sources during run-time. While developers may not always audit third party libraries as thoroughly as they perhaps ought to, with a conventional application the user can at least be reasonably sure that the code they are running is the code the developer did ship. When libraries are loaded from third-party sources at run-time, this assumption no longer holds. The application's attack surface expands to include every machine and network from which libraries are loaded. Even worse, a library developer may turn out to be less honest than initially assumed, surreptitiously replacing a previously audited library with a malicious one when least expected. For this reason, not only do any server nodes need to treat the client as an adversary, but the *client* also needs to treat *parts of itself* as an adversary!

1.2 A language for distributed web applications

This paper presents an EDSL for implementing distributed applications with heterogeneous components, with a particular focus on web applications. Our language is embedded in Haskell and uses its expressive type system to separate nodes based on their roles, capabilities, and localities. To produce binaries for an application's nodes, *the same source code* is compiled multiple times with different compilers or settings: once for browser-targeting nodes, and once for each distinct server-side binary desired. This lets an application span nodes running on any architecture that has a GHC-compatible Haskell compiler. Remote calls between nodes are synchronous and type-safe. By default, any node may call any other node, but a node may optionally declare that it may only be called by nodes that are instances of some type class. Calls may not be made to the main client node as there may be an arbitrary number of clients running – one for each user's web browser – with no way to distinguish them from each other. Programs written in third party EDSLs can be attached directly to the network by providing two simple type class instances, and integrate seamlessly with our language even in cases where there are significant differences in the type universes of the EDSL and the host language.

The structure of this paper In Sect. 2, we describe our language in further detail and implement several common building blocks of web applications, to give an intuition of its use and to demonstrate its flexibility. In Sect. 3, we describe the implementation of our language. In Sect. 4, we discuss in further detail the limitations, design decisions and tradeoffs of our language, as well as its relation to the previous state of the art. Finally, in Sect. 5, we note our conclusions, and discuss possible directions for future research into the topic.

Our contribution The contribution of this paper is threefold.

- We present a domain-specific language for developing distributed web applications. Our language is embedded in Haskell and improves upon the state of the art by enabling type-safe communication between any number of networked components with heterogeneous programming models, executing on any number of different platforms including web browsers.
- We show how our language provides a common framework for integrating domain-specific languages distributed over a network, and enables late reconfiguration of application structure and topology.
- We demonstrate the viability of our language by using it to implement several common building blocks of distributed web applications, including RDBMS integration, load balancing, and sandboxed execution of untrusted third party code.

2 The language

The discussion in Sect. 1 of the limitations of conventional languages and programming models makes it clear what we don't want to see in a language for distributed web applications. But then, what properties *do* we want such a language to have?

- We want our language to support constructing distributed applications as a single program, with boilerplate-free, type-checked communication between nodes. Nodes should be able to run on different platforms, including the user's web browser and some native server-side environment.
- Our language should also support sharing and moving code between nodes to the greatest extent possible, as well as reconfiguring the "shape" of the application with minimal fuss; adding or removing nodes, and re-shaping the network over which the nodes communicate.
- The language should enforce clear boundaries between nodes. As discussed in Sect. 1, web applications suffer from an endemic lack of trust, and any language targeting this domain should strive to make the flow of control and data between components clear, to minimize the risk of information leakage.
- Finally, the language should integrate cleanly with other domain-specific languages embedded in the host language, pre-existing as well as custom-made.

In this section, we give a brief overview of the interface and programming model of our language, and show how it fulfills said design goals.

2.1 Basic programming model

Our programming model is client-centric: programs in our language are written in a monad *Client*, loosely analogous to the *IO* monad inhabited by "normal" Haskell programs. Client code is compiled to JavaScript for execution in the user's web browser, constituting the main user interface of the application.

Distributed applications are constructed from a set of *nodes*, each node is made up of a data type $n : * \rightarrow *$, which is the type of computations performed on said node, and an instance of the *Node* type class, describing how the node integrates with the larger application. A computation of type $n\ a$ always executes on the node n. However, pure code is shared among all nodes, and polymorphic code can be called from any node which meets its type class constraints. Initially, *Client* is the only defined node. By default, any node which also implements a *MonadClient* type

```
type Endpoint :: ∗
type Import   :: ∗ → ∗
type RemotePtr f = StaticPtr (Import f)
type Client :: ∗ → ∗
type Server :: ∗ → ∗

class MonadIO m ⇒ MonadClient (m :: ∗ → ∗)

class Node (n :: ∗ → ∗) where
  type Allowed n (c :: ∗ → ∗) :: Constraint
  type Env n :: ∗
  endpoint :: Proxy n → Endpoint
  init :: Proxy n → CIO (Env n)

class Node n ⇒ Mapping (n :: ∗ → ∗) f where
  type Hask n f
  invoke :: Env n → n f → CIO (Hask n f)

remoteNode :: String → Int → Endpoint
localNode  :: Proxy a → Endpoint
remote     :: Export f ⇒ f → Import n f
dispatch   :: Dispatch f f' ⇒ RemotePtr f → f'
dispatchTo :: Dispatch f f'
           ⇒ Endpoint → RemotePtr f → f'
start      :: Node n ⇒ Proxy n → CIO ()
runApp     :: [CIO ()] → Client () → IO ()
```

Figure 1. The user-facing API of our language

class may call any other node. However, when defining a node, the implementor may choose to put additional restrictions on which other nodes may make calls to it.

Nodes expose *entry points* as *static pointers* (Epstein et al. 2011) on the program's top level. The static pointers extension, supported by GHC since version 7.10, introduces the *static* keyword to assign names to values that are known at compile time. Names assigned in this way are stable across the network, allowing different binaries produced from the same source to refer to each others' values.

