CO600 Project: Jarlang
Technical Report

Chris Bailey `cb661@kent.ac.uk`
Andrew Johnson `apj8@kent.ac.uk`
Nick Laine `nl227@kent.ac.uk`

School of Computing
University of Kent
United Kingdom

Word Count: 6,335 (via texcount)

March 28, 2018

Abstract

This paper describes the development and operation of an Erlang to JavaScript compiler henceforth referred to as *Jarlang*.

The primary goal of Jarlang was originally to explore the feasibility of compiling Erlang source code into JavaScript such that people would be able to simply run it and demonstrate it on the open web. This is not only a convinient feature but also has applications for the educational use of Erlang, as well as simply improving existing documentation for Erlang with the use of real-time examples.

Being able to help create a unified stack for web development has also been a goal of the project, as such a concept is rather attractive to contemporary web developers.

Unlike similar projects such as *LuvvieScript* (Guthrie, 2014), we aimed not to support a *"subset of Erlang"* or create a new language *"based on Erlang"*, but instead wanted to focus on implementing as much of Erlang, and its standard library, runtime and behaviours as possible on the web.

1 Introduction

Jarlang is a compiler implemented in Erlang and JavaScript, whose goal is to compile valid Erlang module source code into valid JavaScript source code.

Jarlang thus provides several benefits to potential users such as the ability to execute, or demonstrate, Erlang code on the web. This could add to available Erlang documentation, as well as provide educators with another means of giving live code demonstrations. Jarlang also allows developers to unify their tech-stack which is a popular trend amongst contemporary web developers. Unifying tech-stacks results in less time spent context switching when switching from front-end programming and back-end programming, as well as making both front-end and back-end programming accessible to any competent Erlang developer.

Most of our code is written in Erlang, which is the same language as our compilation source. The main motivation behind this is to eventually bootstrap the Jarlang compiler (compiling the compiler). If we were to compile Jarlang's source code into JavaScript, we essentially have an Erlang compiler which can run on its own on the web, only furthering educational benefit as Erlang programmers therefore don't need to install an Erlang compiler themselves. This would also allow Erlang to act moreso as a first class citizen on the Web.

Due to time constraints however, the currently implementation of the Jarlang compiler hooks into the existing Erlang compiler and thus has a hard dependency on it. We utilise the Erlang compiler to parse, lex, validate, and optimise Erlang source code. After these steps, the Erlang compiler allows us to manipulate its intermediate language known as *Core Erlang*, which is where the Jarlang compiler performs most of its work.

In order to remove our dependency on the Erlang compiler, we would need to essentially reimplement it (write a parser & lexer for example, as well as AST generation) or leverage Jarlang to compile the Erlang compiler itself into JavaScript. This requires Jarlang to be more fully implemented than it currently is, but the goal can be feasibly reached in the long term.

We also intended to, and in many ways, have succeeded in implementing much of Erlang's functionality — the results of the Jarlang compiler successfully emulate Erlang style modules (including private and exported functions), Erlang style variadic functions, and more abmitiously, a concurrent actor-model system leveraging JavaScript's event loop.

At the time of writing, Jarlang can successfully compile and run not just simple test programs such as Factorial or Fibonnaci, but also concurrent programs such as a Heartbeat protocol client/server, OTP style server/state-machine/event-handlers and interop with JavaScript to the extent where we have succesfully leveraged Jarlang compiled Erlang code to write simple games and instant messaging applications.

2 Background

Some attempts to link Erlang to JavaScript have been made in the past. LuvvieScript (Guthrie, 2014) and Shen (Tonpa, 2014), both inactive since 2014, have taken Erlang as input and produced JavaScript; neither of these can be said to be compilers in their current state. LuvvieScript operates on a subset of Erlang, taking as input specifically written Erlang files always intended to be processed by LuvvieScript. On the other hand, Shen generates idiomatic JavaScript by trying to interpret the functionaility of Erlang code (as opposed to compiling it).

Although neither of these projects are what we tried to do, LuvvieScript did provide insight into the compilation process that we have adopted. A description of the LuvvieScript toolchain (Guthrie, 2017) provided the idea of parsing Core Erlang AST into JavaScript AST and using a third-party codegen module to process that into JavaScript.

3 Aims

The primary aim of *Jarlang* is to generate valid JavaScript from any error-free Erlang module source code which can then be integreated into any front-end codebase.

Secondary aims for *Jarlang* include the ability to leverage Erlang-style concurrency in what is more often than not a strictly single-threaded environment, as well as being able to *execute correctly* the generated JavaScript — this presumes the re-implementation of the Erlang standard library and Erlang runtime environment as the equivalent *Jarlang runtime environment*.

While fully implementing the *Jarlang runtime environment* is not the primary goal of this project, enough of the *Jarlang runtime environment* has been implemented so that we can successfully emulate things such as Erlang's datatypes, message passing and concurrency model — allowing us to run tests and write example applications as already introduced.

