Abstracting Network Calls Away From Client-Server Architectures

Izaak Weiss
The University of Colorado
izaak.weiss@colorado.edu

Alexander Curtiss
The University of Colorado
alexander.curtiss@colorado.edu

Seth Hovestol
The University of Colorado
seth.hovestol@colorado.edu

ABSTRACT

This paper details the design and implementation of a new programming language called Marmalum. Marmalum demos the abstraction of network communication into function calls between functions that have been labeled backend and frontend, which is compiled into a webserver and a client–side JavaScript program. Our implementation language was Scala, and our target language is JavaScript.

Reference Format:
Izaak Weiss, Alexander Curtiss, and Seth Hovestol. 2017. Abstracting Network Calls Away From Client-Server Architectures.

1 INTRODUCTION

1.1 Background

In the last decade Javascript has taken on a whole new status as a target language for compilers. Almost every major programming language has an experimental Javascript backend. Major programming languages are written with Javascript as their primary backend, such as TypeScript and Elm. The Web is the most used platform in the world, beating out every single operating system. It makes sense every language is targeting it.

Despite this, writing complex web apps is still a pain using modern programming languages. This is primarily because most complex web apps require the program running in the browser (the frontend) to communicate to the program running on the webserver (the backend). This requires the programmer to write the frontend program and backend program separately, often in different programming languages, and communcate between them using network calls.

This greatly inflates the amount of code required for simple tasks, and these network calls lack the ease and convenience that modern programming languages offer such as type safety and compiler optimization. Every piece of communication between the frontend and the backend must be serialized into a stream of bytes, sent over the network, and then parsed on the recieving end. It also requires programmers to use asynchronous callbacks in any code that uses a network call.

CSCI 4555, December 2017, Boulder, Colorado USA

1.2 Motivation

Many novice and experienced programmers struggle with the sheer amount of boilerplate code that is required for client–server communication. Beyond having a large barrier to entry, client–server communication also adds unnecessary complexity and potential bugs.

Our language attempts to address the issues with network programming by abstracting away network calls and the client-server architecture. We find this abstraction to be conceptually analogous to garbage collection. Many modern programming languages implement a garbage collector to handle memory allocation without any programmer input whatsoever, allowing for optimizations to be made at at compile time and for allocation and deallocation to be handled automatically at runtime. We wanted to do the same for network calls.

Another inspiration for this work is the language Elm. Elm is a purely function language that compiles to Javascript and interfaces with the DOM in a purely declarative way. Elm's runtime then handles all of the minutea regarding altering the DOM when actions must be taken. We wanted to have the programmer simply declare which code needs to run on the frontend, which code needs to run on the backend, and let the network calls be automatically handled in the compiler.

1.3 Accomplishment

We were able to construct a proof-of-concept compiler for a simple language that allowed functions to be marked as running on the frontend or the backend without addressing how the two sides would communicate. We call this language Marmalum. Programs written in Marmalum are compiled into two files, one meant for the server and the other for the client.

Success was demonstrated by writing the code for a webpage that keeps a running counter of the number of times it's been loaded. The client requests the number using an AJAX request without need for any network calls in the source code.

1.4 Notation introduced

We introduce the concept of *tree cutting* to represent the splitting of one Marmalum program into two Javascript programs, one of which runs on the client and the other of which runs on the server. A *cut statement* occurs when code on one end calls a function on the other end, requiring the compiler to generate a network call.

2 MARMALUM

2.1 Syntax

The ability to express concepts in code facilitates how programmers develop their ideas. As a result we paid special attention to how

ideas are expressed to best make it so that unnessisary details were hidden from the developer.

To both aid developers and ourselves we set up the basic structure of the syntax to follow similar rules as C like languages, with terminal semicolons and braces to denote scope. The main asthetic departure is in the type annotations, which follow the variable rather than preceed it.

2.1.1 Drawing a distinction. The language requires sections of the code to run on either the frontend or the backend at the discretion of the programmer. To implement this, Marmalum includes an annotation for each top level function or variable definition that specifies which side it is defined on.

We looked into the alternative of removing the location annotations on functions but requiring both global and local variables to be annotated, then leaving it up to the compiler to determine whether any line of code should run on the frontend or the backend. This idea was discarded when we decided that this takes too much control from the developer.

2.1.2 Importing javascript. We decided that the proof-of-concept compiler would need access to many Javascript features that would not be part of Marmalum's standard library. To allow access to arbitrary Javascript features we allow for import statements that take a string of Javascript code and a Marmalum-style function declaration that is used to call into the imported code. The benefits and downsides of this approach are detailed in section 3.1.1.

2.2 Semantics

The semantics of Marmalum are very similar to those of Javascript, so we won't discuss the details of structures similar to their javascript counterparts.