The remainder of this section introduces our language by showing how a series of examples can be implemented using our language, the API of which is given in Fig. 1. Note the use of the *CIO* monad, where one would normally expect *IO*. As the Haste compiler, used by our implementation, does not support concurrency in the *IO* monad, our implementation uses *CIO* as an explicitly concurrent drop-in replacement. The *Dispatch* type class denotes any pair of nodes *f* and *f'* such that *f'* is able to make a remote call to *f*. This type class is covered in greater detail in Sect. 3.3.

To give an intuition of the basics of our language, Fig. 2 shows a "hello world" application with two nodes, the web-based client, and a single node running on a server, where the client uses a remote pointer to print a message on the server. Lines 1 and 2 insert the *Server* node into our application, specifying that it can be reached on the host example.com, on port 24601. This information is referred to as the node's *endpoint*. Note the use of the *static* keyword on line 5, which together with the call to *remote* introduces a remote pointer, and the *dispatch* function on line 8, which is used to remotely invoke the greeting function. The invocation of *start*, passed to the *runApp* function on line 7, indicates that any non-client binary of this application can handle requests to the *Server* node. The second argument to *runApp* is the entry point of the application's *Client* computation.

```
1  instance Node Server where
2    endpoint _ = remoteNode "example.com" 24601
3
4  greet :: RemotePtr (String → Server ())
5  greet = static (remote $ liftIO . putStrLn)
6
7  main = runApp [start (Proxy :: Proxy Server)] $ do
8    dispatch greet "Hello, server!"
```

Figure 2. Hello, server!

In a conventional web application, even this simple program would have required significant legwork: separate programs need to be written for client and server, a network transport protocol needs to be chosen, a data serialization format decided upon and implemented, and client-server communication code needs to be implemented. A lengthy and error-prone task, which is not made easier by the lack of type safety between the client and the server.

The discerning reader may find the syntax of remote imports – the *static* keyword and the way it combines with the *remote* function – to be slightly clunky. Regrettably, this syntax is forced upon us by the use of static pointers. However, in addition to allowing us to make use of static pointers, this syntax has the distinct advantage of making it crystal clear where the boundaries between nodes are. These sharp boundaries are helpful in avoiding accidental information leaks between nodes.

2.2 Compiling and executing programs

As our language is embedded into Haskell, compiling and executing programs follows the normal Haskell compilation workflows. There is, however, a twist: programs written in our language need to be compiled *several times*, possibly with different compilers, depending on the number of nodes they contain as well as their respective architectures.

The program in Fig. 2 is relatively simple, and consists of two components: the client which sends the greeting, and the server which receives it. This application needs to be built twice: once with a Haskell compiler which produces a JavaScript program, and once with a compiler which produces a native binary. Once built, the application is started by executing the native binary on example.com, and by loading the JavaScript program in a web browser.

For less trivial applications, spanning more than a single server node, the application may need to be built several times, up to once for each type of node in the application. In this case, each build should change the type of the *start* invocation to match the type of the node that is currently being built. However, each native binary may run any number of nodes, by adding an appropriately typed invocation of *start* for each node it intends to run, and a single binary may be run on more than one physical machine.

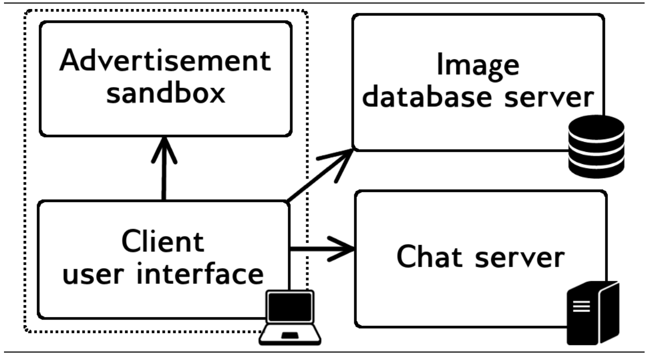

Figure 3. Architecture of our example application

2.3 A "real" application: state and third-party EDSLs

We now move on to a more complex application: a photo browsing site, where users may search for photos by description and discuss the photos in a real-time chat room. This example demonstrates the use of server-side state, as well as how to add a third-party EDSLs as just another node in the application. To monetize our hard work, we also include advertisements in our application, demonstrating how code from a third party may be safely included in an application using our language.

Fig. 3 shows the architecture of our example application. Note that while this application only consists of a client talking to multiple servers, our language allows servers to call each other as well, as discussed in Sect. 2.1.

Integrating with exotically typed third-party EDSLs To allow users of our application to search for images, we use a relational database to store pairs of image identifiers and their corresponding full-text descriptions. For this purpose, we use our language to incorporate the arrow-based *Opaleye* (Ellis 2014) SQL EDSL as a node in our application. The use of Opaleye in this example is entirely arbitrary. As previously discussed in Sect. 2.1, any EDSL, third-party or in-house, may be used as a node in our language.

Opaleye is what we call an *exotically typed* language. That is, the return type of a computation *within* the Opaleye EDSL, is not the same as the type of the value produced by actually *running* the query, reflecting an impedance mismatch between the Opaleye and Haskell type universes. This impedance mismatch is not specific to Opaleye or even to relational database EDSLs in general, but is common among deeply embedded EDSLs: the *Nikola* (Mainland and Morrisett 2010) EDSL for GPU programming, and the *Aplite* (Ekblad 2016) and *Sunroof* (Bracker and Gill 2014) EDSLs for web development also share this trait, just to name a few.