4 Project Results

While we lack a fully implemented Jarlang runtime environment, at the time of writing the runtime environment and Jarlang compiler are complete enough to run simple Erlang programs on the web as well as successfully (and quite easily) integrate into existing JavaScript codebases. This therefore allows JavaScript code to leverage the power that Erlang's simple actor-style concurrency brings to the table.

We have many test examples working ranging from simple 'Hello world!' type applications, to datatype benchmarking and Erlang-compatability testing. In terms of more sizable applications, we were able to compile Erlang assignments from our second year of study also. Interoperability is at a point where it is trivial for JavaScript to interact with compiled Erlang code and Erlang code can trivially call into JavaScript code.

The Jarlang runtime environment, as explained briefly above, is a large hand-written re-implementation of much of the Erlang standard library and Erlang runtime environment in JavaScript (currently sitting at around 5500 lines of code).

We intend to eventually use the Jarlang compiler to compile most of the standard Erlang library, so that it can be utilised in the runtime environment. However, much of it is implemented in C which unfortunately does need to be rewritten from scratch.

Notable features of the Jarlang runtime environment in its current state are:

- Erlang Datatypes - The Jarlang runtime environment successfully reimplements and emulates all of the Erlang datatypes we have deemed to be implementable on the web, and are as follows: *'Atoms', 'BitStrings', 'Floats (BigNum)', 'Integers (BigNum)', 'Lists', 'Maps', 'PIDs', 'Tuples','Unbounds', and 'Processes'*. We will go into more detail about these below.

- Erlang's Concurrency Model - The compiled output of the Jarlang

compiler intelligently wraps function calls into unique processes when needed. Each process has a mailbox and the ability to defer execution while awaiting messages to process and respond to. Processes can send messages to other processes and therefore trigger asynchronous behaviour via this mechanism. Furthermore, Processes live in a pool of active processes and are run asynchronously in JavaScript's event loop which seems to, at a high level, simulate the concurrency model used by Erlang.

- IO and Erlang module - The Erlang IO module allows processes to print to the JavaScript console with a *printf* style syntax. It closely emulates the same interface as the Erlang standard IO module and at a basic level is interoperable. The Erlang module is also implemented to the point where we can spawn processes, register processes in a global *Atom table*, perform type casting and more.

4.1 The Jarlang Runtime Environment

The Jarlang runtime environment is implemented in many separate JavaScript modules, which are processed and bound together by our NodeJS toolchain and tools such as *Gulp* (Gulp, 2018). The runtime is split into three distinct segments.

- Erlang Datatypes - Each Erlang datatype implemented as an ES6-spec JavaScript class in its own closure to keep track of private and public methods/properties.

- Standard Library Modules - Each standard library module is implemented as an ES5-spec JavaScript module which interally also uses a closure in order to keep track of exported/unexported functions.

- Runtime Specific Code - This exists enclosed within a single ES5-spec JavaScript module which creates the appropriate environment expected

of an Erlang-like environment such as setting up a process pool, global atom table and more.

4.1.1 Erlang Datatypes

Values of any given type in Erlang can be compared against one another with the standard arithmatic comparison operators '>', '<', '>=', '=<'. Other operators such as '=' could also be seen as doing a type-to-type comparison also. Due to this, instead of implementing multiple typechecking checks in each of the function calls for the given arithmatic comparison operators, we opted instead to build custom classes which would let us easily implement such behaviour.

In order to easily and painlessly implement this behaviour, all datatype classes inherit from a custom type we defined called *erlangDatatype* and therefore have several properties initialised by default such as *erlangDatatype.value*, *erlangDatatype.precedence* and auxillary functions intended to be overwritten such as *erlangDatatype.toString()*, and *erlangDatatype.match(N)*.

Following this, each datatype overwrites *erlangDatatype.precedence* in its own class with a number which determines how it gets compared to values of a different type. Atoms in Erlang are less than everything for example, and thus in the Jarlang runtime environment, Atoms have a precedence value of 1 whilst everything else has a precedence value greater than 1.

In more complex cases, precedence is not the only value tested to determine the comparison of two values. Data collections such as Lists, BitStrings and Maps all override their *erlangDatatype.match(N)* functionality so that they can correctly implement comparison matching as well as object equality checking which is then used for pattern matching.

Of course, each datatype then goes on to implement type-specific behaviour in an attempt to mimic type behaviour in Erlang. A high level outline of what each datatype does is given below:

- Atoms - Atoms are essentially just constant immutable values which are also singletons. When atoms are created, they register themselves

in the runtime system's global atom table which keeps track of every atom which has ever been initialized. The global atom table is also a place where you can store references to processes, and as such atoms in the Jarlang runtime system (just like in Erlang), double as 'global identifiers' to processes.

```
% Example of atoms in Erlang vs compiled output:
atom       : new Atom("atom");
true       : new Atom("true");
'This too' : new Atom("This too");
```

