CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

The Hopper language

A Haskell-like language on the Erlang VM

Bachelor of Science Thesis in Computer Science and Engineering

William Hughes
Jakob Jarmar
Johan Larsson
David Lindbom
Björn Norgren
Johan Wikström Schützer

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden, Spring 2015

The Hopper Language
A Haskell-like language on the Erlang VM
William Hughes
Jakob Jarmar
Johan Larsson
David Lindbom
Björn Norgren
Johan Wikström Schützer

Supervisor: Nicholas Smallbone, Department of Computer Science and Engineering
Examiners: Arne Linde, Department of Computer Science and Engineering
Jan Skansholm, Department of Computer Science and Engineering

Bachelor Thesis 2015:29
Department of Computer Science and Engineering
DATX02-15-29
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2015

The Hopper language
A Haskell-like language on the Erlang VM
DATX02-15-29
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

The following report aims to give insight into the design and implementation of a statically typed functional language for the Erlang virtual machine, discussing how such an implementation may be approached and whether it appears to be feasible. The primary goal of the project was to design a grammar specification and implement a compiler for such a language.

Over the course of the project a prototype language and a compiler for that language were developed. The project followed an iterative development process with Scrum as a basis. Notable modules of the compiler are the parser generated from a BNF grammar, the type checker implementing a Hindley-Milner type system and the code generator generating Core Erlang source code.

The result of the project is Hopper, a basic functional programming language with an accompanying compiler, featuring polymorphic algebraic data types (ADTs), pattern matching and lambdas. The language also has a module system and some integration with Erlang.

In conclusion, the project was largely successful in its mission to create a typed functional language on the Erlang VM and has the potential to be developed further.

Sammanfattning

Den följande rapporten syftar till att ge insikt i utformningen och implementeringen av ett statiskt typat funktionellt programmeringsspråk för Erlangs virtuella maskin, och diskuterar hur man kan gå tillväga med en sådan implementation samt om det verkar genomförbart.

Under projektets gång har ett prototypspråk samt en kompilator för språket utvecklats. Projektet följde en iterativ utvecklingsprocess med Scrum som utgångspunkt. Noterbara delar av kompilatorn är parsern, genererad utifrån en BNF-grammatik, typkontrolleraren som implementerar ett Hindley-Milner-typsystem samt kodgeneratorn som genererar Core Erlang-källkod.

Resultatet av projektet är Hopper, ett enkelt funktionellt programmeringsspråk med tillhörande kompilator, med polyformiska algebraiska datatyper, mönstermatchning och lambdauttryck. Språket har även ett modulsystem och viss integration med Erlang.

Sammanfattningsvis var projektet till största delen lyckat i sin uppgift att skapa ett typat funktionellt språk på Erlangs virtuella maskin och har potential för vidareutveckling.

Acknowledgements

We would like to give a special thanks to our supervisor Nicholas Smallbone. We would also like to thank examiners Arne Linde and Jan Skansholm, as well as Claes Ohlsson for his reflections on our report.

Contents

Glossary **ix**

List of Figures **x**

List of Tables **xi**

1 Introduction **1**
1.1 Problem . . . 1
1.2 Purpose . . . 1
1.3 Scope . . . 1
1.4 Report overview . . . 2

2 Technical background **3**
2.1 History . . . 3
2.2 Related work . . . 3
2.3 Compilers in general . . . 4
2.4 Parsing source code . . . 4
2.5 Module systems . . . 5
2.6 Type checking . . . 5
2.7 Core Erlang and BEAM . . . 6
2.8 Algebraic data types . . . 6

3 Methods **7**
3.1 Development process . . . 7
3.2 Code revision control . . . 7
3.3 Programming language . . . 7

4 Design and Implementation **9**
4.1 Compiler pipeline overview . . . 9
4.2 Parser . . . 10
4.2.1 The grammar . . . 10
4.3 Renamer . . . 10
4.4 Module system . . . 11
4.4.1 Dependency model . . . 12
4.4.2 Interface files . . . 12
4.5 Type system . . . 13
4.5.1 Implementation . . . 13

4.5.2 Algebraic data types and constructors . . . 14
4.6 Code generator . . . 15
4.6.1 Implementation . . . 15
4.6.2 Function applications . . . 15
4.6.3 Emulating the Erlang compiler . . . 16

5 Results **18**
5.1 The Grammar . . . 18
5.2 The Compiler . . . 19
5.2.1 Modules . . . 19
5.2.2 Type checker . . . 19
5.3 Missing features . . . 20

6 Discussion **21**
6.1 Grammar . . . 21
6.2 Modules . . . 21
6.3 Type checker . . . 22
6.3.1 Algebraic data types and constructors . . . 23
6.4 Code generator . . . 23
6.4.1 Function applications . . . 23
6.5 Code quality and user interaction . . . 24
6.5.1 Documentation . . . 24
6.5.2 Testing . . . 24
6.5.3 Usability . . . 24
6.5.4 Interoperability . . . 24
6.6 Methods discussion . . . 25
6.6.1 Agile development - sprint length . . . 25
6.6.2 Lack of daily communication . . . 25
6.6.3 Lack of a project leader . . . 25
6.6.4 Lack of a hard specification . . . 26
6.6.5 Revision control system . . . 26
6.6.6 Type checker branches . . . 26
6.6.7 Responsibility rotation . . . 27
6.7 Social and economic impact . . . 27
6.8 Future work . . . 27

7 Conclusion **28**

Bibliography **29**

Appendix A Hopper grammar specification **I**

Appendix B `curry` implementation **IV**

Appendix C Hopper example **V**

Appendix D Core Erlang example **VI**

Glossary

ADT algebraic data type. iv, xi, 6, 10, 11, 14, 15, 18, 19, 23

algorithm W Algorithm W is an algorithm which infers the types of expressions in the Hindley-Milner type system. 13

arity The arity of a function is the number of arguments it accepts. 15, 16, 23

AST abstract syntax tree. 10, 15, 23

BEAM Erlang's Virtual Machine. 3, 6, 19

BIF built in function. 11, 18

BNF Backus-Naur form. iv, 4, 10

BNF Converter A program developed at Chalmers for converting BNF grammar to a lexer and parser. See reference [11] 4, 8, 21

constraint A constraint is an equality statement between two types. The constraint $Int \rightarrow Bool = a \rightarrow b$ will, as an example, be resolved into the constraints $a = Int$ and $b = Bool$. 14

Core Erlang An intermediate language in the Erlang compilation suite. iv, 9

currying The process of transforming a function which takes a tuple of `n` arguments into one which takes one argument at a time. When the curried version of `f` has been applied to `n` arguments, it returns the result of passing a tuple containing those arguments to `f`. 16

DAG directed acyclic graph. x, 12

Hindley-Milner type system A type system for the lambda calculus with parametric polymorphism. Also known as Damas-Hindley-Milner. iv, 13, 22

polymorphism parametric polymorphism: a function or type declared with generic parameters that handles them identically. Ex. the identity function id has type $\forall a.a \rightarrow a$. 13

SCC Strongly connected component. A subset of a directed graph in which there is a path from any vertex to all others. 13, 14

semantic analysis In the context of compiler construction semantic analysis is performed after lexing and parsing and checks for errors which can't be captured by the grammar of the language. For example that variables are defined before their use. Type checking is one kind of semantic analysis. 4, 5

topological sorting A topological sort of a directed graph is an ordering of its vertices such that for each edge uv from vertex u to vertex v, u comes before v. A DAG always has at least one topological sorting. 12, 13

type checking The process of proving types of an expressions. 4

type system A type system is a formal system wherein types of expressions can be inferred. Inferring a type amounts to proving the type using the inference rules of the formal system. 5, 13

unification Two types A, B are unifiable if there exists a substitution S, from type variables to types, such that $S(A) = S(B)$.

For example unifying a type (variable) a with the type Int would yield a substitution $S = [a/Int]$ since $S(a) = [a/Int]a = Int = [a/Int]Int = S(Int)$. 14

List of Figures

2.1 A general compiler pipeline 4

4.1 Hopper's pipeline 9
4.2 Simple transformation to a more general form 11
4.3 Transforming an ADT to two function types 11
4.4 Transforming a pattern matching function to a case expression 11
4.5 Example dependency graph 12
4.6 Definition of a list data type and the map function 13
4.7 Example module with mutually recursive components 13
4.8 The application inference rule 14
4.9 Hopper ADT with example value 15
4.10 Runtime representation of an ADT value 15
4.11 Simple Hopper function 16
4.12 Generated Core Erlang translation of function 17

5.1 Export listing and import statements 19

6.1 Compilation with dependencies 22

List of Tables

5.1 Optional arguments for the Hopper compiler 19

1
Introduction

There is a trend toward more type checking, more concurrency and more parallelism in modern programming languages. Type checking to manage correctness in ever more complex systems, concurrency to manage large distributed systems and parallelism to utilize the growing number of cores available in computers. Two examples following this trend are the Swift [1] and Rust [2] languages.