Our language lets us seamlessly integrate such EDSLs using the *Mapping* type class, first seen in Fig. 1. This type class allows us to define on a node by node basis, for each type *t* in that node's type universe, a corresponding type *Hask t* in the type universe of the host Haskell program. To enable translating values returned from such nodes, *Mapping* also lets us define a function *invoke*, which describes how computations on this particular node should be executed.

Opaleye requires us to first define the structure of any tables we want to use. We define a table consisting of two fields: an image's *id* and its *description*.

```
photoTable :: Table (Column PGInt4, Column PGText)
                    (Column PGInt4, Column PGText)
photoTable = Table "photos"
  (p2 (required "id", required "description"))
```

Then, we can use Opaleye's *Query* arrow to create a function which lets us perform a remote full-text search in our image database.

```
findImage :: RemotePtr
  (String → Query (Column PGInt4))
findImage = static (remote $ λkey → proc() → do
    (id, desc) ← queryTable photoTable -< ()
    restrict -< desc `like` pgString key
    returnA -< id)
```

Note the return type of this remote function. Our EDSL has no built-in idea of how to deal with values of type *Column PGInt4*, but we still want to be able to import this function.

For our image search database, we need to map *Column PGInt4* to *[Int]*, like in Opaleye proper, and to execute queries over the appropriate database file. After adding this *Mapping* instance, all we need to create a fully functional database node is the customary *Node* instance.

```
instance Mapping Query (Column PGInt4) where
  type Hask Query (Column PGInt4) = [Int]
  invoke _ q = liftIO $ do
    withConnection "images.sqlite" $ λc →
      runQuery c q
instance Node Query where
  endpoint _ = remoteNode "db.example.com" 24601
```

Stateful servers To support type-safe handling of server-side state, we introduce two new members of the *Node* type class: an associated type *Env*, and a method *init*. *Env* specifies the *environment type* of a node, and defaults to () – denoting an empty environment – unless explicitly given. The method *init* is executed when a node is started, and allows programmers to perform any initialization, before creating and returning the node's environment. From here on, the environment is *immutable*. An immutable environment may contain any kind of mutable synchronization primitive, such as *IORef*s or *MVar*s. By only providing a well-defined method to access any such constructs, we let the developer choose which kind of synchronization fits their application best, instead of forcing some arbitrary storage upon them.

In our application, we need two items in our environment. A mutable reference to a list of strings, representing all lines sent to chat so far, and a list of pairs of user names and password hashes, as we only want to allow users with proper credentials to send messages. In our *init* function, we initialize this environment to contain no chat lines as of yet, and to have a single user, *john*, with a spectacularly bad password. As usual, we also set up an endpoint at which the chat server can be reached.

```
postMessage :: RemotePtr
  (String → String → String → ChatSrv ())
postMessage = static (remote $ λuser pass msg → do
    (ref, creds) ← ask
    case lookup user creds of
      Just db_hash | db_hash == hash pass → do
        liftIO . atomicModifyIORef' ref $ λms →
          ((user ++ ": " ++ msg):ms, ())
      _ → throw (AuthError "bad login"))

checkMessages :: RemotePtr (ChatSrv [String])
checkMessages =
  static (remote $ ask >>= liftIO . readIORef . fst)
```

Figure 4. Implementation of the chat server

```
type State = (IORef [String], [(String, Hash)])
type ChatSrv = ReaderT State Server

instance Node ChatSrv where
  type Env ChatSrv = State
  init _ = liftIO $ do
    msgs ← newIORef []
    return (msgs, [("john", hash "password1")])
  endpoint _ = remoteNode "chat.example.com" 24601
```

Now, we are ready to define the two operations supported by the chat server: *postMessage* and *checkMessages*. *postMessage* takes a user name, a password and a message. It posts the given message to the chat, prefixed with the sender's name, if the sender has the appropriate credentials. *checkMessages* simply fetches all messages sent to the chat server so far, assuming the user has the requisite credentials. Fig. 4 gives the implementation of the chat server.

It should be noted that our implementation of *checkMessages* is not optimal: instead of polling for messages to download, a proper implementation of this function would block waiting for new messages to arrive. We have opted to use a "dumb" implementation here, to keep our example application as simple as possible.

Tying it together: the client As we use the Haste compiler to compile our client nodes, it is natural that we also use Haste's GUI library for our user interface. In keeping with the theme of using domain-specific languages wherever possible, the *structure* of the user interface is defined in an external HTML file, with only its *behavior* given here. Fig. 5 gives the implementation of the client.

The code in Fig. 5 starts by fetching the widgets associated with the HTML identifiers *search*, *img*, etc. On lines 9 to 17, an event handler is set on the *search* input, to search the image database for an image matching the description given in the input field. On lines 19 to 24, an event handler is set on the text box where chat messages are typed, to send a message when the user hits return while the input if focused. Finally, we set up a timer to poll the server for new messages once every second. As previously noted, the polling model is suboptimal and easily replaced by a more sophisticated solution, but was chosen for our example in the interest of brevity.

