# CO600 Project: Jarlang
# Personal Report

Chris Bailey
cb661@kent.ac.uk


School of Computing
University of Kent
United Kingdom

Word Count: 1698 (via texcount)

March 31, 2018

# 1 Project Contribution

Throughout the project, it was primarily Andrew working on the Erlang codebase; implementing and refining our abstract-syntax tree translation module which quite literally is the main functionality of Jarlang as a compiler. On the other hand, Nick's contribution to the project was spent almost entirely in implementing the Jarlang runtime environment (our JavaScript codebase) so that we could actually execute the result of Andrew's work.

My contributions to the Jarlang project as a whole however, have been more or less equally distributed between both codebases, in both actually implementing features or bugfixes, as well as ideating and helping both Nick and Andrew acheive what they have. I will outline my main contributions below:

## 1.1 Project Architecture

One of the most notable contributions I've made to this project as a whole involve the several architectural decisions I've made in order to make this project as clean, extensible, testible and simple as possible.

I was the one who originally instigated the project, as well as the one who first started writing code for the project and as such paved the way for the initial direction we took. In addition to this, the majority of the research done for this project (i.e. finding other similar projects such as LuvvieScript, as well as finding out about useful tools which aided in our AST generation / design) was done by me which laid a strong foundation for future development.

As the one in our group who multitasked most, working on both the Erlang and JavaScript codebases, whenever I noticed something get very messy, or unneccessarily complex, I would call for a refactoring to make our codebases nicer, more extensible and better designed. Whilst these refactorings took valuable time away from functionality implementation, they made future functionality implementation much easier to perform.

This happened many times across both our codebases. For example, I

called for a heavy refactoring of Andrew's AST translation module when he needed help and his code was indecipherable in terms of the complex nested pattern matching he used to determine what functions needed to be called in order to perform translation. We ended up refactoring that to switch into several much smaller functions based primarily on the nodetype of a given AST node ultimately allowing us to easily add more features in terms of things we support translation of, as well as allowing me and Nick to help out if need be.

Another example in our JavaScript codebase is the fact that we were originally creating classes for our JavaScript representations of Erlang datatypes with the ES5 closure method, which works very nicely but gives us some drawbacks. The main drawback this faced was discovered when we wanted all of our datatypes to share an interface to allow the runtime system to assume things about all types (i.e. how to read the value of a type, how to stringify it etc). The most ideal and most elegant solution to this was to refactor our codebase to use ES6 Class constructors which allow us to use the 'extends' keyword — allowing all datatypes to inherit from a base class which not only perscribes a standard interface but also allows us to easily implement datatype matching via simple overridding of inherited methods and values.

Thus at a high level, one could say that I was responsible for our project following good software engineering practices, as well as consulting, commenting on and improving on our codebase iteratively throughout the project duration. This not only extends to the JavaScript or Erlang codebase but includes me taking it upon myself to set up a decent toolchain to build Jarlang, including writing Erlpkg (discussed below) and writing the initial makefiles and shell scripts needed to build Jarlang easily from scratch.

## 1.2 The Jarlang Compiler

As well as architectural / software engineering based contributions, I also worked on several features in the Jarlang compiler which follow.

Being the one who started our Erlang codebase, I was the one who researched methods of hooking into the Erlang compiler and extracting from it our CoreErlang AST which our project depends on. In addition to this, I also wrote an open source tool, Erlpkg, which not only builds binaries for Erlang projects, but also generates help text for good UX as well as parses command line options for users allowing us to support plenty of quality-of-life features such as outputting our intermediate compilation steps or toggling multithreaded compilation for better Erlang errors.

I was also the one who implemented module compilation in the Jarlang compiler, so whilst Andrew did the much more difficult task of compiling functions into JavaScript, I implemented code which took functions and a module prototype and created a correct JavaScript module out of these. To aide with this I implemented an interface for generating JSON nodes which represent JavaScript AST nodes which is used extensively by the core of the compiler.

