Planning report

Hopper - a Haskell like language for the Erlang VM

Group 29

Background

The Erlang virtual machine is appreciated for its excellent support for concurrency, distribution and error handling, allowing soft-real-time systems with high uptimes - famously demonstrated by the 99.9999999% uptime of Ericsson's AXD301 telecom switch from 1998. The features of the BEAM (Erlang's main virtual machine implementation) aren't tied to Erlang itself: one can enjoy the benefits of the runtime with a different syntax and semantics. Previous efforts in this field include: Elixir, a language with Ruby-like syntax; Javascript flavoured Erlang, Lisp flavoured Erlang and a YHC Haskell to Erlang transpiler that preserves laziness.

Attempts to port Erlang features to Haskell have also been made, illustrating the demand for a combination of the languages' features. Cloud Haskell is 'Erlang in a library', but differences in the Erlang and Haskell runtime system make shipping expressions between nodes far more verbose than in Erlang.

Haskell is known for its terse and elegant syntax, in addition to a powerful type system that prevents many programming errors and makes it easier to reason about code. Erlang developers already use code analysis tools to write typed code, but they don't interact well with message passing where the type received can be hard to predict. Combining the compile-time features of Haskell with the run-time benefits of Erlang into a language interoperable with the latter would be a great boon for Erlang developers.

Purpose

The overarching idea with the project is to design and implement an alternative language, named Hopper, to the Erlang language on top of the Erlang Virtual Machine. The language will feature a Haskell-like syntax, offer a typesystem similar to that of Haskell and expose, in a safe way, the powerful features of the Virtual Machine. To utilize the big Erlang ecosystem it will also integrate with current OTP and Erlang libraries.

The purpose of this bachelor project is for the group to learn about functional programming, compilers and how a typesystem works and is constructed.

Problem and Task

In general

"The goal of this project is to design and develop a prototype compiler for this new language".

The group wants to design and implement a language that utilize the benefits of the Erlang VM and the OTP libraries combined with the clean syntax and static typing of Haskell.

Additional requirements the group pose on the solution are:

- Practical usage
- Real benefits from the type system
- Well documented/specified

The group is not aiming at a complete and polished product but rather a prototype or proof of concept.

Design language

There are two sides of computer languages. Purely syntactic - the structure of the language, and semantic - the meaning of the language. The syntax will be given by a grammar and the semantics of our features will be defined in terms of Core Erlang. The group will design and formalize the interaction between haskell code and native erlang code.

A list of design features desired in the language:

- Static typing
- Clean syntax
- Transparent access to Erlang BIFs
- Support for message passing
- Functional and pure
- Abstract Data Types

Possible later additions:

- Type classes

Some example code of what Hopper might look like:

```
module Server where
exported
    startServer :: IO ()
    serverAtom :: Atom

newType State a = State a

-- Create the initial server state
initState :: State Int
initState = State 0

-- Modify server state
inc :: State Int -> State Int
inc (State n) = State (n+1)

-- Get Int from state
get :: State -> Int
get (State n) = n

serverAtom :: Atom
serverAtom = stringToAtom "server"

-- Start a new server process and register its pid with the "server" atom
startServer :: IO ()
startServer = do
    pid <- spawn (server initState)
    register serverAtom pid
    return ()

-- Server process. Receives messages of the type (PID, String).
-- If no new message in 1000ms, shut server down.
server :: State Int -> IO ()
server s = do
    receive :: (PID, String)
        (_, msg) -> putStrLn (show s ++ ": " ++ msg)
    after
        1000     -> do
            unregister serverAtom
            return ()
    server (inc S) -- TCO!
```

Design compiler

The compilation process goes through several stages: lexing/parsing, type checking and code generation. The group will initially use the BNFC tool to specify

the lexer and parser. Later the group might implement their own lexer and parser. For the code generation the group intends to use an intermediate language called Core Erlang to simplify the generation of BEAM code. The group will put emphasis on ease of use by for example having readable error reports.

Implement compiler

The specifics of the implementation depends heavily on the results from the language and compiler design phases. Two important aspects during this phase will be documentation and quality assurance, mainly through testing. The compiler will be written in Haskell.

Limitations

The main goal is a working prototype with a core set of features. The focus of the project will be these features. Any additional features will be deferred until the group has achieved the main goal.