Late design decisions: moving code between nodes One advantage of our programming model is the ease of moving code between similar nodes. Moving code between two completely different nodes

```
1  main = runApp
2      [ start (Proxy :: Proxy ChatSrv)
3      , start (Proxy :: Proxy Query)
4      ] $ do
5    withElems
6      [ "search", "img", "box"
7      , "chat", "user", "pass"
8      ] $ λ[search, img, box, chat, user, pass] → do
9        search `onEvent` KeyUp $ λkey → do
10         when (key == keyReturn) $ do
11           term ← getProp search "value"
12           imgs ← dispatch findImage term
13           case imgs of
14             (i:is) →
15               set img ["src" =: (show i++".jpg")]
16             _      →
17               alert "No matches for that term."
18
19       chat `onEvent` KeyUp $ λkey → do
20         when (key == keyReturn) $ do
21           [u, p, m] ← mapM (flip getProp "value")
22                            [user, pass, chat]
23           setProp chat "value" ""
24           dispatch postMessage u p m
25
26     setTimer (Repeat 1000) $ do
27       msgs ← dispatch checkMessages cycle
28       setProp box "value" (unlines msgs)
```

Figure 5. Combined image search and chat client

– say, the database and the chat server – would be hard due to their differing programming models, but moving code between semantically compatible nodes is as easy as refactoring any non-distributed application. For instance, to protect the confidentiality of the user's password from the possibly untrusted server operator in the chat server example, one would simply need to move the call to *hash* from the *postMessage* remote function to the client. Similarly, user authentication could be offloaded to a dedicated server simply by adding a new function of the appropriate type, with the appropriate *Node* instance. In this way, developers can easily experiment with different architectures and implementations.

2.4 Sandboxing third party code

Recall from the discussion in Sect. 1, that web applications must often assume *parts of themselves* to be potentially malicious. To combat this problem, web browsers provide a mechanism called *sandboxed iframes*: a compartment in which sections of code can be executed, without being able to inspect or influence the rest of the application. Unfortunately, this construct is relatively clunky. Code executing in a sandboxed iframe can only communicate with the larger application through explicit message passing. As the JavaScript execution model is explicitly non-concurrent, both the sandboxed code and the application at large effectively need to be written in continuation-passing style. Additionally, glue code needs to be written to allow third party libraries – which usually follow a standard imperative programming model and definitely don't

```
data AdRotation
type MySbx = Sandbox AdRotation

instance Node MySbx where
  endpoint = localNode
  type Allowed MySbx a = a ~ Client
  init = dependOn ["http://example.com/ads.js"]

randomAd :: RemotePtr (MySbx URL)
randomAd = static (remote $ liftIO . ffi "getAdvert")

main = runApp [start (Proxy :: Proxy MySbx), ...]
  ...
  setTimer (Repeat (10*60*1000)) $ do
    adURL ← dispatch randomAd
    withElem "ad_banner" $ λbanner_img → do
      setProp banner_img "src" adURL
```

Figure 6. Adding sandboxed ads to the chat client

```
instance Node Server where
  endpoint _ = remoteNode "1.example.com" 24601

balance :: MonadIO m ⇒ [Endpoint] → m Endpoint
balance endpoints = do
  i ← liftIO $ randomRIO (0, length endpoints-1)
  return (endpoints !! i)

work :: RemotePtr (String → Server String)
work = static (remote performHeavyWork)

main = runApp [start (Proxy :: Proxy Server)] $ do
  ep ← balance
    [ remoteNode (show n ++ ".example.com") 24601
    | n ← [1..5] ]
  result ← dispatchTo ep work "[a massive data set]"
  presentResult result
```

Figure 7. Load balancing using dynamic dispatch

expect to deal with message passing – to execute in a sandbox, and to allow the larger application to create and communicate with the sandbox as well. For this reason, sandboxing is relatively unusual in conventional web applications, and is often limited to encapsulating relatively large, monolithic components.

Our language improves upon this situation by allowing a sandboxed iframe to be integrated into an application as just another node. Similar to how programs are split at compile-time, to produce one web client and any number of native servers, the *web client itself* is "split" at *run-time* to produce one "true" client, as well as multiple "server" nodes, each running in its own sandboxed iframe. The web browser's built-in sandbox implementation ensures that each sandbox is unable to communicate with the rest of the application except by responding to requests from the client. This lets us isolate not only external third-party code, but *any* piece of code, which enables developers to move more code from the trusted code base of the main application into the untrusted and locked-down environment provided by the sandbox. Like other nodes, sandboxes are identified by their type. Once created, a sandbox persists for the duration of the application, allowing sandboxed code to keep its state between invocations.

Sandboxed nodes are *local* to the client, as importing JavaScript client code onto a random server does not make much sense. For this reason, it should only be callable by the client. This is easily accomplished by setting the *Allowed* associated type class constraint of the *Node* class to only match the specific node *Client*.

In Fig. 6, we augment the chat application from section 2.3 with an advertisement, which is updated every 10 minutes by code from a third-party ad service. Since the ad rotation code is executed in a unique sandbox, parameterized over the *AdRotation* type, it can't interfere with the rest of the application or code running in other sandboxes, ensuring the integrity of the user's passwords, messages and image browsing habits.

2.5 Load balancing and dynamic dispatch

While having a single node type correspond to a single server has several merits, such as centralizing endpoint configuration and reducing communication boilerplate, there are several use cases where this model is a poor fit.

Often, a single physical server per logical node is not enough to meet the computational needs of the application, requiring tasks to be *load balanced* across multiple servers. This requires the ability to deploy several identical nodes of the same type, randomly choosing which node to call at run-time.

For interactive or bandwidth-intensive applications, it is often advisable to station servers at strategic locations throughout the world, and serve content from the server closest to each user, as is commonly done by content distribution networks. Like load balancing, this requires several identical nodes to be deployed, but now the call routing is no longer random but informed by external factors.

Certain services, such as in-home media streaming solutions, run on a user's local network, but registers with a central server from which the user can find the service without having to know its exact address on their home network. Implementing such an application in our language would involve fetching the endpoint for a node at run-time, via request to *another* node.

To accommodate these use cases, we provide a companion to the *dispatch* function, which takes the endpoint as an extra argument in addition to the the function to be dispatched: *dispatchTo*. When using this function, the provided endpoint *overrides* the endpoint defined by the node's *Node* instance. Fig. 7 shows an example implementation of load balancing using the *dispatchTo* construct. A "heavy" task is distributed randomly across five different servers, located at *n*.example.com, all running an instance each of the *Server* node.