1.1 Problem

Each of these concerns have spawned viable solutions, notably Haskell [3] for type systems and Erlang [4] for concurrency and parallelism. But as far as the group knows there have been few successful attempts at combining these solutions into a language which offers the benefits from a strong type system together with the benefits of native concurrent and parallel features. The group will take on the challenge of combining the merits of these fields in a new programming language called Hopper, endowed with an expressive type system as well as native facilities for concurrent and parallel programming. The question that the project aims to answer is whether a prototype for this new language can be implemented in such a way that it is powerful, containing the desired features, and user friendly.

1.2 Purpose

The main focus in the following chapters is on the development of the Hopper compiler. The purpose is to give insights into problems that arise in the melding of type systems with concurrent and parallel capabilities. Further, it should also provide some guidance in how these problems could be solved, or in the worst case how they should not be solved.
This report might interest the programming language community and serve as a springboard into a more exhaustive attempt at combining an expressive type system with concurrent and parallel primitives.

1.3 Scope

As programming language design and implementation is a complex field of study, the group will need to limit the ambition of the project to fit the format of a bachelor's

thesis project. The aim is to create a prototype of the target language. This will give some room to prioritize design at the expense of a polished implementation.

1.4 Report overview

In the following chapters the reader will be guided through some preliminaries, to the design and implementation of the language, to finally end up in discussion of the results and methodology.

Chapter 2 details a short history and background of the technological frameworks.
Chapter 3 explains the methodology used by the group and describes some project management tools.
Chapter 4 explains the algorithms and implementation decisions made developing the prototype.
Chapter 5 takes inventory of the results achieved in the project.
Chapter 6 contains a discussion of the results.
Chapter 7 concludes the report with some final thoughts.

2

Technical background

This chapter gives a brief history of Haskell and Erlang, and a technical background to some topics covered in the report.

2.1 History

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 telecommunication switch from 1998 [5].

Haskell is known for its terse syntax, in addition to an expressive 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 these tools don't interact well with message passing where the type received can be hard to predict.

The features of the BEAM, Erlang's Virtual Machine, are not tied to Erlang itself. BEAM hosts a variety of languages, so it is possible to enjoy the benefits of the runtime system with a different syntax and semantics. Previous efforts in creating alternatives to Erlang as a front-end language include Elixir[6], a language with Ruby-like syntax, and Lisp flavored Erlang [7].

2.2 Related work

Previous attempts to combine the features of Erlang and Haskell have been made, illustrating the demand for a combination of the languages' features.

Attempts to port Erlang features to Haskell include Cloud Haskell [8], which is "Erlang in a library", essentially implementing Erlang's runtime system in Haskell. However, a lack of syntactic support for message passing in Haskell make shipping expressions between nodes more verbose than in Erlang.

Efforts have also been made to port Haskell features to Erlang. One example is the Haskell to Erlang transpiler backend to YHC [9], which preserves laziness. A similar project is Haskerl [10], which compiles a subset of Haskell to the BEAM. However, the group did not find projects which also attempted to integrate BEAM's concurrency capabilities on a language level, which is one of Hopper's goals.

2.3 Compilers in general

New programming languages are often developed to allow for a higher level of abstraction and a higher level of reasoning about algorithms. To be able to run these new languages, compilers are built to convert the high level code either down to machine code, to run directly on hardware, or to an intermediate language to make use of already existing compilers.

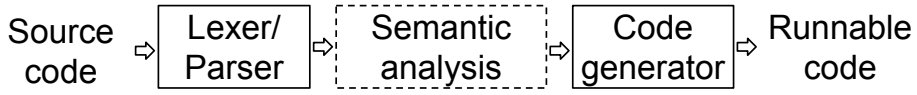

Figure 2.1: A general compiler pipeline

The structure of a compiler can be likened to a pipe where text flows through. The text flowing through the pipe is transformed between different representations, and is in the end output as runnable code to either run on the computer itself or on a virtual machine that emulates a computer.

To ensure that the compiled program is correct and does what it is intended to do, semantic analysis is performed upon the code. An example of such analysis is building a call graph to identify unreachable code and to make sure that variables are defined before use. Another example is type checking, where type correctness is proved, which is used heavily in Haskell.

Figure 2.1 shows the flow handling of the source code. It should be noted that semantic analysis is optional: languages may rely entirely on the grammar as their only way to expose errors. This will lead to a more expressive, or less restricted, language, but increases the risk of runtime crashes due to type errors.

The next section continues by describing the first step of the compilation process, lexing and parsing.

2.4 Parsing source code

A programming language is similar to a natural language in that it is described by a grammar. Programming languages are however different in that they are built from their grammar and are thus unambiguous. The grammar describes ways in which the smaller pieces of the language can be put together to form more complex constructs.

From such a grammar a lexer is produced. The job of the lexer is to match all words in the source code to tokens in the grammar. For example, in English the word "boat" would match a noun token. This list of tokens is combined with a parser to make expressions or sentences. To make the language as expressive as possible bigger expressions are built from smaller expressions. This creates a tree structure that will be easier to work with.

To save a lot of work, the lexer and parser can be automatically generated from a grammar written in Backus-Naur form (BNF) using a tool called the BNF Converter[11], or BNFC for short. This tool is developed at Chalmers.

2.5 Module systems

As computer programs increase in complexity, a need for separation of concerns arises. Programming languages typically solve this problem by enabling the programmer to organize code into cohesive units known as modules, sometimes also referred to as packages or components. Besides the organizational benefits, there are technical merits to modular software as well. If a single module is changed, the entire project does not necessarily have to be recompiled; only the parts which depend on the module that was changed.

The concept of dependency is central in module systems, but details differ between programming languages. Some languages allow the programmer to define the interfaces between modules, for example by marking certain functions as `public` or giving a list of functions to export, while others automatically make all declarations available to other modules. Other features also differ between languages, such as allowing cyclical inter-module dependencies or whether dependencies between modules must be explicitly declared with `import` statements or similar.

2.6 Type checking

Programmers inevitably make mistakes. While there are many tasks computers excel at, understanding the programmer's intent and detecting differences between it and the actual behavior of the programs they write is not one of them. To aid in the automatic detection of errors, semantic analysis may be used. One such kind of analysis is type checking, which is enabled by having a type system.

Type systems are formal systems which enable proof of absence of certain kinds of errors. They are typically defined in terms of inference rules.

Type systems can be used to help in rejecting invalid programs. They group values in the programming language into types, and specify rules about what is or what is not a valid interaction between them. For example, a type system might be defined to reject a program that tries to apply a number to a value as if it were a function.

There are two main kinds of type systems: dynamic and static. The most noticeable difference between dynamically and statically typed programming languages is whether type errors are discovered at run time or compile time.

Static type systems automatically detect type incorrect code at compile time, such as a program treating a Boolean as a list of Booleans. This allows the programmer to diagnose errors during the implementation phase. In contrast, dynamic type systems handle types during runtime, allowing the programmer to manipulate data in a more free manner.

An additional benefit of type systems is that they can be used as a means of specifying and documenting properties of a program in a concise way. Typically, a function may be annotated with a type signature which states the types of its arguments and its return type.

2.7 Core Erlang and BEAM

Core Erlang [12] is an intermediate language in the Erlang compilation suite, mainly used for simplifying operations on Erlang source code. Such operations can include, but are not limited to: parsing, compiling and debugging. Some of the main purposes of Core Erlang are to be as regular as possible and to provide clear and simple semantics. The language is since release 8, released in 2001, an official part of the OTP/Open Source Erlang Distribution.

BEAM is the name of the Erlang virtual machine. Erlang and Core Erlang source code may be compiled to `.beam` files, which contain assembler instructions for the BEAM virtual machine. These instructions are interpreted by the virtual machine at run time.

2.8 Algebraic data types

ADTs are types composed using constructors, and are often used to define complex data structures. Constructors take zero or more arguments and return a composite value containing the given values. Such a composite value may then be deconstructed and analyzed, typically using pattern matching. An ADT may have several constructors with different combinations of data, and may also be recursive. This provides a simple but powerful way of expressing rich data structures.

ADTs do not exist in Erlang in the same way as they do in Haskell, i.e. with their heavy reliance on constructors. When using pattern matching in Erlang, entities called atoms are instead used to identify different cases. An atom is a literal constant with a name.

3
Methods

The following chapter describes the methods and tools that were used when developing the Hopper language and its compiler. The approach to design and implementation taken was intended to enable a productive and stable work flow for group members working simultaneously on different parts of the compiler pipeline.

3.1 Development process

The development process used for the project was a derivation of Scrum [13]. The project was divided into short sprints with each sprint being planned at its start. Sprint lengths spanned from one to two weeks depending on the estimated work load. During the planning phase of a sprint each member was assigned one or more goals that they would be responsible for. The members then directed the design, implementation and testing of their areas of responsibility throughout the sprint, either alone or with help from other members of the group. Sprints were then closed after an evaluation process held during weekly meetings.