2.2.1 Assignments. Our language distinguishes between creation of a variable and asignment to a variable. We have four locations in which new names can be bound. These locations are function definitions, global variable declarations, import statements, and local variable declarations.

Import statements bind a snippet of Javascript to a name, providing an extensible foreign function call syntax. This allows our programming language to integrate with existing Javascript codebases, as well as simplifying our process of creating a standard library. Import statements annotated with the frontend keyword must be written in browser compatable Javascript, and import statements annotated with the backend keyword must be written in Node.js.

Global functions are annotated with the frontend or the backend keyword, and the keyword indicates where the function's code will be run. In the standard library we introduce some functions that are location-agnostic. This functionality is capable of being provided as a first class language feature, although it has not been implemented yet in the Marmalum compiler.

Global variables are also annotated as either frontend or backend. Backend global variables maintain their state for the entirety of the server's lifespan, and are shared among all instances of the webpage. Frontend global variables are unique to the client's current browser session.

2.2.2 Network Calls. When a frontend function calls a backend function, the arguments passed to the function are sent to the backend via an AJAX call, where they are passed to the backend code that operates on those values. The backend code's return value is passed in the AJAX response to the frontend, where the code picks up where it left off just after the call was made.

3 COMPILER

3.1 Parsing and Abstract Syntax

The Marmalum compiler parses input using Scala's parser combinator library. The grammer is simple and similar to many existing languages, with the one new addition of location annotations. Despite containing both frontend and backend code the parser only generates a single AST, which isn't seperated until the cutting phase.

3.1.1 Import Statements. The Javascript code inside import statements is not parsed, but instead is treated as a single string and gets inserted into the compiled file verbatum. This approach is similar to many compiled languages that allow for inline C or Assembly code, but it has a number of drawbacks. The most notable downside is that the Javascript code is not typechecked and there is no guarentee that the function name and parameter types given to the import statement match the Javascript function it represents, or even that the Javascript code implements such a function. This could be avoided by expanding the standard library, but that is outside the scope of this project.

3.2 Cutting

The concept of cutting is introduced in order to transform the procedural control flow of our language into the asynchronous code that JavaScript requires of its network calls. To handle this we move all of the code after a cross-network function call into a callback that will execute asynchronously after the network call has completed. This process is fairly simple for a sequence of statements.

Figure 1: Cutting a simple sequence of statements

However, more complicated methods must be put in place for more complex structures. If a cut statement occurs inside a conditional statement, the code after the conditional must be executed regardless of whether the condition was met or not. Our method for cutting a statement from inside a conditional is below:

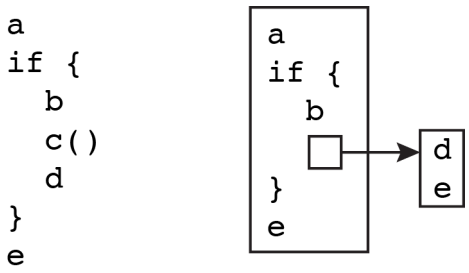

Figure 2: Cutting a section of code inside of a conditional

While loops are even more complex. We transform the while loop into a recursive structure, allowing for iteration without destroying the asynchronous nature of these function calls. However, note this adds an overhead, and it may be possible to optimize this sort of loop further. For instance, when this recursive loop ends, it must walk all the way up the call stack. However, this is a tail-recursive program, and it might be possible to implement this in a way that takes advantage of this. (JavaScript does not perform tail call elimination.)

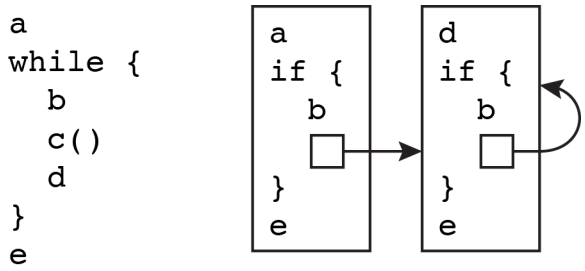

Figure 3: Cutting a section of code inside of a loop

We also discussed the difficulty of cutting nested structures. We investigated and we found that, ignoring the creation of two or more callbacks with identical semantics, cutting inner structures first was functionally identical to cutting outer structures first. However, we have not implemented cutting this advanced yet, so considerations due to runtime or implementation simplicity have not yet been explored.

3.3 Generation and Runtime

Most of Marmalum's code generation is fairly straightforward, as the target language is as high level as the source is. Basic expressions, assignments, and control flow are represented similarly in both languages and no major changes are needed.

To generate network calls and cut statements, the Marmalum compiler relies on a fairly substantial runtime library that is capable of creating and handling AJAX requests. To make function calls over the network, the frontend runtime implements a Javascript function that builds an AJAX request that stores the backend function name and parameters as JSON objects. The background runtime includes the code for a generic server which listens for any post requests, runs the requested function, and sends the return value back as more JSON encoded text.