3 Implementation

This section describes the implementation of our language, abstracting over some implementation details: the minutiae of network communication and event handling. The interested reader may refer to our full implementation, freely available from our website[1].

[1]https://haste-lang.org

```
class Serialize a where
  toJSON   :: a → JSON
  fromJSON :: JSON → Maybe a
class MonadIO m ⇒ MonadClient m where
  call :: Endpoint → StaticKey → [JSON] → m JSON
```

Figure 8. Assumed functions

3.1 Prerequisites and assumptions

Our implementation targets the de facto standard GHC Haskell compiler, to produce server-side binaries, and the Haste (Ekblad 2015) compiler to produce browser-based client code. Haste is a GHC-based, JavaScript-targeting Haskell compiler, which mainly differs from GHC Haskell in that concurrency is simulated within the previously mentioned *CIO* monad, as it targets non-concurrent JavaScript implementations. Our language is implementable on any recent GHC-based Haskell compiler, but relies crucially on several language extensions.

- Our language relies on *static pointers* (Epstein et al. 2011) to allow users to export remote functions.
- *Scoped type variables* and *type families*, both open and closed (Chakravarty et al. 2005; Eisenberg et al. 2014), are used extensively in our implementation to configure nodes and to implement remote function dispatch.
- The *CPP* extension, enabling the use of the standard C preprocessor, is used to implement program splitting into client and server programs.
- *Flexible instances*, *overlapping instances*, *flexible contexts*, *constraint kinds* and *multi-parameter type classes* are all used to ease Haskell's normal constraints on type class instances and constraints.
- *Default signatures* are used for certain methods and types of the *Node* type class, to reduce the amount of boilerplate necessary for common use cases.

Our implementation uses JSON as its data serialization format. To abstract over the details of data serialization, we assume the existence of a type class *Serialize*, which allows the conversion of values to and from a JSON-encoded string.

We also assume the existence of a type class *MonadClient*, for each node specifying how a packet representing a remote call is sent from one networked machine to another, to abstract over the details of network communication. Instances of this type class are also responsible for converting network errors and remote exceptions sent in reply to requests into errors appropriate for the instance. A listing of assumed functions is given in Fig. 8.

3.2 Exporting remote functions

In order to make a remote function available to clients, it must first be converted to a *remote import*. A remote import consists of a static pointer to a function from an environment for the node on which the import is intended to execute and a list of serialized arguments, to a serialized return value in the *CIO* monad previously discussed in Sect. 2:

```
newtype Import f =
   Import (Env (Aff f) → [JSON] → CIO JSON)
```

In our implementation, we need to refer to the node on which a function is intended to execute, the function's *affinity*, and the value resulting from fully applying the function and executing the resulting computation on its affinity node, its *result*. We define two closed type families to extract this information.

```
type family Aff f :: ∗ → ∗ where
  Aff (a → b) = Aff b
  Aff (n a)    = n
type family Res f :: ∗ where
  Res (a → b) = Res b
  Res (n a)    = a
```

We then use these two type families to define a type class *Remote* with a single method *toRemote*, to transform any exportable function into another function from a list of serialized arguments to a *CIO* computation returning a *serializable* value. The conversion is completed by the *remote* function, which constitutes the user interface for exporting functions as discussed in Sect. 2. An exportable function is any function where all of its arguments are serializable, and where the *mapping* of its result – the host Haskell type corresponding to the function's possibly domain-specific result type – is also serializable. The *Export* constraint captures this concept more precisely:

```
type Export f =
  ( Remote (Aff f) f
  , Serialize (Hask (Aff f) (Res f)))
```

Fig. 9 gives the implementation of the *Remote* type class and *remote* function. The *toRemote* function recurses over the argument types of the function to be exported, in each step applying it to the next in a list of serialized values provided by the caller. Once the exported function has no more arguments, we have reached our base case. At this point, we simply run the fully applied function using the *invoke* function. As discussed in Sect. 2, *invoke* can be seen of as a node's "runner function" – a function that executes an EDSL computation, returning some result back to the host language – and is given by each node's instance of the *Mapping* type class. This basic method used to transform one polymorphic function into another is well known, and is used by, among others Ekblad (2016), to transform function written in the Aplite EDSL into functions callable from the Haskell host – a use case quite similar to ours.

3.3 Dispatching remote calls

The remote import conversion is a server-side process, which converts eligible functions into remote imports *before* exporting them as static pointers. Similarly, there is some conversion needed on the side of the caller before those static pointers can be used to make remote calls. This conversion is in some sense the inverse of the remote import conversion.

Again, we recurse over the arguments of the remote function, this time using the *Remotable* type class, with its single *fromRemote* method. This time, however, we do not gradually apply the function, but instead we gradually *build up* the function that gathers its arguments, in serialized form, in a list. Once we reach the base case – again, when the remote function has no arguments left to process – we dispatch the remote import with the argument list we built up during the recursion, and wait for the server's reply to arrive. Fig. 10 gives the complete implementation of remote function dispatch.

The user-facing interface to this machinery is the *dispatch* function. While its implementation is simple, only consisting of a single

```
class (Aff f ~ n) ⇒ Remote n f where
  toRemote :: f → Env n → [JSON]
           → CIO (Hask n (Res f))

instance (Serialize a, Remote n b) ⇒
         Remote n (a → b) where
  toRemote f env (x:xs) | Just x' ← fromJSON x =
      toRemote env (f x') xs
  toRemote _ _ _ =
      throw CommunicationError

instance ( Aff (n a) ~ n
         , Res (n a) ~ a
         , Mapping n a) ⇒
         Remote n (n a) where
  toRemote f env _ = invoke env f

remote :: ∀f. Export f ⇒ f → Import f
remote f = Import $ λenv xs →
  let c = toRemote f env xs :: Hask (Aff f) (Res f))
  return (toJSON c)
```

Figure 9. Transforming functions into remote imports

call to *fromRemote*, its type is more interesting; more specifically, the *Dispatch* constraint. Some of the sub-constraints of *Dispatch* are expected: that remote functions must be actually remotable, and that their return types must have a mapping to equivalent host Haskell types – recall the discussion on exotically typed nodes in Sect. 2. However, the constraint that the type *HaskF (Aff f') f* must be equivalent to *f* is, perhaps, slightly opaque.