Figure 1: Demonstration of the Atom datatype

- Bitstrings - Bitstrings are contiguous arrangements of binary data in the Erlang world. We were originally not going to implement the Bitstring class because it's not something you'd commonly work with on the web; however, strings in Erlang are often represented as Binary strings and as such we needed to implement these to support such a common usecase. Bitstrings can be pattern matched against like Lists, Maps and Tuples but posess the unique property of being able to match byte patterns into variables allowing for simple implementation of binary format parsers. In the Jarlang runtime system, Bitstrings are implemented as arrays that store integers which are clamped (via modular arithmetic, as in Erlang) to values which can be expressed in eight bits. For values that use a size expression (an integer that specifies the length of a Bitstring segment) that's greater than eight, the value is expressed across as many Bitstring segments as necessary. The size expression for each segment is stored in a separate array.

```
% Example of Bitstrings in Erlang vs compiled output:
<<"Hello, world!">> : new BitString("Hello, world!");
```

Figure 2: Demonstration of the Bitstring datatype

- Floats/Ints - Numbers in Erlang have arbitrary precision and are implemented via the inclusion of the *bignumber.js* library. This is the only part of Jarlang runtime which has an external dependency. Whilst the classes Float and Int aren't entirely identical, they share most of their interface. These classes implement typical mathematical functions such as addition, subtraction and multiplication which is what gets used by the Jarlang runtime system whenever it performs mathematical calculations. As in Erlang, all operations that involve a float return a float, and all operations that involve only integers (sans division, if an integer is not divisible by another integer) return an integer.

```
% Example of atoms in Erlang vs compiled output:
12    : new Int("12");
103.4 : new Float("103.4");

% Operations in Erlang vs compiled output:
1 + 1         : new Int("1").add(new Int("1"));
isInteger(12) : Int.isInt(new Int("12"));
```

Figure 3: Demonstration of the Floats/Ints datatype

- Lists - Lists are implemented as immutable linked lists to allow us to have an efficient implementation of the *cons operator* which mimics Erlang's very own implementation. Each list contains a value and a *next* value. A list is said have a length of one when it has a value and points to an empty list. Erlang allows lists to be *improper* however, which means that lists can point to any other data-structure/primitive but does not promise that common list operations will work on these improper lists. Erlang strings are also denoted as lists — any list containing only integers that correspond to characters in the Latin-1 character set are considered and printed as strings by Jarlang as well as Erlang.

```
[1, 2, 3]      -> new List(1, 2, 3);
[1, 2 | [34]]  -> new List(1, 2).cons(new List(34));
[1 | improper] -> new List(1).cons(new Atom("improper"));
"hello!"       -> new List("hello!")
             (or) new List(103, 101, 108, 108, 111, 100);
```

Figure 4: Demonstration of the List datatype

- Maps - Maps were originally implemented as wrappers over JavaScript objects. Maps are currently implemented using two JavaScript arrays — one for keys (which can be non-serialized objects of any kind), and the other for values such that looking up the i^{th} value of the keys array returns the i^{th} value of the values array. This was implemented like this because it was the easiest way for us to implement the size comparison functions for Maps internally, however we realise now in retrospect that we ought to refactor this into a more efficient tree data-structure in the future should we revisit this datatype.

```
#{a=>1, b=>2} -> new Map([a, b], [1, 2]);
```

Figure 5: Demonstration of the Maps datatype

- PID - PIDs are process identifiers and are usually only created when new processes are spawned. PIDs act simply as an identifier and thus they don't have any special functionality outside of being keys to processes in the runtime system's global process table. PIDs are made up of three identifying features: the node where the PID exists (allowing for distributed processing), and two other IDs.

```
<12.9.12> -> new Pid(12, 9, 12);
        (or) new Pid([12, 9, 12]);

% Message sending
<0.1.0> ! 'hi' : new Pid(0, 1, 0)["!"](new Atom("hi"));
```

Figure 6: Demonstration of the PID datatype

- Processes - At a high level, these are agents which execute a given behaviour (encapsulating a function call) asynchronously. These are explained in depth in section 4.1.4.

```
// Processes look like this in JavaScript:
{
    ...
    messages: [],
    currentBehaviour: lambda,
    stack: []
    ...
}

// Process live in an array of processes, and we
// run them via:
setInterval(runProcesses, 0);
runProcesses = () => processes.map(process =>
    return process.currentBehaviour();
);
```

Figure 7: Demonstration of the Process datatype

- Tuples - Tuples behave like lists of fixed size. They don't have the

ability to *cons* with other tuples but are implemented in a similar way to lists for easier pattern-matching implementation.

```
{1, 2, 3} -> new Tuple(1, 2, 3);
```

Figure 8: Demonstration of the Tuple datatype

Because each datatype inherits properties such as *erlangDatatype.value*, these datatypes are often interacted with by querying the value stored in *erlangDatatype.value*. The only gap in our datatype abstraction is the lack of a *fun* datatype as we are compiling Erlang functions into JavaScript functions.

Any time a user makes a call to compiled Erlang code, they are in fact invoking the Jarlang runtime system. All Erlang related code is processed purely using these datatypes and as such there is an issue of JavaScript-Jarlang interoperability.

4.1.2 JavaScript - Jarlang Interoperability

Originally, there was a troublesome lack of interoperability between the 'outside JavaScript world' and our 'internal Jarlang world'.