Some scrum characteristics were left out, for example the appointment of a scrum master and a daily meeting. Sprint review and planning was carried out during the weekly meetings with few intermediate whole group activities.

3.2 Code revision control

When developing the Hopper language and implementing its compiler, Git [14] was used for code revision control. Revision control systems enable development using several simultaneous contributors. A central repository is used to host the latest version of the code with group members being able to add features locally and push any changes to the central repository when done.

To enable a flexible and parallel development process, different components and features in the project were implemented using branches. When said components and features were believed to be in a usable state, the containing branch was merged into a shared development branch.

3.3 Programming language

The main implementation language chosen was Haskell. Since the aim of the project was to implement a Haskell-like language on the Erlang VM, Haskell and Erlang were

prime candidates, as those choices would allow us to make use of their similarities to Hopper.

Haskell has the advantage of being syntactically closer to what we wanted Hopper to be and being good for describing high level abstractions. Haskell also has well-used tools for compiler writing, such as Alex [15] and Happy [16] (Haskell's versions of the standard tools Lex [17] and Yacc [18]), as well as the BNF Converter. Erlang, on the other hand, is closer to what would be sent to the Erlang VM but has less used tooling for writing compilers. Also, the group members were less experienced with Erlang.

Sketching out the project there were plans of possibly moving from a Hopper compiler written in Haskell to a Hopper compiler written in Hopper. This is sometimes called a self-hosting compiler since it would then compile itself. However, this was quickly considered to be out of scope for the project due to time constraints. A self-hosting compiler could be a project of its own with implementing all the standard libraries for lexing and parsing as well as common data structures.

4

Design and Implementation

This chapter describes the design and implementation of the Hopper language and the Hopper compiler with all its components. There are overviews of each part describing the choices made and the reasoning behind those choices.

When first designing Hopper there was already an outline of what the language should aim to be. The type safety and clean syntax of Haskell combined with the powerful framework for concurrency and parallelism supplied by the Erlang virtual machine were the key features from which Hopper would be built. Within those confines there were further choices as to which features to include, and which to exclude, to make the language relevant and useful.

4.1 Compiler pipeline overview

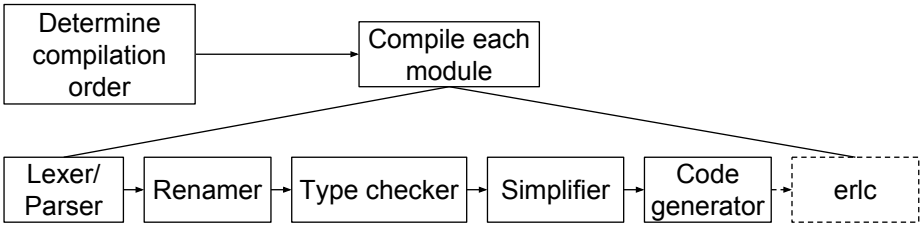

Figure 4.1: Hopper's pipeline

The core compiler functionality in Hopper is built to work on a single source file at a time. In order to support programs consisting of multiple modules the Hopper compiler generates a compilation order for all modules in that program so that they can be compiled one at a time (see section 4.4).

As seen in figure 4.1, the Hopper language has five major steps in its compiler pipeline before it hands over to the Erlang compiler. Each step works on the source code representation in a certain way.

The steps can be summarized as follows: the lexer and parser read in text files and convert them to an abstract representation, the renamer then simplifies this representation for the type checker. The type checker confirms and determines the types of all expressions. Lastly there is another simplifying step before the code generator, which produces Core Erlang code. The Erlang compiler is then invoked on the produced Core Erlang code, and the result is Erlang assembly which can be run on the Erlang VM.

4.2 Parser

The parser is implemented by writing a BNF grammar and feeding it to the BNFC [11] tool. BNFC, as described in the background, is the BNF Converter developed at Chalmers which the group used to automatically generate a lexer and a parser.

To have as clean syntax as Haskell, the grammar should be indentation sensitive instead of depending on braces and semicolons as in C-like languages. Parsing code with braces and semicolons is easier because one does not need to keep track of which indentation context the code is in. BNFC can help with this by converting an indentation sensitive language to a brace dependent one. It does this in the lexer by inserting tokens around expressions, making it easy to match against these in the parser.

4.2.1 The grammar

Writing the grammar one describes the ways in which programs accepted in the language can be written. Hence it takes great consideration to ensure that the grammar allows everything that the language is supposed to be able to express but disallows everything else. This does not include expressions that are semantically wrong. To deal with this, semantic analysis of the abstract syntax tree (AST) can be done with a type checker, for instance.

The grammar in the project was produced in incremental steps that allowed more and more syntax to be added. This made it possible to have functionality implemented through the whole pipeline before new syntax was added.

A full rewrite of the grammar was done half way through the project. The reason for this was that some grammatical rules were intertwined and that the understanding of how a correct grammar should be defined had been improved. The new grammar was more modular which also made further extensions easier.

Some syntactic sugar for `if-then-else` was implemented and more was planned, for example sugar for lists and tuples.

4.3 Renamer

While the parser generated by BNFC is a great tool for creating a parse tree from raw text the resulting tree structure is complex and verbose. To simplify this representation a renamer step was introduced in the pipeline. This step converts the parse tree to the language's simplified AST.

The renamer is implemented as a traversal over the parse tree. In this traversal the renamer builds up a new tree of constructors for the AST. A few grammar rules are translated to more general expressions to simplify the AST and reduce the number of cases the type checker and code generator need to cover. In figure 4.2, an example of an `if-then-else` expression and its equivalent `case` expression translation.

The Renamer has a few more tasks to take care of. First it transforms the ADTs and gathers functions defined with pattern matching and merges them. When the

```
f = if a                        f = case a of
      then b          ⇒               True  -> b
      else c                          False -> c
```

Figure 4.2: Simple transformation to a more general form

renamer encounters an ADT definition it converts each constructor to a function with just a type. With this the type checker can verify that it is used correctly and leaving it fully to the code generation to implement it without any dependencies. This transformation can be seen in figure 4.3.

```
data Maybe a =                  Just :: a -> Maybe a
  Just a   |          ⇒        Nothing :: Maybe a
  Nothing
```

Figure 4.3: Transforming an ADT to two function types

Second it annotates all the identifiers, except variables bound in a pattern matching scope, with the module name. This removes the ambiguity between functions that have the same name but are defined in different modules. Lastly it checks a list of built in functions (BIFs) to switch them out for calls to the virtual machine instead of regular user defined function calls.

While implementing the renamer much consideration was taken as to what later stages of the compilation needed. For example, while Erlang allows pattern matching in functions Core Erlang does not. This, as shown in figure 4.4, is the reason for merging function definitions to a single definition with a case expression and each previous function as a clause. For the same reason the type representation was changed to fit better with the type checker.

```
f 0 = 0                         f = \ a -> case a of
f n = n * f (n-1)      ⇒                 0 -> 0
                                         n -> n * f (n-1)
```

Figure 4.4: Transforming a pattern matching function to a case expression

4.4 Module system

The module system of Hopper is designed to be simple to implement. Hierarchical module names are supported, e.g. `A.B.C`. Only a single source directory is supported, so there is a straightforward translation from module names to file paths. For instance, the above example maps to `A/B/C.hpr`. This restriction removes the possibility of conflicts where multiple matching source files exist in different source directories.

4.4.1 Dependency model

Modules explicitly declare what functions and data types they export, and declare dependency on other modules using `import` statements. Importing a module means importing everything that module exports, so fully qualified names are required when referring to imported functions or data types. For example, if module `A` defines a function `f` which module `B` wishes to use, module `B` must first `import A` and then use the notation `A.f` whenever it wants to use `f`. This ensures that imported functions do not cause name clashes.

The import statements are used to build a graph representing the dependency relations of the program. Verifying that the graph is a directed acyclic graph (DAG) ensures that no cyclical dependencies occur. Topologically sorting the nodes of the DAG gives a compilation order where the dependencies of each module are compiled before that module. This design ensures that type information for a module's imports is available at compile time, so that the compiler only has to consider a single module at a time. For example, with the module hierarchy in figure 4.5, attempting to compile module `A` yields a compilation order of modules `D`, `B`, `E`, `C`, `A`.