Method and implementation

In general

The group will implement an iterative development process. The product, being the combination of the language specification, the typechecker and the compiler, will be designed, implemented, tested and analyzed. The analysis of the test results will then be used for improving the design in the next iteration. This process will be used throughout the project. If the product is considered to be (or be close to) non improvable during the analysis step, additional features will be considered in the design step in the next iteration. The test results and decisions made will also serve as the main source of data for the final report

Design

There will be three main parts of this project that require designs: the language specification, the typechecker and the compiler. The language specification, the typechecker and high level compiler organization will be designed by close collaboration between all group members, while the concrete components of the compiler will be distributed amongst the members. A member will be responsible of leading the design of its component with help from other group members. The work will be distributed in such a way so that each member has worked with every component of the language specification and the compiler on some level. Quality criteria will be developed for the language specification and the compiler components during the design step. Quality criteria regarding the

compiler components will address how well the compiler parses and translates the specified language. The criteria regarding the language specification will address how well the produced code runs on the Erlang VM, and how well it interoperates with Erlang libraries.

Implementation

The typechecker and the compiler will be implemented by the design developed in the design step. The members responsible for the design of a component will also be responsible for its implementation. As in the design step, members will help the responsibles with the implementations and the work will be distributed in such a way so that each member has worked with every component of the product on some level. The language specification does not require this step; it will be developed in the design step.

Testing

Automated and/or manual tests will be designed and implemented following the quality criterias produced in the design step. The concrete components of the compiler will be tested using black box testing, using mock objects if integration testing within the compiler should be needed. Mock objects will be implemented on demand. The group will produce a collection of Haskerl example code used when testing. All members will participate in deciding quality criterias and designing as well as developing the test suites. Tests used in passed iteration that are related to any developed or revised software should be invoked during this step.

Analysis

Analysis of the test results will form the main decision basis. The conclusions from this step will be used for determine the quality of the product. The data gathered from this step is closely related to the quality criterias that the group defines in the design step. The analysis will investigate how well the language specification and the compiler fulfill these requirements. Depending on what the analysis result in will then influence the next iteration, being that the language specification and/or the compiler will be improved, revised or changed on a larger scale.

Time plan

As explained in the method section, the group will use an iterative development process.

The purpose of the first iteration is to get a handle on the tools that we will be using as well as the process itself, and will run for about two weeks. Less focus is put on the language, whose main goal is to be as simple as possible in order to not obscure other difficulties.

After that a second, fuller iteration will be timed so that it will be finished about the time of the half time presentation in the course.

After these two initial versions, an additional number of iterations will follow. Their individual length and scope will be determined based on the testing and analysis of previous iterations.

This is illustrated in the attached Gantt chart.

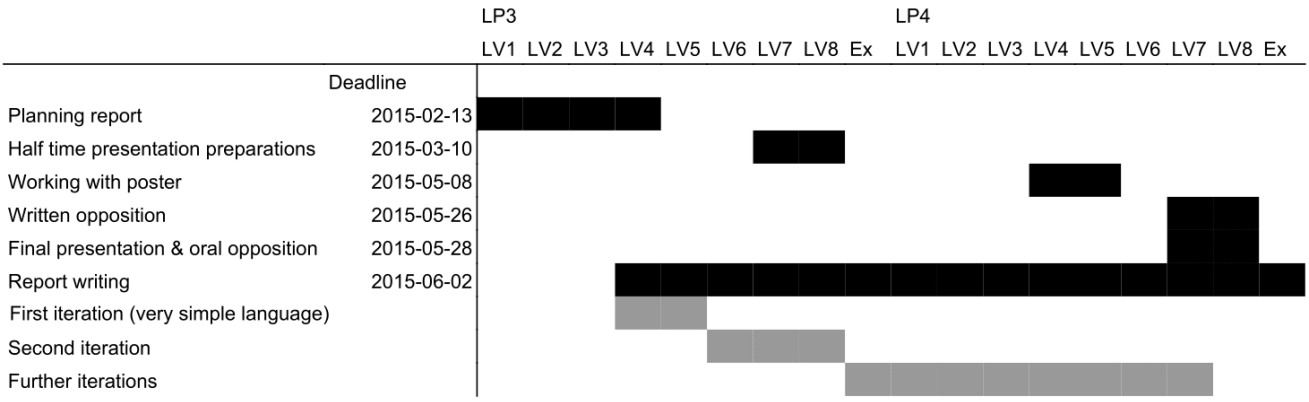

Figure 1: Gantt chart