Given an Erlang function which is called with the following: *fibbonaci:fibb(12)*, originally, a user of Jarlang would have had to execute *fibbonaci.fib(new Int(12))* in order to get it to work as they expected. This issue affected us as well while testing; more complex datatypes such as, in Erlang: *[1, 2, 3, 3]*, would have had to been written, in JavaScript, as *new List(new Int(1), new Int(2), new Tuple(new Int(3), new Int(3)))* which not only was much more verbose, but was simply not friendly to use.

We got around this by implementing functions which try their best to convert JavaScript types into their natural Erlang types where applicable. These functions are automatically wrapped around any potential user input such as around exported functions of any modules, as well as any potential output. This means that for all simple cases, the user can run *fibbonaci.fib(12)* and expect it to work just like the Erlang equivalent.

When converting JavaScript types to Erlang types, there the following types are mapped to eachother:

- Number ⇒ Int/Float (depending on value of given Number)
- String ⇒ List
- Array ⇒ List
- Uint8Array ⇒ BitString
- Object ⇒ Map

Any non-mapped datatypes will need to be manually created as before though, so functions which require an Atom will need to call said code with *new Atom(...)*.

When converting Erlang types into JavaScript, the converted types are:

- Atom ⇒ String
- BitString ⇒ Uint8Array
- List ⇒ String or Array, depending the values within the List
- Map ⇒ An Object in the form of { *keys: [...], values: [...]* }
- Int/Float ⇒ Number
- Pid ⇒ Not converted. Exposed as raw Erlang datatype.
- Process ⇒ Not converted. Exposed as raw Erlang datatype.
- Tuple ⇒ Array

The main downside to this automatic type conversion system is when dealing with anonymous functions on both the JavaScript side and the Erlang side. Since we're using JavaScript functions as the datatype to represent Erlang functions, we can pass them around and invoke them as normal first

class objects, however the issue is that it is impossible for the arguments going into these functions and coming out of these functions to be automatically converted.

Lastly, since Erlang function calls in the form *module:function(...)* are simply compiled to *module.function(...)*, Erlang code can call native browser APIs, as well as user defined functions by simply making a call to that code. Calling *window:function(...)* allows access to any globally accessible functions, though for ease of use, the Jarlang runtime will eventually include its own wrapper around much of the standard JavaScript APIs allowing even better interoperability.

4.1.3 Sequential Programming (or lack thereof)

As stated in the section above, the Jarlang runtime environment is the only environment in which compiled Erlang code is run. Because of this, we have a clear entrypoint and exitpoint *(Erlang → JavaScript function calls notwithstanding)* where we can perform some logic to improve the interoperability between the Jarlang runtime environment and the external JavaScript environment.

Another thing we do in this layer is instantiate new processes for any function calls the user makes. Whenever the user calls Erlang functions compiled through the Jarlang compiler, the runtime system intercepts these function calls and insteads wraps those function calls inside the *Process.behaviour* property of a new Process whose PID gets returned.

Because Processes are run asynchronously, interleaved among other units of work during the JavaScript event loop, this neccessitates that the returned PIDs — essentially the return value of any user-run Erlang code — share a similar interface and design to native *JavaScript Promises* such that they embody the idea of *"work that needs to be done"* or *"a promise of a return value in the future"*.

This means that when interfacing with Erlang code in JavaScript, the code needs to be structured to listen for return events in the future; in short,

Erlang code compiled by the Jarlang compiler cannot be run synchronously by a user.

Moreover, multiple processes spawned sequentially side-by-side do not guarentee any order of execution, thus one cannot rely on this mechanism to ensure sequential code.

This was a design decision we made relatively late on since originally the Jarlang compiler and Jarlang runtime environment produced only seqential Erlang code. The tradeoff for allowing sequential Erlang code to be generated was that, at that point in time, the Jarlang compiler and runtime environment did not support the compilation or running of Erlang processes.

Because all code, even completely sequential Erlang, is executed in an asynchronous manner — even typing commands in the Erlang shell execute in a process, as evidenced by the result of running the command *self()* — we wanted to mimic this despite the slight inconvinience of forcing such a paradigm shift on any potential users.

In line with our attempt to mimic Erlang's concurrency model however, we do make the guarentee that any *sequential-only Erlang code* executed within any process is entirely sequential. We go into this in detail in the following section where we discuss how our Processes actually work and how this enables us to utilise concurrent programming paradigms in JavaScript.

4.1.4 Concurrent Programming

As we have detailed above, all user calls into compiled Erlang code runs in the context of a process executing a *behaviour*. Processes are architected such that they resemble *JavaScript Promises* for ease of interoperability, but also act like traditional actor model agents.

On a high level, the BEAM virtual machine (responsible for running Erlang code) schedules processes based on how long processes have been running. It has the ability to pause and resume processes as and when it sees fit.

While it is not impossible for us to implement such a scheduler, it would

force us to perform heavy modifications during compilation to add timing code calls throughout Erlang modules. This would adversely impact readability and performance to a greater degree than we are willing.