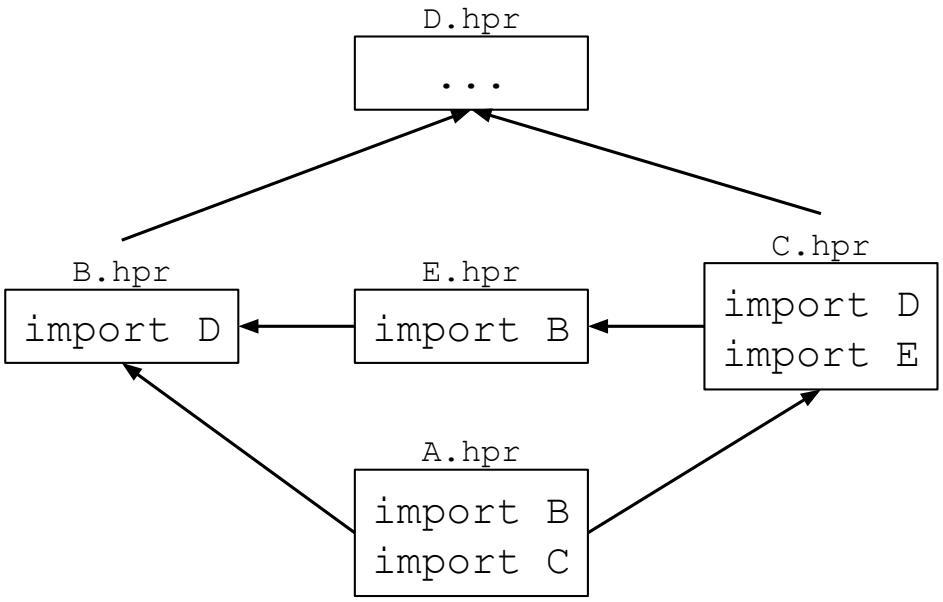

Figure 4.5: Example dependency graph

4.4.2 Interface files

In order to propagate type information for a given module's exports into the modules that depend on it, the concept of interface files is used. Hopper's interface files are similar in concept to those of Haskell's [19], but much more basic. They contain a text representation of maps from identifiers to type information (which might have been given explicitly or inferred by the type checker) for everything that is exported from a given module.

4.5 Type system

Hopper's type checker was designed to provide compile-time guarantees about program behavior on Erlang's virtual machine. To do this it uses a variant of the Hindley-Milner [20] type system upon which Haskell's system was initially based. While making Hopper's type system as powerful as Haskell's is today is beyond the scope of this project, Hindley-Milner still allows one to write useful typed programs. For example, one can define polymorphic data types and functions. To illustrate, figure 4.6 shows a definition of lists and the mapping function over it.

```
data List a = Cons a (List a) | Nil

map f (Cons x xs) = Cons (f x) (map f xs)
map _ _           = Nil
```

Figure 4.6: Definition of a list data type and the map function

The type for map is inferred; in general, type signatures do not need to be given to functions.

4.5.1 Implementation

The type checking module is based on the standard algorithm for Hindley-Milner type checking, Algorithm W. It works by taking a list of definitions as input: tuples of a name, a syntax tree that represents the value the name is defined to be and an optional type signature. The type checker then divides the definitions into those which have type signatures and those that do not. Definitions without signatures are further divided into groups of mutually recursive definitions. Those groups are strongly connected components (SCCs) in the functions' dependency graph. A dependency graph is a directed graph with names as nodes; if the definition of a value `n1` refers to `n2` then there is an edge from `n1` to `n2` in the graph. For example, in the module shown in figure 4.7, `noTypeSig` will be divided into its own group of mutually recursive values, `mutual1` and `mutual2` will be in the same and `withTypeSig` will be treated separately.

```
module Example where
noTypeSig = 3
mutual1 = mutual2
mutual2 = mutual1
withTypeSig :: String
withTypeSig = "I'm well typed!"
```

Figure 4.7: Example module with mutually recursive components

Once the definitions have been grouped, the next step is to infer the types of the definitions without type signatures. The mutually recursive groups are topologically

sorted by dependency. Therefore any SCC `A` which depends on another SCC `B` already having been inferred is type checked after `B`.

When a group of definitions `(f1 = e1;f2 = e2 ... fn = en)` is type checked and inferred, each name `(f1 ... fn)` is associated with a type variable. The type of each expression `(e1 ... en)` is inferred by following a number of inference rules. In Figure 4.8 such a rule is displayed. The example shows the mathematical notation for when a lambda expression of type $\tau \to \tau'$ is applied to a value of type τ. This yields a new value of type τ'.

$$\frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0\ e_1 : \tau'}$$

Figure 4.8: The application inference rule

These rules generate a number of equations over type variables, for example `t = a -> a, List a = List Bool`, which are solved via a process called unification. This process converts such systems to a constraint. A constraint is a system of equations where every left-hand side is a type variable.

Unification of two types is done by traversing both values and applying a rule for each form the type takes. For example, when unifying two type constructors (such as `Bool` and `List`) they are checked for equality. If they are not equal, unification fails. When unifying a type variable `t` with another type τ, the algorithm checks whether τ mentions `t`. If it does (and the type isn't just `t`) then `t` is an infinite type, which is not acceptable in Hindley-Milner. Otherwise, the equation `t =` τ is added to the constraint and type variables in the right hand side of every equation are expanded as far as possible. When unifying `(`τ_1 τ_2`)` and `(`τ_3 τ_4`)`, the pairs `(`τ_1`,` τ_3`)` and `(`τ_2`,` τ_4`)` are recursively unified. In the example `t = a -> a, List a = List Bool`, the solution would be `t = Bool -> Bool, a = Bool`. Examples of equations with no solution are `Int = Bool` and `t = List t`, which would generate an infinite type.

The type checker remembers the constraint and successively adds more information to it with every unification. When the types of `(f1 ... fn)` have been inferred to be of types `(t1 ... tn)`, the named values `(f1 ... fn)` will be assumed to be of those types during the rest of the type checking process. Once all SCCs have had their types inferred, the definitions with type signatures have their types inferred and unified with their signatures. If at any point in the type checking process a unification fails, the type checker fails with an error message.

4.5.2 Algebraic data types and constructors

The arity of a constructor depends on its type. For example, `Cons :: a -> List a -> List a` has arity 2. That means that a constructor's runtime representation is dependent on its type. To simplify the code generation step the conversion from type signature to function definition is done by the type checker. This means that the code generator does not have to handle ADTs.

In Hopper, ADTs are implemented and translated to Core Erlang in two primary cases: when they are used, and when they are declared. The implementation drew

inspiration from the book *Implementing functional languages: a tutorial* [21], which describes an approach to this problem where the values of ADTs are translated to tuples, holding the constructors (represented by unique integer values) as a tag, and their additional values. ADTs in Hopper are translated in the same manner, but translates the constructors to atoms instead to mimic the conventional structure of data in Erlang. Declared constructors are translated to functions that take a number of parameters and returns a tuple containing the constructor name and the arguments. Figures 4.9 and 4.10 displays how ADTs values are translated from Hopper to Core Erlang.

```
data Tree = Tree Number Tree Tree | Empty

Tree 5 (Tree 3 (Tree 1 Empty Empty)
               (Tree 4 Empty Empty))
       (Tree 7 Empty Empty)
```

Figure 4.9: Hopper ADT with example value

```
{'tree', 5, {'tree', 3, {'tree', 1, 'empty', 'empty'},
                        {'tree', 4, 'empty', 'empty'}},
            {'tree', 7, 'empty', 'empty'}}
```

Figure 4.10: Runtime representation of the Hopper ADT seen in figure 4.9

4.6 Code generator

Generally, the last step of a compiler is the code generation step. The code generator component should take a collection of ASTs and translate them to the target language. Additionally, this component may perform some optimizations on the generated code before it finishes.

4.6.1 Implementation

The code generator is implemented using the `Language.CoreErlang` [22] package. It contains a Core Erlang parser, a set of data types used for representing Core Erlang ASTs and a pretty printer for said data types. The code generator makes use of this package by building Core Erlang ASTs and then invoking the pretty printer on them to produce Core Erlang source code.

4.6.2 Function applications

A substantial difference between Haskell and Erlang is how functions are applied to arguments. In Haskell, all functions are of arity 1, and when functions are applied

to several arguments, they are curried during runtime. This means that functions can be partially applied, in which case they return a function. This is useful when for example there is a need for composing new functions from already defined ones in a terse manner.

Hopper attempts to mimic Haskell's support for partial application. This, in combination with the fact that the Erlang compiler requires all function definitions to have a determined arity at compile time, results in the translation of functions being one of the non-trivial tasks of the code generator.

This problem was solved by defining an Erlang function named `curry`, which turns all expressions either into functions of arity 1 or into values. The Core Erlang code produced by Hopper will contain calls to this function, resulting in functions being curried during runtime. The implementation of the curry function can be seen in Appendix B.

In figure 4.11 and figure 4.12 a simple Hopper function and the resulting Core Erlang code are shown respectively.

```
f :: Int -> String
f 0 = "False"
f 1 = "True"
```

Figure 4.11: Simple Hopper function

4.6.3 Emulating the Erlang compiler

One downside to the choice of compiling to Core Erlang and then invoking the Erlang compiler is the loss of additions and optimizations. To deal with this the code generator has been given additional functionality to emulate the behavior of the Erlang compiler prior to generating Core Erlang.