Lastly, we all originally had wildly different programming styles and as such, over time, I enforced a common style guide for our Erlang modules such that it was easy and regular to read no matter what module you were reading. I integrated Dialyzer (a static analyzer for Erlang), as well as Elvis (a linter for Erlang) to aide with this.

## 1.3 The Jarlang Runtime

During the course of the project, I not only architected how we should implement the runtime system, but I also actively took a role in implementing parts of it.

The segments of the Jarlang runtime I implemented primarily centered around the actor-model features we offer including writing the actor agents themselves, implementing message passing functions to allow actors to communicate, as well as implementing the PID and Process datatypes such that PIDs would be unique identifiers for Processes, which would run asynchronously and encapsulate Erlang style concurrent behaviour.

In addition to this, whilst Nick primarily worked on designing the other datatypes Jarlang uses, as well as implementing much of the base modules currently integrated into the runtime (IO and Erlang modules), I implemented the core of the runtime including emulation of the process queue, atom table, and ETS table. I also implemented just about anything which needs to reason or work with processes as outside of a few key places, our concurrency abstraction is completely transparent.

I also implemented the layers responsible for interopability between the JavaScript runtime and our Jarlang runtime.

# 2 Project Reflection and Evaluation

In retrospect, I personally think we as a group did many things right for Jarlang. I feel as though Jarlang in its current state is at least functional and can successfully compile and execute Erlang code on/for the web.

There are a few things I'd change if given the chance to go back and start the project from scratch again though. As the person solely responsible for most of the architectural decisions we've made, I admit that it took us far too long to get to a point where we worked under best practices for Erlang programming — not only did we try to write our own typechecker, we completely failed to successfully test our code or utilise Erlang tools such as dialyzer until halfway through the project.

In addition to this, we ended up refactoring our code a lot, which whilst not a bad thing perse, definitely did waste time I would have liked to dedicate to implementing new features. This is because whilst everyone on our team is a strong programmer with a diverse range of expertise, everyone was implementing things by themselves before pushing their changes to our Git repository, which means that it was only after features were fully implemented that we were able to critically comment on them.

If we had utilised more pair-programming or at least utilised branches more, we would be able to either critically comment on how things were be-

ing implemented before it would require a large refactor, or be able to push more incremental updates to our Git repo without worrying about breaking anything, allowing other team members to look at our changes in more incremental steps.

Because we failed to utilise Git branches successfully, and lack any sort of continuous integration or testing framework, often times we would end up breaking features which previously worked, only setting us back for swathes of time if the feature being broken wasn't noticed posthaste.

Despite having makefile targets for dialyzer, linting and testing, very few tests were written and most of the time my colleagues wouldn't remember to lint or run dialyzer before asking for help or trying to debug blind. Our JavaScript code also didn't have any tests or even a bundler until very recently, which did slow down development.

Though critically speaking, I personally do believe our project has reached at very least what I wanted out of it at the beginning of the year; when I proposed the project to Nick and Andrew, I thought it was an insane idea which was completely out of reach. We all approached the year ready to take the outcome of the project, no matter what, as a huge learning experience and we simply gave it everything we thought it needed which seemingly paid off.

Despite parts of it not functioning exactly as we want it yet, as well as the likely possibility of there being bugs in our codebase from simple lack of testing and lack of knowledge, I would still say that from an academic perspective as well as a pragmatic one, our year spent developing Jarlang has been successful.

# 3 The Future

There is a common consensus amongst our group that whilst we recognise the current flaws of our project, we are all rather proud of where the project has gotten in such short a time. We are however, obviously dissapointed that

we haven't implemented everything to a quality which we'd like — as stated, there are quite a few things I'd personally change if given a chance to restart — but this does not mean that our project will statically remain how it is.

At very least, me and Andrew have agreed that after the conclusion of this project, we both want to continue working on maintaining Jarlang as an opensource project free for anyone to use. This includes not only bugfixes but also continued implementation of Jarlang for the forseeable future, which I can say is honestly quite exciting as I genuinely enjoy working on this project.