Instead, we opted for an extremely high level simulation of how the actor model is implemented in Erlang. Erlang code is essentially strictly sequential in all but one case — if a function contains a *receive block*, the function essentially defers execution of the containing process until the *receive block* has passed. This is also where our guarentee of sequential order comes from as non-sequential Erlang code is compiled to sequential JavaScript.

The behaviour property of a given process is a reference to the function that process should perform in the next tick of work allocated to said process.

Processes also have an internal API which allows them to set the behaviour property to either the currently executing behaviour (repeating the current unit of work again in the next tick) or another function (execute a new behaviour in the next tick) With this simple API, we are able to implement basic actor-model concurrency.

For this to work, all processes also contain a inbox for messages to accumulate in. During *receive blocks*, we pattern match against these messages in order to determine what we will execute in the next tick. As far as concurrency is concerned, this is the only place in Erlang where concurrency can occur, and as such, our implementation of actor-style concurrency is compatible with basic Erlang examples.

One of the places where our implementation makes a notable difference to Erlang's implementation is when a given process never ends from a sequential standpoint. All sequential Erlang code is run sequentially and atomically and thus, an infinite loop would lock up the browser's JavaScript thread indefinitely. In Erlang, this is not the case as the BEAM virtual machine is still free to switch to another process.

As a more detailed example then, of how we are implementing concurrency as far as the Jarlang compiler & runtime environment is concerned:

1. Compile normal sequential Erlang code into normal sequential JavaScript.

If we reach a receieve block, split the function into three pieces such that we have a PRE-RECEIVE function, a POST-RECEIVE function and finally the receieve block itself.

2. Assemble these function segments as inline anonymous function calls, passing variables from one segment of the function to another by use of a closure — this allows us to bind variables from one function into another quite naturally.

3. Modify the receive blocks such that if no messages are matched, set the process behaviour to the same receive block allowing us to indefinitely defer processes until a message is matched.

4. When a function containing non-sequential Erlang is run, the process runs sequentially until the receive block is met (the PRE-RECEIVE block). Once the receive block is met, the process defers itself indefinitely such that the only thing it does each tick of worktime it gets given is check its message inbox.

5. When a message is matched, it sequentially runs the code coupled with the matched case before deferring again, but this time to the POST-RECEIVE function.

Via this process, messages can trigger deferred processes awaiting messages to continue execution, and can themselves await messages. With this basic behaviour we can implement basic OTP-like server functionality as well as state-machine like functionality.

One of the things yet to be implement is a mechanism to cancel a receive block after a period of time. This can be implemented via simply counting the time which has passed in between process ticks and exiting the receive block after so long but this has not been implemented at the time of writing.

It is also important to note that while for simple cases of concurrent behaviour we successfully mimic Erlang code, for more complex patterns containing nested receive blocks, the Jarlang compiler currently does not

produce the correct output as we cannot so naively simply split a function into segments. We do plan on leveraging JavaScript's async/await concurrency constructs which will be able to help us mimic this behaviour better in the future.

4.1.5 Distributed Programming

Another important facet of Erlang's concurrency model is that it is fairly trivial to expand across multiple different nodes to allow easy distributed programming.

Whilst at this point in time Jarlang doesn't provide any official mechanism to allow for such distributed programming — and likely never will, due to the following explanation — it would be trivial to implement a similar system in JavaScript to facilitate distributed programming.

Because the only facility processes have to trigger asynchronous behaviours is via message passing, and all messages are sent to either known PIDs in the global PID table or known Atoms in the global Atom table, one could easily implement a channel which listens for messages sent to *virtual PIDs/Atoms* belonging to distributed processes. This channel could then simply forward the message via any web mechanism capable of real time communication such as *WebRTC* or more commonly *WebSockets*.

Following this, a layer would need to be implement to route received distributed messages to their intended target but once this was done, any given process would be able to trivially and arbitrarily send and receive messages from distributed nodes.

4.1.6 Current Runtime Limitations

While as a whole, the Jarlang runtime environment works quite successfully in emulating the behaviour of running Erlang code at a high level, there are some drawbacks and limitations to what can be done with Jarlang at this point in time. While some of these can be handled by small or large

changes, some are limitations on our architectural approach and likely won't be resolved in the near future if at all.

They are listed as follows:

- Currently we provide no mechanism for processes to supervise or react to events of linked processes.
- Currently, scope details can be lost if receive blocks are heavily nested which breaks more complex concurrent Erlang behaviour.
- The *spawn/N* functions cannot currently spawn functions without relying on JavaScripts *eval* function. This is because we cannot access programmatically the contents of compiled Erlang modules as they are declared as a *const variable* and thus do not bind to the global scope. We chose not to simply use a *var variable* instead but to have Erlang modules register themselves with the runtime environment upon initialisation. This is currently a pending task.
- We currently do not and cannot garbage collect proceses. Upon ending execution they defer forever since we can never be sure there are no references to these processes either inside the runtime environment (which is implementable) or outside in the open JavaScript environment (which isn't implementable).