To handle the asynchronous nature of the network calls, any code after a cut statement is stored as a callback that runs when the cut statement returns. Dealing with variables is made trivial by representing each callback as an inline closure, giving it access to any variables in the same scope.

4 CONCLUSION

In this paper we've presented a method for compiling code for a client-server architecture with abstracted network calls, as well as provide a basic implementation of these concepts. There is still a lot to be done before an implementation exists that satisfies the average developer's use cases, but we believe that such a solution could cut code length and development time by more than half for network-enabled projects.

4.1 Future Work

In the future, we want to explore a few different avenues of progress, which range from modifications to the runtime and code generation, to overhauls of the language itself.

4.1.1 WebSockets. Our current implementation uses AJAX requests to implement cross-network function calls. However, AJAX calls must be initiated from the browser, and thus make certain applications hard to write. For example, a chat application requires the backend to push information to the frontend whenever a new message comes in.

WebSockets are a protocol that allows the frontend and backend to communicate bidirectionally across the network. When taking this project forward, we intend to reimplement the runtime using WebSockets instead of AJAX requests.

4.1.2 Functional Idioms. Our current language is a very procedural and the style is reminiscent of C and JavaScript. One of the features we want to implement is first class functions, but we also are considering going further into the functional programming paradigm.

Functional programming might simplify the cutting algorithms detailed in Section 3.2. Cutting is difficult for state-based structures such as while loops, requiring us to transform the while loop into a recursive function. A language that was written in a functional programming style could avoid this complex transformation.

4.1.3 Meaning of backend calling frontend. Currently the language specification allows for the backend to call the frontend, but the compiler disallows it. With the change to websockets, the language specification will need to be updated with information on what it means for a backend function to call a frontend function. Specifically, how the single backend will handle calling functions on one or more frontend clients.

4.1.4 First class functions. First class functions do not currently exist in Marmalum. Functions cannot be encoded in the JSON data used to pass function arguments between the frontend and backend, so allowing first-class functions would require some amount of code duplication on both output files. We have yet to determine the feasibility of such an approach.

A EXAMPLE CODE

A.1 Source

```
frontend javascript
    "(id, str) => {
        document.getElementById(id).innerHTML = str
    }"
  as set(id: String, val: String): Void

frontend pageload(): Void {
  let x: Number = get_and_increment();
  set("counter", x);
}

backend counter: Number = 0;

backend get_and_increment(): Number {
  counter = counter + 1;
  print(counter);
  return counter;
}
```

A.2 Frontend Output

```
function print(x) {
  console.log(x);
}

require('http').createServer(
function(request, response) {
  const url = require('url')
  const fs = require('fs')
  const path = require('path')
  const baseDirectory = __dirname

  try {
    const requestUrl = url.parse(request.url)

    if (request.method == "POST" &&
        requestUrl.pathname == "/call") {
      var data = '';
      request.on('data', function(chunk) {
        data += chunk;
      });
      request.on('end', function() {
        data = JSON.parse(data);
        response.writeHead(200, {
          'Content-Type': 'application/json'
        });
        response.end(JSON.stringify(
          eval(data.func).apply(
            this, data.args
          )
        ));
      });
      return;
    }

    var fsPath = baseDirectory +
      path.normalize(requestUrl.pathname)
    if (fsPath.endsWith("/") ||
        fsPath.endsWith("\\")) {
      fsPath += "index.html"
    }

    var fileStream =
      fs.createReadStream(fsPath)
    fileStream.pipe(response)
    fileStream.on('open', function() {
      response.writeHead(200)
    })
    fileStream.on('error', function(e) {
      response.writeHead(404)
      response.end()
    })
  } catch (err) {
    response.writeHead(500)
    response.end()
    console.log(e.stack)
  }
}).listen(80);

var counter = 0.0;

function get_and_increment() {
  counter = (counter + 1.0);
  print(counter);
  return counter;
}
```

A.3 Backend Output

```
function print(x) {
  console.log(x);
}

function get_backend_function(f) {
  return (nf) => {
    xmlhttp = new XMLHttpRequest();
    xmlhttp.open("POST", "/call", true);
    xmlhttp.onreadystatechange = function() {
      if (xmlhttp.readyState == 4 &&
          xmlhttp.status == 200) {
        if (xmlhttp.responseText === "") {
          nf()
        } {
          nf(JSON.parse(xmlhttp.responseText));
        }
      }
    }
    data = {
      func: f,
      args:
        Array.prototype.slice.call(
          arguments, 1
        ),
    }
    xmlhttp.send(JSON.stringify(data));
  }
}
var set = (id, str) => {
  document.getElementById(id).innerHTML = str
}

function pageload() {
  function pageload0(asdf) {
    var x = asdf;
    set("counter", x);
  };
  return get_backend_function(
    'get_and_increment',
  )(pageload0);
}
```