The *HaskF* type family extends the concept of host Haskell equivalent types to function types: if *Hask n a* denotes the equivalent host Haskell type of domain-specific type *a* for some node *n*, then *HaskF n f* denotes the equivalent host Haskell type of any function $f : a \rightarrow \ldots \rightarrow n\ b$ for some domain-specific type *b* and some node *n*. Note that the *argument types* of *f* are here assumed *not* to be domain-specific. The reason for this is that most EDSLs are easily able to accommodate conversion from host values to domain-specific values within the language itself, whereas conversion of return values are entirely dependent on an EDSL's runner function.

3.4 Servers and program splitting

The server-side implementation of remote calls is comparatively simple. For each node, its *init* method is invoked once to perform any setup necessary and to obtain its environment. Then, a thread is started to listen for incoming connections on the port specified by the node's endpoint. When a remote call arrives at a node, the node attempts to de-serialize it. If it could not be de-serialized, or if the static pointer denoting the call's entry point is invalid or does not match its accompanying arguments, an error is reported back to the caller where it is raised as an exception. This may happen if, for instance, there is a version mismatch between the deployed client and server, or if the client program has been modified by a malicious actor. Otherwise, the static pointer to the remote import is de-referenced to obtain the computation to be performed. The computation so obtained is invoked with the node's environment

```
class Remotable a where
  fromRemote :: Endpoint → StaticKey → [JSON] → a

instance (Serialize a, Remotable b) ⇒
         Remotable (a → b) where
  fromRemote e k xs x = fromRemote e k (toJSON x : xs)

instance (MonadClient from, Serialize a) ⇒
         Remotable (from a) where
  fromRemote e k xs = do
    res ← fromJSON <$> call e k (reverse xs)
    case res of
      Just r → return r
      _      → throw NetworkError

type family HaskF from a where
  HaskF from (a → b) = (a → HaskF from b)
  HaskF from (m a)    = from (Hask m a)

type Dispatch f f' =
  ( Node (Aff f)
  , Allowed (Aff f) (Aff f')
  , HaskF (Aff f') f ~ f'
  , Remotable f')

dispatch :: ∀f f'. Dispatch f f'
         ⇒ RemotePtr f → f'
dispatch f = fromRemote ep (staticKey f) []
  where ep = endpoint (Proxy :: Proxy (Aff dom))

dispatchTo :: Dispatch f f'
           ⇒ Endpoint → RemotePtr f → f'
dispatchTo ep f = fromRemote ep (staticKey f) []
```

Figure 10. Dispatching remote functions

and the argument list received with the remote call. If no error is raised during execution of said computation, the result is sent back to the caller. If an error *is* raised during execution, the error is instead reported back to the caller.

More interesting, then, is the implementation of how nodes are *started*, as this is where programs are split to determine where each node should run: on a native server, in a sandbox, on the client, or not at all? This splitting happens in two steps. First, at compile-time, native nodes are separated from nodes intended to run in a browser. This is accomplished by subtle differences in the implementations of the *start* and *runApp* function, depending on which compiler was used to build our language. If the program was built using the web-targeting compiler, the *Client* computation given to *runApp* is executed, but the *start* computations for any remote nodes are not. Conversely, when built using a native-targeting compiler, remote nodes *are* started, while the main client computation isn't.

The second step happens at run-time, and separates the client computation from any other client-side nodes – that is, the nodes running in sandboxes. The principle is the same as in the web/-native situation, but instead of splitting programs based on their compiler environment, programs are split based on their *browser* environment. The JavaScript program produced by our web-targeting compiler is executed both in the client browser context and in any sandboxes.

If the program detects that it is not running in a sandbox, it proceeds to create an iframe sandbox for each sandbox node. The sandboxes are then set up to execute *the same* program, after which the program – still outside the sandboxes – proceeds to execute the *Client* computation passed to *runApp*. If the program instead detects that it is indeed inside a sandbox, it sets up an event handler to wait for requests from the main client node and then goes dormant until a request arrives.

4 Discussion and related work

4.1 Our language as a collection of remote monads

It is worth noting that nodes in our language are almost, but not quite, instances of the *remote monad* design pattern identified by Gill et al. (2015). We say not quite, because in the remote monad design pattern computations are composed on the client and sent off to a server for execution. In light of the web-specific issues highlighted in Sect. 1, this is not desirable in a language targeting web applications. Consider the following function.

```
safelyLaunchMissiles :: Credentials → Server ()
safelyLaunchMissiles cred = do
  can_launch_missiles ← isPresident cred
  when can_launch_missiles launchMissiles
```

If this computation is exposed to the client as an atomic unit, there is no problem: if anyone but the president attempts to fire missiles, the permission check fails and the missiles stay where they are. However, if this computation was assembled on the client and only then dispatched to the server, a malicious user might simply *remove* the permission check completely! For this reason, we do not allow clients to arbitrarily compose computations before executing them remotely, but instead only allow computations exported via static pointers to be remotely executed.