4.2 The Jarlang Compiler

We utilise the Erlang compiler to help us leverage some priliminary optimisations performed, and also to allow us to work in a much smaller language representation allowing us to ignore much syntactic sugar which exists in standard Erlang — this smaller language is Erlang's intermediate representation called *Core Erlang*.

One huge gain from utilising Core Erlang instead of standard Erlang is the ability to easily parse it and get an Abstract Syntax Tree out of it using standard Erlang tools. This allows us to not have to implement a parser or

a lexer for our compiler (though as stated, if we ever want to bootstrap the Jarlang compiler, we will likely need to change how this is handled).

Since the Erlang compiler also has to get to the stage where it is generated code output, we also know that any programs passed into the Jarlang compiler are valid Erlang programs. More details follow:

4.2.1 Between the Erlang Compiler and the Jarlang Compiler

As mentioned above, before the Jarlang compiler is able to do any work, the Erlang compiler has to more or less successfully compile Erlang source code files into Erlang's intermediate language — Core Erlang — which means we don't have to worry about certain classes of errors such as referencing undefined variables or throwing errors for invalid conditional logic or pattern match logic.

As an example, the Erlang function shown in Figure 9 will throw an runtime error if the input is anything other than the atom 'apple'. Because we have not manually defined a clause which does so for us, the Erlang compiler automatically generates on as you can see in Figure 10. This means that as long as we've programmed the Jarlang compiler to correctly parse and generate an equivalent JavaScript abstract syntax tree from our Core Erlang, everything will work as intended (see the Jarlang compiler output for this error at Figure 11).

The generated Core Erlang also removes much of the syntactic sugar from Erlang code; pattern-matching appears only in case statements or receive statements, all case statements include catch-all clauses, all function arities and modules are stated for us without the need for inference. This allows for a much more streamlined compilation to JavaScript though the largest issue of Core Erlang is that it is largely undocumented – as such, a large amount of project time was spent reverse engineering the behaviour of Core Erlang language features.

After this, we load the generated Core Erlang AST into the Jarlang compiler and begin the compilation process to JavaScript. We do this by recur-

sively descending through each AST node, performing some analyses on the current AST node (and parental/sibling/descendent nodes if needed) and output an equivalent JavaScript AST node.

4.2.2 The JavaScript AST

The JavaScript AST is very well defined and many tools exist to allow us to inspect the JavaScript AST of any given JavaScript program, as well as code generation tools to turn a valid JavaScript AST into JavaScript code and as such we haven't had much of an issue writing an equivalent JavaScript AST for a given CoreErlang AST node.

The main issue with generating the JavaScript AST is that the JavaScript AST is expressed as a deeply nested JSON object which has no built-in representation in Erlang. LuvvieScript (Guthrie, 2014) approaches this issue by hand-encoding these JSON objects as nested binary strings which we think is too restrictive. Because we were not attracted to the idea of hand-encoding JSON strings whenever we needed to generate a particular JavaScript AST node, we wrote two modules: *estree.erl* and *json.erl* to help with JavaScript AST generation.

4.2.3 estree.erl and json.erl

estree.erl is an implementation of the JavaScript AST interface as defined by the Mozilla Foundation (2017) as well as reverse engineering the AST node structure of newer as of yet not-officially-defined AST nodes.

estree.erl went through many changes during the course of the project as our needs grew. It originally simply tried to mimic the format and semantics of JSON by using Maps and Lists in place of JSON objects and arrays. This allowed us to easily implement new nodes into our interface but also meant that standard Erlang tools such as *Dialyzer (static type analysis for Erlang code)* (Ericsson AB, 2018) wouldn't work as it doesn't support matching specific entries at certain positions in Lists nor Maps.

We tried working around this issue by writing our own naive type checker

which we quickly realised was not extensible nor a good idea in the long run. We then heavily refactored *estree.erl* to use a custom datatstructure made up of nested tagged tuples. You can see an example of both pre-factored *estree.erl* code and post-refactored *estree.erl* in Figures 12 & 13 respectively in Appendix A.

Tagged tuples are easy to spec via Dialyzer as well, which lets us statically analyse our code to ensure certain data constraints are met; i.e. certain AST nodes only take other AST nodes are arguments in certain fields.

Originally, we wrote a JSON encoder to translate Maps and Lists literally into Objects and Arrays in JSON notation but was this a huge bottleneck due to lots of heavy recursive text manipulation (which Erlang isn't good at). Instead, we pulled and relied on external Erlang libraries such as *jsone.erl* to do this for us. Thankfully both our implementation of JSON encoding and *jsone.erl*'s both supported the same underlying datastructures so transitioning to *jsone.erl* was extremely simple.

4.2.4 Variable Declaration & Assignment in Matching

Integral to the functionality of Erlang programs is variable assignment via pattern matching; when the inputs of a pattern match clause are matched, any unbound variables within these inputs is assigned and bound to their corresponding literal. This process contains several non-obvious complexities:

- *Variables in a data-structure:* It is possible for a pattern to contain tuples or lists, and for subsequent code to use variables that are defined inside these data-structures (e.g. `{Var1,_,Var2}`). In this case Jarlang must recursively process the definition of the data-structure in the AST, identify all variables and generate code to access & assign the value.