The currently implemented behavior is the generation of `module_info` functions that return information about the module when invoked, and an additional case to all case expressions which catches any unmatched case and throws a run time exception.

```
'f'/1 =
  fun(X@1) ->
    case {X@1} of
      {P_arg1} when 'true' ->
        case {P_arg1} of
          {0} when 'true' ->
            "False"
          {1} when 'true' ->
            "True"
          ({_cor1} when 'true' ->
             primop 'match_fail'({'case_clause', _cor1})
           -| ['compiler_generated'])
        end
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
    end
'__f'/0 =
  fun() ->
    apply call 'erlang':'make_fun'
               ('Prim', 'curry', 1)('f'/1)
```

Figure 4.12: Core Erlang translation of the Hopper function seen in figure 4.11

5
Results

The result of the project is the Hopper language and its compiler. The language is a clean basic functional programming language with indentation sensitive syntax featuring polymorphic ADTs, pattern matching and lambdas. The compiler also has a module system and some integration with Erlang. Not included in the final product is easy access to message passing through send/receive and full Erlang interoperability. The features of each part are presented in more depth below.

5.1 The Grammar

The final grammar for the language supports most of the necessary syntax for a pure functional programming language. The full grammar can be found in Appendix A. The main supported features are:

- Indentation sensitivity
- Function definitions and type signatures
- Polymorphic abstract data types
- Lambda and case expressions
- Primitive operators
- Pattern matching of primitives, tuples, constructors and variables
- Module definitions with imports and exports
- Qualification of functions, constructions and type constructors

Additional grammatical constructs were planned and designed but not implemented. Most of those features were various kinds of syntactic sugar, for example lists and tuples, but also syntax for simplifying repetitive tasks such as `IO` sequencing.

Some primitive operators, such as arithmetic and relative, were hard coded into the product. There were plans for implementing a more generic solution for this where users would be able to import any Erlang BIFs and functions as desired in a type safe manner.

A code example displaying some of the available features is available in Appendix C.

5.2 The Compiler

The Hopper compiler takes one or more `.hpr` files and compiles them one by one, and the modules they depend on if needed. Apart from this, it may also take a selection of command line options for producing different kinds of helpful output. Optional command line arguments and explanations of their functionality can be seen in table 5.1.

Shorthand flag	Full flag	Functionality
`-v`	`--verbose`	Print info about compilation progress
`-p`	`--parse`	Write the parse tree to a `.parse.hpr` file
`-a`	`--ast`	Write the abstract syntax tree to a `.ast.hs` file
`-t`	`--typecheck`	Write the type checked tree to a `.typed.hs` file
`-c`	`--core`	Write the generated Core Erlang code to a `.core` file
`-b`	`--nobeam`	Skip writing the target `.beam` file

Table 5.1: Optional arguments for the Hopper compiler

5.2.1 Modules

```
module Hop (f, g, h) where

import Tree.BinaryTree
import MyModel

...
```

Figure 5.1: Export listing and import statements

Hopper supports a module system by providing export listings for declaring interfaces and import statements for declaring dependencies. Cyclical dependencies are reported as errors for simplicity and as a way to enforce separation of concerns. Information about the modules are stored in dedicated interface files. Type information propagation is however broken in the current version. In figure 5.1 the syntax of exports and imports are shown, where `f`, `g` and `h` are exported functions.

The fact that the Hopper compiler generates BEAM modules allows for a degree of interoperability: Hopper modules may be integrated in applications written in Erlang in a transparent way. Some parts of the compiler are prepared for enabling Hopper code to make use of modules written using Erlang.

5.2.2 Type checker

The type checker is able to diagnose and report type errors in Hopper code as well as infer the types of definitions. It provides support for polymorphic ADTs such as lists and `Maybe` and polymorphic functions such as `map` and `fold`.

5.3 Missing features

The language and compiler has succeeded in implementing type safety run on the Erlang VM. Easy to use syntax is however missing for some key features. Specifically, the group desired that sending/receiving of messages and the spawning of processes would be intuitive and concise. The interoperability with Erlang was also not complete. The initial goals of accessing the concurrency and parallelism from Erlang have thus not been fully met.

6
Discussion

This section starts by discussing the results of the Hopper project. The implemented features for each compiler part will be listed together with comments on how the features raise the relative strength of the language and compiler. There will also be discussions on how the desired features that did not make it affect the finished prototype. After the results discussion follows sections handling overall code quality and the effects of the chosen development methods.

6.1 Grammar

Using the BNF Converter to generate a lexer and parser from our grammar was overall very successful. BNFC has all the features wanted for implementing the product. It halved the work by producing both a lexer and a parser which would otherwise have had to be written separately from scratch. It also has indentation sensitivity built in which is used to insert extra tokens to match against. The group had some problems at the beginning of the project to understand when this rule activated, but they were solved by studying the generated source code. There is a simple way to test if a problem in the grammar is due to either the layout rule or a grammatical rule. This is done by overriding the layout rule by inserting the tokens manually to check if the specified grammar still behaves correctly.

At the start of the project there were discussions in the group on whether or not to use BNFC to generate the lexer and parser. Choosing to get an easy start with the generated lexer and parser was accompanied by the notion that the group might want to switch to using Alex [15] and Happy [16] directly at a later stage. This did not come to happen during the project but might still be something to consider for the future. Implementing the group's own lexer and parser would mean greater control over the process.

The rewrite done half way through the project temporarily held up the introduction of new features. It was a necessary change however; new syntax could not otherwise have been added without ambiguity.

6.2 Modules

Several of the choices made while designing and implementing the module system were made with the goal of easy implementation, rather than the Hopper language being as useful as possible to the end user. However, those choices were made with

consideration as to whether it would be difficult to add in such features at a later stage.

Only qualified imports, in the Haskell sense of the word, are available since this meant the group did not have to worry about handling name conflicts which could occur if identifiers by the same name were imported from different modules. However, since there already is a step in the compiler where module-internal names are converted into their qualified form, this step could be modified to use a lookup table of imported identifiers.

The restriction to a single source tree is somewhat more grave, as it makes the concept of Hopper libraries very impractical. However, this restriction is reasonable given the scope of the project.

Another shortcoming in the module system is the fact that programs implemented using Hopper are always recompiled in their entirety. A possible future improvement would be to make the compiler recompile only those that depend on changed modules.

Despite these shortcomings the basic module system as implemented is, with the exception of some bugs, well functioning. It successfully allows for projects consisting of multiple modules without too much inconvenience for the programmer.

Another consideration when implementing the module system was whether or not to use interface files. As the compiler currently does not support partial recompilation, the interface files are recreated entirely each time the project is compiled. In theory, the information contained in the interface files could instead be propagated internally in the compiler process. However, the usage of interface files allowed for very simple implementation of dependency compilation. The main `compile` function is literally two lines of code: first generating a compile order and then compiling each file in order, as shown in figure 6.1. Further, a solution without interface files would have to be changed if partial recompilation was added in.

```
compile :: [Flag] -> FilePath -> IO ()
compile opts f = do
  fs <- dependencyCheck f
  forM_ fs (compileFile opts)
```

Figure 6.1: Compilation with dependencies

6.3 Type checker

Since Hopper was designed to be similar to Haskell in terms of its syntax and type system and Haskell's system is based on the Hindley-Milner type system, it was the obvious choice. The existence of relevant material [23] on implementing such a type checker eased development considerably.

In addition to implementing Hindley-Milner, the type system contains a number of interesting ideas. It implements a function which provides access to arbitrary Erlang functions given their name and arity as an argument. In a production version

of Hopper, this functionality would be wrapped and provided with a user friendly syntax. It also gives access to type safe and non type safe message passing in an experimental form, allowing for processes written in Hopper to both communicate in a type safe way with each other and with processes written in Erlang in a non type safe way. Hopper's experimental form of ad-hoc polymorphism allows for type safe sending of messages, but is less powerful than Haskell's and would likely be replaced in production Hopper.

6.3.1 Algebraic data types and constructors

The approach to translate ADTs to tuples tagged with atoms was deemed successful. The generated beam code operates as expected, and is fairly analogue with beam code generated from Erlang code from a run time point of view. However, if values generated with Hopper constructors were to be accessed from Erlang written code, it would prove tedious since the user would have to write out the tuple representations of the ADTs manually. In this case it would also be required that the user understands how Hopper ADTs are translated.

6.4 Code generator

The choice of Core Erlang as a target language proved favorable. The structure of a Hopper module and a Core Erlang module are similar enough that many translation cases are trivial. It also enabled the use of the `Language.CoreErlang` [22] package, which enabled generating the target code by building an AST and then invoking the included pretty printer. The fact that the Erlang compiler can take Core Erlang as input made this choice possible. Early in the project, BEAM instruction code was considered as the target language, but the idea was discarded since Core Erlang proved more suitable given the similarities in the ASTs, as well as the sparse documentation for BEAM.