4.2 Relation to microservice-based architectures

Microservices is another, increasingly popular, methodology for writing distributed applications, which takes a radically different approach from our language. Applications are written explicitly as a collection of small components communicating over some network protocol or message bus. Even components which may traditionally be seen as monolithic programs are usually split up, to achieve the highest possible granularity. While this approach achieves excellent decoupling, it offers very few, if any, safety guarantees, as each component is considered to be a truly independent program. It does, however, offer good reliability properties, as components are split into separate processes, possibly running on different machines.

Our language sacrifices a degree of decoupling in exchange for type safety. However, this does not mean that each application must be deployed as a monolith. As long as the type signatures of the entry points exported by a node in our language do not change and all remote imports are declared on the top level, said node – and any nodes that call it – may be independently updated and deployed. Leveraging this property and compiling each node into a separate binary, as described in Sect. 2.1, an application written in our language can achieve the same degree of reliability as an equivalent microservices-based application, but not the same level of decoupling.

4.3 Simplicity and flexibility

In the design of our language, we have attempted to strike a balance between flexibility and simplicity. Often this tradeoff has taken the shape of a more flexible, but more complex, behavior for the general case, with sensible defaults for common special cases. The *Node* type class, first introduced in Sect. 2, and the accompanying *Mapping* class, is perhaps the prime example of this design philosophy. While the combination admits considerable flexibility – crucial to our support for exotically typed EDSLs – type class defaults reduce the amount of complexity required to implement simpler applications to a significant degree.

When a more complex, possibly desirable behavior can be implemented on top of a simpler one with relatively little hassle, we have opted for the less complex option. *Push notifications* are an instance of this design choice. Recall that our language is client-centric, and does not allow calls to be made to the *Client* node. At first glance, this would seem to make push notifications impossible. However, even though this mode of operation is not supported intrinsically by the programming model, it is easily implemented on top of our language using concurrency and *long polling* – a technique where a synchronous remote call blocks indefinitely, until the server is ready to provide a reply.

4.4 Related work

Multi-tiered programming models for web applications have been a popular research topic of late. The problems with JavaScript as an implementation language for complex applications are well known, as are the advantages of using a coherent client-server framework to eliminate ad hoc network protocols and error-prone boilerplate code. Most existing attacks on this problem assume a relatively simple problem description: a web application consists of a client communicating with a single server, both of which share the same programming model.

However, with modern web applications, this assumption often does not hold. Not only can the traditional "server" of a multi-tiered web application often be broken down into domain-specific "sub-servers" – a conventional server which stores large files and a database server which stores metadata is a very common case – but clients themselves in reality often talk to a multitude of servers apart from the canonical one. Functionality to, say, geographically locate users or look up the postal code of every town on earth, resides on some third party server, a credit card payment processor resides on a second, and advertisements are served from a whole network of them.

This section attempts to give an overview of the field of multi-tier web application languages, comparing our language to the current state of the art.

Eliom The *Eliom* (Radanne et al. 2016) OCaml dialect provides a programming model for client-server web applications in which components are allocated to either the client or to a single server using explicit annotations. An interesting property of its programming model is that it is not only multi-tiered, but *multi-staged* as well: server-side code can manipulate client-side data as first class values, enabling a form of partial evaluation. Eliom also acknowledges the fact that the type universes of the client and the server may be different, with some types having different representations and some only existing on either the client or the server. Any type that is to be passed between client and server needs an explicit *converter*, to ensure proper representation on both sides of the rift. This concept is quite similar to the *mapping* concept used by our language to enable the use of exotically typed EDSLs.

Haste.App and AFAX *Haste.App* (Ekblad and Claessen 2014) uses a similar approach to our language, in which clients and servers are separated based on their type. Also like our language, Haste.App is embedded in Haskell, and uses compile-time program splitting to enable the client and the server to be produced by compiling the same application with two different compilers. *AFAX* (Petricek and Syme 2007) takes a very similar approach to Haste.App, but is embedded in F#. AFAX also requires certain minor changes to the type system of F# in order to be truly type-safe. Neither language supports more than a single server, which is assumed to have the same programming model as the client.

JavaScript dialects The *Conductance* application server (Cuthbertson 2014), and the *StratifiedJS* JavaScript extension upon which it is built equip the JavaScript language with an array of new features, including concurrency and a tierless programming model. This model allows the client to make remote procedure calls to the server, but also enables communication through the use of shared mutable variables. This model is less explicit about the client-server separation than the models we have seen so far, and is perhaps the most "JavaScript-like" of the lot. *Opa* (Bourgerie 2014), a competing framework, provides a similar programming model but dispenses with the remote procedure calls, instead basing all its communication on implicit data flows enabled by shared mutable variables. Opa also includes dedicated syntax for database queries, an important special case of domain-specific web application components.

Hop.js (Serrano and Prunet 2016) is a JavaScript dialect which compiles down to the *Hop* language, a Scheme dialect which in turn compiles down to JavaScript, for the client part, and a native binary for the server part. It supports Eliom-like multi-stage programming and inline HTML fragments – another special case of domain-specific components. Remote communication with Hop.js is centered around the concept of *services*: first-class objects representing remote machines. While the client may only invoke the services of the server from which it was served, that server may in turn invoke other servers belonging to the application.

Neither of these JavaScript dialects provide type safety or the ability to integrate multiple programming models in a single language.

Java-style Remote Method Invocation Our language is loosely related to Java's *Remote Method Invocation* (Downing 1998). This programming model allows programs to be automatically split into different servers, where each node can call methods on objects from another, as long as they possess a reference to such a remote object. RMI supports any number of nodes, but the relatively coarse-grained type system of Java forces all servers into the same programming model. The boundaries between nodes are also relatively floating, making it hard to determine where one node stops and another begins. RMI also does not provide a means for nodes to restrict which nodes may make calls to them, making it impossible to implement, for instance, the sandboxing from Sect. 2.4 using RMI.