- *Guards used matched values:* Guard conditions associated with each clause may use variables given value in the pattern match, so these variables must be assigned within the condition of the JavaScript if statement. Due to language restrictions this requires the condition to

be a lambda function.

4.2.5 Conditional Logic, Pattern Matching & Message Receiving

While Erlang has a variety of useful conditional logic (case, if, receive and pattern matching) all of these compile into the same format in Core Erlang. Figure 14 shows examples of pattern matching, case and if statements and a receive block. Figure 15 shows the generated Core AST, clearly showing the structure of parent node (either `c_case` or `c_receive`) and a list of `c_clause` nodes.

This simplification in Core Erlang allows us to use common code between pattern matching & receive blocks, and pay no special attention to case and if blocks.

4.2.6 Erlang Module Implementation

Outside of straight compilation of Erlang code to JavaScript code, the Jarlang compiler also does things that aren't neccessarily straight one-one translation.

JavaScript has no real definition of a module and as such, when compiling Erlang modules into JavaScript, we've had to essentially emulate how Erlang modules act. Erlang modules contain both private and public functions (in Erlang terms, exported and non-exported) as well as macros and module attributes (though he Jarlang compiler does not need to worry about macro expansion because this is done in our Erlang Compiler hook-in step).

On a high level, we simulate a module in JavaScript by creating a *const variable declaration* bound to the return value of a closure which contains anonymous functions representing our non-exported functions. Exported functions get put into a JavaScript object keyed by function name, which refer to a non-exported function in the closure body. This means that exported functions are seen to a user of our generated code as an object containing functions which is what we want.

One technicality of Erlang modules is that multiple functions can share the same name. Because Erlang functions are actually named by Module:FunctionName:Arity,

this isn't a problem. Encoding this in our JavaScript modules isn't an issue but because we want interoperability with JavaScript, as well as make it easy and intuitive to call into Erlang code from the JavaScript environment, we implement a faux-function overloading.

For example, even if our Erlang module contains the functions *fn/1, fn/2 and fn/3*, in our JavaScript module we only ever export a single *fn* function. This exported *fn* function switches based on the number of arguments provided to multiple cases representing the *fn/1 case, fn/2 case, fn/3 case* and a generic error case. This accurately mimics function arities in Erlang.

Module attributes are also encoded as object parameters and are ready to be returned and used by standard compiler generated debug functions though this is disabled by default as generally these attributes aren't used.

4.2.7 Outputting JavaScript & Other Tools

Once we've completely descended through the Core Erlang AST for a given module and have successfully translated it into an equivalent JavaScript AST, we simple write the JavaScript AST out to a temporary file. We then spawn a new shell process which calls NodeJS which is a pre-requesite for this project, which then runs our wrapper around an open-source NodeJS library — *escodegen.js* (ECMAScript Tooling, 2018) — to convert a JavaScript AST into code.

In order to make the user experience of Jarlang better and simpler, we also wrote a free and open source package builder for our project called *Erlpkg* (Bailey, 2018) which does things such as handler our command line argument parsing needs, as well and compressing all of our compiled Erlang source code modules into one binary file. This is because while there are official build tools for Erlang, we did not write Jarlang in line with the OTP specification which is required for these official build tools, and it was too late into the project to refactor what does work, and works well.

5 Conclusions

We consider the project to be a success, having contributed a valuable open-source resource to the Erlang community and creating the only publicly available Erlang transpiler that prioritises accurate representation of Erlang functionality over project simplicity.

Jarlang is able to parse most Erlang source code into valid JavaScript while preserving functionality. All implemented data-types function as they would in Erlang and the Runtime environment permits concurrency.

In the future we intend to:

- Implement the Erlang Standard Library so that transpiled Erlang will run without the user needing to implement the required functions in JavaScript.
- Integrate code optimization and minimising libraries to improve the output code.
- Implement an example distributed system, showing users how to override the default process spawning and message passing to run a program across multiple browsers.

6 Acknowledgement

The authors wish to thank and/or acknowledge the following:

- Scott Owens - For supervising our project, and being generally available whenever we needed help or assistance, and talked us through just about all of our technical decisions.
- Simon Thompson - For convincing us that rolling our own type checking system was not a good idea (and introducing to us the amazing *Erlang Dialyzer*), and talking with us about the feasibility in our concurrency approach.

- Gordon Guthrie - For his similar project — *LuvvieScript (URL now defunct)* — and though no useful source-code was shared, his talk about how he tackled hooking into the Erlang compiler and tools he used (*such as ESPrima and ESCodegen*) provided a brilliant starting point for *Jarlang*.

References

Bailey, C. (2018). Erlpkg github archive. `https://github.com/Vereis/erlpkg`.

ECMAScript Tooling (2018). Escodegen github archive. `https://github.com/estools/escodegen`.

Ericsson AB (2018). Dialyzer. `http://erlang.org/doc/apps/dialyzer/dialyzer.pdf`.

Gulp (2018). Gulp js github archive. `https://github.com/gulpjs/gulp`.