6.4.1 Function applications

The goal for Hopper function applications was to make their behavior as close to Haskell's function applications as possible. This meant that the function applications written in Hopper would result in the function being curried at run time. One could argue that this adds overhead, but this overhead was deemed trivial, since it is relative to the arity of the function, which at any time should be small. In addition, two versions of each function are generated, one where the run time curried solution is used, and one where the full arity is used. The first case is used internally by Hopper, while the second one permits all Hopper functions to be used from Erlang, making it more interoperable with the language.

6.5 Code quality and user interaction

For the language and compiler to be considered useful in a context outside of the bachelors thesis there are some criteria which need to be fulfilled, for example descriptive error messages and good documentation. This section discusses choices made in regards to usability.

6.5.1 Documentation

A general consensus between group members to keep the code base well documented was not followed up or enforced by the group. This led to parts of the code being very well documented and others being less readable and understandable than preferred. This is something that could be fixed and should be if the Hopper language project is continued.

6.5.2 Testing

Plans for a thorough test suite were followed initially. However, large changes made to the compiler pipeline made maintaining the test code a challenge as tests quickly became outdated. A stronger focus on testing and maintenance could have led to more reliable code. When a more stable pipeline is up this could once again be a priority.

6.5.3 Usability

The intent was for the language and compiler to be user friendly with an easy to use syntax and descriptive error messages. For the resulting prototype of the project and the features it contains, the goal of an easy to use syntax is mostly fulfilled. The resemblance between Hopper code and Haskell code is evident and should make Hopper accessible to many functional programmers. Error messages and warnings were designed but not implemented. The project focused on getting the compiler working for correct code to begin with and did not reach a point where these additions could be made.

6.5.4 Interoperability

The fact that the Hopper compiler generates BEAM modules allows for a degree of interoperability: Hopper modules may be integrated in applications written in Erlang. Some parts of the compiler are prepared for enabling Hopper code to make use of modules written using Erlang. This would enable Hopper written code to access the already established Erlang library, and opens up for a future where Hopper written libraries may be accessed in Erlang written applications.

6.6 Methods discussion

This section discusses the methods used while developing the Hopper language and its compiler. A lack of initial structure would become apparent later in the project but the chosen path still proved to be productive. Going in to the project the group had limited knowledge of the problem area and of the capabilities of the group which made initial planning difficult.

6.6.1 Agile development - sprint length

Sprint lengths were initially short and had small incremental goals. This worked well for parts of the compiler where the structure was relatively flat and new content could be added without affecting old content to any greater extent.

One notable exception was the type checker where implementing a small subset proved to be a big challenge. Longer sprints would probably have seen the same difficulties though since they were due to differences between compiler parts more than the sprint lengths.

6.6.2 Lack of daily communication

Not having continuous follow-ups might have prolonged the difficulties in getting the full pipeline working. The weekly meetings had the long term plans running but not many insights into the detailed problems discovered in each part. Workshops were held to emphasize group work and communication channels were set up to have group members help each other between meetings. This all proved very helpful but some problems were still discovered a bit later than they might have been with more frequent face to face meetings. In hindsight there could not have been much done about the number of meetings with the six group members having different schedules. Better communication through the supplied channels might have been a good middle ground.

6.6.3 Lack of a project leader

No one person had the responsibility to catch on to potential problems or coordinate development. Most sections of the compiler were worked on by only a couple of team members, making it difficult for others to make modifications or understand the code. This led to considerable latencies any time a change across several different modules was required. Had a project leader coordinated the development of new features across the pipeline, that slowdown might have been avoided.

Further, appointing a technical lead responsible for specifying the interfaces between components (i.e. the data types representing the program and the type signatures of exported functions) could have made it easier for members to use code they had not themselves written.

Sharing the responsibility as a group felt natural going in to the project and not knowing each other made choosing a leader difficult. Having the group leader responsibility rotate between group members might have been a solution. Another

could have been to hold off on assigning a leader until the group got to know the problem, and each other, better.

6.6.4 Lack of a hard specification

Starting the project the group agreed on potential features that could be added to the language. A list of such features was compiled to use as a general direction in which to take development.

The incremental development approach that then followed, beginning with a small subset of the language, quickly lost track of these initial goals. This meant that group members mostly made their own decisions about what steps to take in the development of their areas of responsibility. For the most part this worked fine but in some cases it led to too much freedom and decisions that should possibly have been made as a group were made by fewer members. In hindsight, a more formal specification agreed on by the group might have had group members working more together and towards a common goal.

6.6.5 Revision control system

The use of Git [14] as our revision control system was successful. The ease of branching, working locally and merging was a good fit for the group dynamic employed.

There were times during the project when the number of branches rose a bit high. With six group members there should often not be a need for eight or nine simultaneous versions of the code. Efforts were made to merge branches and keep the number low with one branch hosting the main development version and separate branches for new features being developed.

6.6.6 Type checker branches

The type checking took two different directions early on which led to two separate branches. They were in some ways similar in concept but different in their approach to the subject. This led to time lost both through other compiler phases waiting for a functioning type checker and through one of the branches in the end being discarded in favor of the other branch.

The type checker development was in many ways not suited for the incremental growth that the project started with. Beginning from a smaller grammar and expanding it felt natural but there was no easy way of beginning with a small type checker.

Since the type checking was also a heavy theoretical subject there were many sprint cycles before any useful code was added to the type checker in the compiler pipeline. To reduce the effect of this on the development of the rest of the compiler a "mock" type checker was implemented which assumes all programs are fully and correctly typed, and does nothing. This allowed developing the other parts of the compiler independent of the progress on the type checker.

6.6.7 Responsibility rotation

Beginning with the small subset of Hopper there was a division into groups which became static due to problems with the type checker. The initial idea for the project was to have all group members' work on every part of the project. This would not only support the learning goals of the project but would also ensure that each member of the team wrote code in each of the modules. Rotations could have made it easier for team members to make their own modifications to any module when needed, thus avoiding large latencies and deadlocks caused by the need for alteration to some other compiler phase.

6.7 Social and economic impact

One of the main benefits of exposing Erlang functionality to more users, by adding clean syntax and type safety, is increased productivity. The Erlang environment is good for quickly producing functioning concurrent code and less time implementing means more time using and testing. A clean and readable syntax is also more understandable which could cut down time spent on reading code, which is a big part of programming.

A strong type system gives another increase in productivity. The cost of fixing bugs grow significantly for each step in the life of the software and finding any bugs early on is beneficial.

Through increased parallelism there might also be positive effects on energy use. More parallelism could see decreased clock speeds and less energy spent in total.

6.8 Future work

To ensure that the Hopper language and its compiler develops into a useful product, work beyond what was done in this project needs to be laid out. The next step that was planned in the project but not fulfilled was the support for accessing the concurrency features of Erlang. Experimental work done during the project was carried out and it showed that access to the concurrency features can be given with ease, however the implementation needs to be finalized. After this, a thorough cleanup of the code should take place together with fixes of the known bugs, and a robust and quality ensuring test suite should be implemented. When this is done, Hopper will have a stable foundation that will open up for future additions and improvements such as descriptive error messages, compile time optimizations and a richer library of built in functions.

7
Conclusion

We set out to see if it was possible to combine an expressive type system with native facilities for concurrent and parallel programming. As a model for our type system and language syntax we chose Haskell and for the concurrency and parallelism primitives we chose Erlang. Both are very competent languages in their respective area. From the results and discussion chapters it is clear that it is possible to implement a type system for a functional language that runs on the Erlang VM. It is however still not clear how to utilize the concurrent and parallel features of Erlang in this typed environment.

It is evident from the discussion that most of the shortcomings of the project originates from administrative errors and a lack of communication coupled with a tight time frame. A lot of time that could have been put into design, experimental work and implementation was instead put into integrating contradicting components, reworking features and converging of ideas. The time that was actually spent on feature development and experimentation proved fruitful.

The developed product, though somewhat limited, does what it is intended to do rather well. Its syntax is expressive and simple, it has a strong type system, and executing generated code produces the correct result for the examples we have written. There were a number of features designed and planned that did not make it into the final product, for example full support for message passing between processes written in Hopper and rich informative error output from the compiler. This adds to our belief that Hopper has potential.

We conclude that the future of Hopper is promising. Given time and resources, the programming language Hopper has the potential of growing into a useful alternative when developing applications for the Erlang virtual machine.