Stand-alone languages Additionally, there are several stand-alone domain-specific languages for multi-tier web applications; *Links* (Cooper et al. 2007), *Ur/Web* (Chlipala 2015) and *ML5* (Vii et al. 2007) being the more prominent ones. A property common to these languages are their focus on the three conventional tiers of web applications: client, server, and database. Consequently, they include some form of support for all three tiers, including type-safe communication between them. While this approach does acknowledge the need for a domain-specific language for web programming, little heed is paid to the possible benefits of also using domain-specific languages in the web application's *business domain*. Links and Ur/Web only support a single server, while ML5 supports multiple, homogeneous servers. From a practical standpoint, stand-alone languages are at a disadvantage compared to an embedded language or a dialect of an existing one. Not only do they miss out on the opportunity to leverage existing ecosystems in general, but supporting heterogeneous servers becomes both easier and more advantageous in the presence of a large number of pre-existing, third-party EDSLs and libraries.

5 Conclusions and future work

We have presented an embedded, domain-specific language for building web-targeting, distributed applications out of heterogeneous components. Our language supports seamless, type-safe integration of components written in a variety of *other* embedded, domain-specific languages, in addition to components written in plain Haskell. We improve upon the state of the art by supporting an arbitrary number of heterogeneous nodes, with automatic communication access control enforced by the type system, and by allowing seamless, boilerplate-free integration of exotically typed EDSLs.

We have demonstrated the flexibility of our language by giving sample implementations of several important web application building blocks, including load-balancing and isolation of untrusted third-party code. Our language enables developers to easily share and move code between nodes, allowing design decisions regarding application topology and configuration to be pushed later into the development process. While we have not yet made any rigorous evaluation of our language, it has been used to good effect in the implementation of an educational proof assistant, deployed in a course of over 200 students. Block et al. (2016) conducted an evaluation of a previous, less capable, version of our language for the use of online multiplayer board games, with generally positive results.

Future work Our requirement that each node of an application is able to run a full Haskell binary can be problematic in some application areas, such as the most low-powered spectrum of the Internet of Things domain. For these areas, we intend to extend our programming model by generating platform-specific C code, including scaffolding and communication code, for nodes intended to run on low-powered devices, enabling them to take part in the same distributed applications as their full Haskell counterparts.

Our language currently only supports relatively rudimentary handling of runtime errors and reliability failures: throw an exception to which the client may choose to react. Exploring how errors may be handled in smarter ways, such as the possibility of automatically repairing certain classes of errors, looks like a promising line of future work.

Acknowledgments

Many thanks to Koen Claessen, Simon Marlow and Michal Palka for their valuable insights, feedback and discussion.

References

Benjamin Block, Joel Gustafsson, Michael Milakovic, Mattias Nilsen, and André Samuelsson. 2016. Evaluating Haste.App: Haskell in a web setting. Effects of using a seamless, linear, client-centric programming model. (2016).

Quentin Bourgerie. 2014. The Opa framework. http://opalang.org/. (2014).

Jan Bracker and Andy Gill. 2014. Sunroof: A Monadic DSL for Generating JavaScript. In *Practical Aspects of Declarative Languages*. Vol. 8324. Springer International Publishing.

Manuel MT Chakravarty, Gabriele Keller, and Simon Peyton Jones. 2005. Associated type synonyms. In *ACM SIGPLAN Notices*, Vol. 40. ACM.

Adam Chlipala. 2015. Ur/Web: A simple model for programming the Web. In *ACM SIGPLAN Notices*, Vol. 50. ACM.

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web programming without tiers. In *Formal Methods for Components and Objects*. Springer.

Tim Cuthbertson. 2014. The Conductance application server. http://conductance.io/. (2014).

Troy Bryan Downing. 1998. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc.

Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM.

Anton Ekblad. 2015. *A Distributed Haskell for the Modern Web*. Licentiate Thesis. Chalmers Institute of Technology.

Anton Ekblad. 2016. High-performance client-side web applications through Haskell EDSLs. In *Proceedings of the 9th International Symposium on Haskell*. ACM.

Anton Ekblad and Koen Claessen. 2014. A Seamless, Client-centric Programming Model for Type Safe Web Applications. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. ACM.

Tom Ellis. 2014. Opaleye. https://github.com/tomjaguarpaw/haskell-opaleye. (2014).

Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones. 2011. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM Symposium on Haskell*. ACM.

Andy Gill, Neil Sculthorpe, Justin Dawson, Aleksander Eskilson, Andrew Farmer, Mark Grebe, Jeffrey Rosenbluth, Ryan Scott, and James Stanton. 2015. The remote monad design pattern. In *ACM SIGPLAN Notices*, Vol. 50. ACM.

Máté Karácsony and Koen Claessen. 2016. Using fusion to enable late design decisions for pipelined computations. In *Proceedings of the 5th International Workshop on Functional High-Performance Computing*. ACM.

Geoffrey Mainland and Greg Morrisett. 2010. Nikola: embedding compiled GPU functions in Haskell. *ACM Sigplan Notices* 45 (2010).

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

Gabriel Radanne, Jérôme Vouillon, Vincent Balat, and Vasilis Papavasileiou. 2016. ELIOM: tierless Web programming from the ground up. (2016).

Manuel Serrano and Vincent Prunet. 2016. A glimpse of Hopjs. In *International Conference on Functional Programming (ICFP)*.

Tom Murphy Vii, Karl Crary, and Robert Harper. 2007. Type-safe distributed programming with ML5. In *International Symposium on Trustworthy Global Computing*. Springer.