Guthrie, G. (2014). Luvvie script github archive. `https://github.com/hypernumbers/LuvvieScript`.

Guthrie, G. (2017). Luvvie script website via the internet archive. `https://web.archive.org/web/20171104224420/http://luvv.ie/toolchain.html`.

Mozilla Foundation (2017). Parser api. `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API`.

Tonpa, N. (2014). Shen erlang javascript parse transform. `https://github.com/synrc/shen`.

Appendix A. Sample code figures

```
...
error_if_not_apple(apple)->
ok.
```

Figure 9: Erlang source

```
...
[{{c_var,[],{error_if_not_apple,1}},
 {c_fun,[],
 [{c_var,[],'_cor0'}],
 {c_case,[],
     {c_var,[],'_cor0'},
     [{c_clause,[],
         [{c_literal,[],apple}],
          {c_literal,[],true},
          {c_literal,[],ok}},
          {c_clause,
              [compiler_generated],
              [{c_var,[],'_cor1'}],
              {c_literal,[],true},
              {c_primop,
                  [{function_name,{error_if_not_apple,1}}],
                  {c_literal,[],match_fail},
                  [{c_tuple,[],
                       [{c_literal,[],function_clause},
                       {c_var,[],'_cor1'}]}]}]}}},
...
```

Figure 10: Core Erlang AST of Figure 9

```
...
'error_if_not_apple/1': function (_cor0) {
  if (function () {
    if (_cor0.match(new Atom('apple'))) {
      return new Atom('true');
    }
  }.bind(this)()) {
    return new Atom('ok');
  } else if (function () {
    return new Atom('true');
  }()) {
    let _cor1 = _cor0;
    throw '** match_fail: TODO Errors dont parse nicely\\n' + 'Message';
  }
},
...
```

Figure 11: Jarlang transpiled equivalent of Figure 9

```
%% Generates a ThisExpression node
ThisExpression(Callee, Arguments) ->
    ?spec([{Callee, expression}, {Arguments, list_of_expression}]),
    #{"arguments" => #{"type" => [<<"ThisExpression">>]},
      "callee" => #{"type" => <<"ThisExpression">>},
      "type" => <<"NewExpression">>}.
```

Figure 12: Early ESTree AST representation plus custom typechecking macro

```
%% Generates a node data structure of a given type
%% with no additional parameters
-spec node(es_identifier() | atom()) -> es_node().
node(T) ->
node(T, []).

%% Generates a node data structure of a given type
%% while also setting additional parameters
-spec node(es_identifier() | atom(),
es_node_fields()) -> es_node().
node(T, Fs) ->
NewNode = {'__estree_node', T, [{"type",
    list_to_binary(atom_to_list(T))}]},
update_record(NewNode, Fs);

%% Updates a given node with the given NodeFields
-spec update_record(es_node(),
                    es_node_fields()) -> es_node().
update_record({_, _, NodeFields}, NewFields) ->
UpdatedNodeFields = merge_node_fields(NodeFields,
                                      NewFields),
{_, NodeType} = lists:keyfind("type",
                                   1,
                                   UpdatedNodeFields),
{'__estree_node', list_to_atom(binary_to_list(NodeType)),
                                UpdatedNodeFields}.


%% Generates a ThisExpression node
-spec this_expression() -> this_expression_node().
this_expression() ->
update_record(expression(), [{"type", <<"ThisExpression">>}]).
```


Figure 13: Refactored ESTree AST representation

```
...
pattern_match(apple)->
    ok.
f_case(Atom)->
    case Atom of
        apple -> ok;
        _ -> error
    end.
f_if(Atom)->
    if
        apple == Atom -> ok;
        true -> error
    end.
f_receive()->
    receive
        apple -> ok;
        _ -> error
    end.
...
```

Figure 14: Erlang code demonstrating various conditional logic syntax

```
...
[{{c_var,[],{pattern_match,1}},
 {c_fun,[],[{c_var,[],'_cor0'}],
  {c_case,[],{c_var,[],'_cor0'},
    [{c_clause,[],[{c_literal,[],apple}],
     ...
    {c_clause,[compiler_generated],[{c_var,[],'_cor1'}],
     ...
{{c_var,[],{f_case,1}},
 {c_fun,[],[{c_var,[],'_cor0'}],
  {c_case,[],{c_var,[],'_cor0'},
    [{c_clause,[],[{c_literal,[],apple}],
     ...
    {c_clause,[],[{c_var,[],'_cor3'}],
     ...
{{c_var,[],{f_if,1}},
 {c_fun,[],[{c_var,[],'_cor0'}],
   {c_case,[],{c_values,[],[]},
     [{c_clause,[],[],{c_call,[],
      ...
     {c_clause,[],[],{c_literal,[],true},{c_literal,[],error}}]}}},
{{c_var,[],{f_receive,0}},
 {c_fun,[],[],
   {c_receive,[],
    [{c_clause,[],[{c_literal,[],apple}],
     ...
    {c_clause,[],
     ...
   ],
...
```

Figure 15: Core Erlang AST generated from Figure 14