Bibliography

[1] Chris Lattner and Apple Inc. (2014) *Apple Swift Language* [online] Available at: `https://developer.apple.com/swift/` (retrieved 2015-05-12)

[2] Graydon Hoare and Rust Project Developers. (2010) *Rust Language* [online] Available at: `http://www.rust-lang.org` (retrieved 2015-05-12)

[3] haskell.org. (1990) *Haskell Language* [online] Available at: `https://www.haskell.org/` (retrieved 2015-05-13)

[4] Joe Armstrong. (1986) *Erlang Language* [online] Available at: `http://www.erlang.org/` (retrieved 2015-05-13)

[5] Joe Armstrong. (2007) *What's all this fuss about Erlang?* [online] Available at: `https://pragprog.com/articles/erlang` (retrieved 2015-05-12)

[6] José Valim. (2012) *Elixir* [online] Available at: `http://elixir-lang.org` (retrieved 2015-05-12)

[7] Robert Virding. (2008) *Lisp Flavored Erlang* [online] Available at: `http://lfe.io` (retrieved 2015-05-12)

[8] Jeff Epstein. (2011) *Cloud Haskell* [online] Available at: `http://haskell-distributed.github.io` (retrieved 2015-05-12)

[9] Dimitry Golubovsky. (2008) *York Haskell Compiler: Haskell on BEAMs* [online] Available at: `http://yhc06.blogspot.se/2008/05/haskell-on-beams.html` (retrieved 2015-05-12)

[10] Torbjörn Törnkvist. (2015) *Haskerl: Haskell subset compiled to the Beam machine* [online] Available at: `https://github.com/etnt/Haskerl` (retrieved 2015-06-01)

[11] Markus Forsberg, Aarne Ranta. (2005) *The Labelled BNF Grammar Formalism* [online] Available at: `http://bnfc.digitalgrammars.com/LBNF-report.pdf` (retrieved 2015-05-09)

[12] Carlsson, R. (2001) *An introduction to Core Erlang* [online] Available at: `http://www.it.uu.se/research/group/hipe/cerl/doc/cerl_intro.ps` (retrieved 2015-03-16)

[13] Scrum.Org and ScrumInc (2014) *The Scrum Guide* [online] Available at: `http://www.scrumguides.org/scrum-guide.html` (retrieved 2015-03-16)

[14] Linus Torvalds. (2015) *Git* [online] Available at: `http://git-scm.com` (retrieved 2015-05-12)

[15] Chris Dornan, Isaac Jones, and Simon Marlow. (2003) *Alex User Guide* [online] Available at: `https://www.haskell.org/alex/doc/alex.pdf` (retrieved 2015-05-09)

[16] Simon Marlow and Andy Gill. (2001) *Happy User Guide* [online] Available at: `https://www.haskell.org/happy/doc/happy.pdf` (retrieved 2015-05-09)

[17] Lesk, Michael E., and Eric Schmidt. (1975) *Lex: A lexical analyzer generator.*

[18] Johnson, Stephen C. (1975) *Yacc: Yet another compiler-compiler.* Murray Hill, NJ. Bell Laboratories.

[19] GHC Compiler Wiki. (2015) *Interface files* [online] Available at: `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/IfaceFiles` (retrieved 2015-05-12)

[20] Damas, L. and Milner, R. (1982) *Principal type-schemes for functional programs* [online] Available at: `http://web.cs.wpi.edu/~cs4536/c12/milner-damas_principal_types.pdf` (retrieved 2015-02-05)

[21] Peyton Jones, S. and Lester, D. (1992) *Implementing functional languages: a tutorial* [online] Available at: `http://research.microsoft.com/en-us/um/people/simonpj/Papers/pj-lester-book/` (retrieved 2015-05-05)

[22] Pérez, D. C. and Garzía, H. F. (2008) *The CoreErlang package* [online] Available at: `https://hackage.haskell.org/package/CoreErlang` (retrieved 2015-03-16)

[23] Peyton Jones, S. (1987) *The Implementation of Functional Programming Languages* [online] Available at: `http://research.microsoft.com/en-us/um/people/simonpj/papers/slpj-book-1987/start.htm` (retrieved 2015-03-16)

Appendix A

Hopper grammar specification

```
--
-- Layout
-- ======
-- Inserts "{" "}" around definition after the keywords below

layout "=", "::", "of" ;
layout toplevel        ;

--
-- Module
-- ======
--

entrypoints Module;
MMod. Module ::= "module" IdCon Exports "where" ";" [Import] [Def] ;

--
-- Exports
-- =======
-- NEmpty allows leaving the () out
-- NExp now allows exporting constructors

NEmpty. Exports ::=                  ;
NExps.  Exports ::= "(" [Export] ")" ;

NExp. Export ::= Id  ;
separator Export "," ;

--
-- Imports
-- =======
--

IImport. Import ::= "import" IdCon ;
separator Import ";" ;

--
-- Definitions
-- ===========
-- Moved out definitions to make them more flexible and reuseable

DFun.  Def ::= Func ;
DSig.  Def ::= Sign ;
DAdt.  Def ::= Adt  ;
separator Def ";"   ;

--
-- Functions
-- =========
--

FFun. Func ::= TIdVar [Arg] "=" "{" Expr "}" ;

--
-- Arguments
```

```
-- =========
-- Is just a pattern but allows to be empty

APat. Arg ::= Pat ;
terminator Arg "" ;

--
-- Expressions
-- ===========
-- Renamed from Exp to Expr to remove name clash with Prelude
-- Pattern in clauses now allows 'Just _' without parenthesis

EId.      Expr2 ::= Id                                    ;
EPrim.    Expr2 ::= Prim                                  ;
EOpr.     Expr2 ::= "(" IdOpr ")"                         ;
EInfix.   Expr1 ::= Expr1 IdOpr Expr2                     ;
EApp.     Expr1 ::= Expr1 Expr2                           ;
ECase.    Expr1 ::= "case" Expr1 "of" "{" [Clause] "}"    ;
EIf.      Expr1 ::= "if" Expr1 "then" Expr2 "else" Expr2  ;
ELambda.  Expr  ::= "\\" [Pat] "->" Expr                  ;
coercions Expr 2                                          ;

CClause. Clause ::= ClausePat "->" Expr ;
separator nonempty Clause ";"           ;

CCPPat. ClausePat ::= Pat         ;
CCPCon. ClausePat ::= IdCon [Pat] ;

--
-- Patterns
-- ========
--

PCon.   Pat ::= IdCon              ;
PVar.   Pat ::= TIdVar             ;
PPrim.  Pat ::= Prim               ;
PWild.  Pat ::= "_"                ;
PTuple. Pat ::= "(" [PatTuple] ")" ;
terminator nonempty Pat ""         ;

PTCon. PatTuple ::= IdCon [Pat] ;
PTPat. PatTuple ::= Pat         ;
separator PatTuple ","          ;

--
-- Signatures
-- ==========
--

SSig. Sign ::= IdVar "::" "{" [Type] "}" ;

--
-- Types
-- =====
-- If TypeTuple has one element, treat it as parenthethis
-- Replace TName and TVar with TId ::= Id ?

TName.  Type ::= IdCon [TypeArg]     ;
TVar.   Type ::= TIdVar              ;
TTuple. Type ::= "(" [TypeTuple] ")" ;
separator nonempty Type "->"         ;

TTTuple. TypeTuple ::= [Type] ;
separator TypeTuple ","       ;

TTAId.    TypeArg ::= Id                  ;
TTATuple. TypeArg ::= "(" [TypeTuple] ")" ;
terminator TypeArg ""                     ;

--
```

```
-- ADT declarations
-- ================
--

AAdt. Adt ::= "data" TIdCon [AdtVar] "=" "{" [AdtCon] "}" ;

AVVar. AdtVar ::= TIdVar ;
terminator AdtVar ""    ;

ACCon. AdtCon ::= TIdCon [AdtArg] ;
separator nonempty AdtCon "|"    ;

AAId.    AdtArg ::= IdCon                 ;
AAVar.   AdtArg ::= TIdVar                ;
AATuple. AdtArg ::= "(" [AdtArgTuple] ")" ;
terminator AdtArg ""                      ;

AATCon. AdtArgTuple ::= IdCon AdtArg [AdtArg] ;
AATArg. AdtArgTuple ::= AdtArg      ;
separator nonempty  AdtArgTuple "," ;

--
-- Identifiers and primitives
-- ==========================
-- To simplify lists without any separators

IdVarNQ. IdVar ::= TIdVar  ;
IdVarQ.  IdVar ::= TQIdVar ;

IdConNQ. IdCon ::= TIdCon  ;
IdConQ.  IdCon ::= TQIdCon ;

ICon. Id ::= IdCon ;
IVar. Id ::= IdVar ;
terminator Id ""   ;

IInteger. Prim ::= Integer ;
IDouble.  Prim ::= Double  ;
IString.  Prim ::= String  ;
IChar.    Prim ::= Char    ;

--
-- Tokens
-- ======
-- TIdVar: identifiers, starting with a lower letter
-- TIdCon: constructors, starting with a upper letter
-- IdOpr: operators

token TIdVar (lower (digit | letter | '_' )* ) ;
token TIdCon (upper (digit | letter)* )        ;
token IdOpr ( [ ".:-^*+=<>&%$!#%|/\\" ]+ )     ;

--
-- Qualified names
-- ===============
-- Have to be implemented using tokens in order to disallow spaces

token TQIdVar ((upper (digit | letter)* '.')+ lower (digit | letter | '_' )* ) ;
token TQIdCon ((upper (digit | letter)* '.')+ upper (digit | letter )* )       ;

--
-- Comments
-- ========
-- Can't handle nested comments, bug in bnfc

comment "--"      ;
comment "{-" "-}" ;
```

Appendix B

`curry` implementation

```
curry(F) when is_function(F) ->
         {arity,N} = erlang:fun_info(F,arity),
         curry(F,N,[]);
curry(X) -> X.
curry(F,0,Args)->curry(apply(F,lists:reverse(Args)));
curry(F,N,Args)->fun(X)->curry(F,N-1,[X|Args]) end.
```

Appendix C

Hopper example

Simple binary search tree implementation in Hopper.

```
module BST (empty, insert, delete) where

data Tree = Tree Number Tree Tree | Empty
data Bool = True | False

empty :: Tree
empty = Empty

isEmpty :: Tree -> Bool
isEmpty Empty = True
isEmpty _     = False

insert :: Tree -> Number -> Tree
insert Empty x = Tree x Empty Empty
insert (Tree v t1 t2) x =
  case x == v of
     True  -> Tree v t1 t2
     False -> if x < v
                 then (Tree v t1 (insert t2 x))
                 else (Tree v (insert t1 x) t2)

delete :: Tree -> Number -> Tree
delete Empty _ = Empty
delete (Tree v t1 t2) x =
  case x == v of
    True  -> deleteRoot (Tree v t1 t2)
    False -> if x < v
                then (Tree v (delete t1 x) t2)
                else (Tree v t1 (delete t2 x))

deleteRoot :: Tree -> Tree
deleteRoot (Tree v Empty t2) = t2
deleteRoot (Tree v t1 Empty) = t1
deleteRoot (Tree v t1 t2) = Tree (leftmostElement t2) t1 (delete t2 (leftmostElement t2))

leftmostElement :: Tree -> Number
leftmostElement (Tree v Empty _) = v
leftmostElement (Tree v t1 _) = leftmostElement t1
```

Appendix D

Core Erlang example

Core Erlang translation of Hopper code as seen in Appendix C.

```
module 'BST' ['__delete'/0, '__empty'/0, '__insert'/0, 'delete'/2,
              'empty'/0, 'insert'/2, 'module_info'/0]
  attributes []
'False'/0 =
  fun() ->
    case {} of
      {} when 'true' ->
        'false'
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
    end
'__False'/0 =
  fun() ->
    apply call 'erlang':'make_fun' ('Prim', 'curry', 1)('False'/0)
'Empty'/0 =
  fun() ->
    case {} of
      {} when 'true' ->
        'empty'
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
    end
'__Empty'/0 =
  fun() ->
    apply call 'erlang':'make_fun' ('Prim', 'curry', 1)('Empty'/0)
'Tree'/3 =
  fun(X@1, X@2, X@3) ->
    case {X@1, X@2, X@3} of
      {Px1, Px2, Px3} when 'true' ->
        {'tree', Px1, Px2, Px3}
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
    end
'__Tree'/0 =
  fun() ->
    apply call 'erlang':'make_fun' ('Prim', 'curry', 1)('Tree'/3)
'True'/0 =
  fun() ->
    case {} of
      {} when 'true' ->
        'true'
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
    end
'__True'/0 =
  fun() ->
    apply call 'erlang':'make_fun' ('Prim', 'curry', 1)('True'/0)
'delete'/2 =
  fun(X@1, X@2) ->
```

```
    case {X@1, X@2} of
      {P_arg1, P_arg2} when 'true' ->
        case {P_arg1, P_arg2} of
          {'empty', _} when 'true' ->
            apply '__Empty'/0()
          {{'tree', Pv, Pt1, Pt2}, Px} when 'true' ->
            case
            apply apply apply apply apply apply call 'erlang':'make_fun'
                                                ('Prim', '__apply',
                                                 0)()("erlang")("=:=")(2)(Px)(Pv)
            of  'true' when 'true' ->
                  apply apply '__deleteRoot'/0()
                              (apply apply apply apply '__Tree'/0()
                                                       (Pv)(Pt1)(Pt2))
              'false' when 'true' ->
                case
                apply apply apply apply apply apply call 'erlang':'make_fun'
                                                    ('Prim', '__apply',
                                                     0)()("erlang")("<")(2)(Px)(Pv)
                of  'true' when 'true' ->
                    apply apply apply apply '__Tree'/0()
                                            (Pv)(apply apply apply '__delete'/0()
                                                                   (Pt1)(Px))(Pt2)
                  'false' when 'true' ->
                    apply apply apply apply '__Tree'/0()
                                            (Pv)(Pt1)(apply apply apply '__delete'/0()
                                                                        (Pt2)(Px))
                  ({_cor1} when 'true' ->
                     primop 'match_fail'({'case_clause', _cor1})
                   -| ['compiler_generated'])
                end
              ({_cor1} when 'true' ->
                 primop 'match_fail'({'case_clause', _cor1})
               -| ['compiler_generated'])
            end
          ({_cor1} when 'true' ->
             primop 'match_fail'({'case_clause', _cor1})
           -| ['compiler_generated'])
        end
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
    end
'__delete'/0 =
  fun() ->
    apply call 'erlang':'make_fun' ('Prim', 'curry', 1)('delete'/2)
'deleteRoot'/1 =
  fun(X@1) ->
    case {X@1} of
      {P_arg1} when 'true' ->
        case {P_arg1} of
          {{'tree', Pv, 'empty', Pt2}} when 'true' ->
            Pt2
          {{'tree', Pv, Pt1, 'empty'}} when 'true' ->
            Pt1
          {{'tree', Pv, Pt1, Pt2}} when 'true' ->
            apply apply apply apply '__Tree'/0()
                                    (apply apply'__leftmostElement'/0()(Pt2))
                                    (Pt1)
                                    (apply apply apply '__delete'/0()
                                                       (Pt2)
                                                       (apply apply '__leftmostElement'/0()
                                                                    (Pt2)))
          ({_cor1} when 'true' ->
             primop 'match_fail'({'case_clause', _cor1})
           -| ['compiler_generated'])
        end
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
```

```
    end
'__deleteRoot'/0 =
  fun() ->
    apply call 'erlang':'make_fun' ('Prim', 'curry', 1)('deleteRoot'/1)
'empty'/0 =
  fun() ->
    case {} of
      {} when 'true' ->
        apply '__Empty'/0()
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
    end
'__empty'/0 =
  fun() ->
    apply call 'erlang':'make_fun' ('Prim', 'curry', 1)('empty'/0)
'insert'/2 =
  fun(X@1, X@2) ->
    case {X@1, X@2} of
      {P_arg1, P_arg2} when 'true' ->
        case {P_arg1, P_arg2} of
          {'empty', Px} when 'true' ->
            apply apply apply apply '__Tree'/0()(Px)(apply '__Empty'/0())(apply '__Empty'/0())
          {{'tree', Pv, Pt1, Pt2}, Px} when 'true' ->
            case
            apply apply apply apply apply apply call 'erlang':'make_fun'
                                                  ('Prim', '__apply',
                                                   0)()("erlang")("=:=")(2)(Px)(Pv)
            of  'true' when 'true' ->
                  apply apply apply apply '__Tree'/0()(Pv)(Pt1)(Pt2)
              'false' when 'true' ->
                case
                apply apply apply apply apply apply call 'erlang':'make_fun'
                                                      ('Prim', '__apply',
                                                       0)()("erlang")("<")(2)(Px)(Pv)
                of  'true' when 'true' ->
                    apply apply apply apply '__Tree'/0()
                          (Pv)(Pt1)(apply apply apply '__insert'/0()(Pt2)(Px))
                  'false' when 'true' ->
                    apply apply apply apply '__Tree'/0()
                          (Pv)(apply apply apply '__insert'/0()(Pt1)(Px))(Pt2)
                  ({_cor1} when 'true' ->
                     primop 'match_fail'({'case_clause', _cor1})
                   -| ['compiler_generated'])
                end
              ({_cor1} when 'true' ->
                 primop 'match_fail'({'case_clause', _cor1})
               -| ['compiler_generated'])
            end
          ({_cor1} when 'true' ->
             primop 'match_fail'({'case_clause', _cor1})
           -| ['compiler_generated'])
        end
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
    end
'__insert'/0 =
  fun() ->
    apply call 'erlang':'make_fun' ('Prim', 'curry', 1)('insert'/2)
'isEmpty'/1 =
  fun(X@1) ->
    case {X@1} of
      {P_arg1} when 'true' ->
        case {P_arg1} of
          {'empty'} when 'true' ->
            apply call 'erlang':'make_fun' ('', '__True', 0)()
          {_} when 'true' ->
            apply call 'erlang':'make_fun' ('', '__False', 0)()
          ({_cor1} when 'true' ->
```

```
            primop 'match_fail'({'case_clause', _cor1})
          -| ['compiler_generated'])
      end
    ({_cor1} when 'true' ->
       primop 'match_fail'({'case_clause', _cor1})
     -| ['compiler_generated'])
  end
'__isEmpty'/0 =
  fun() ->
    apply call 'erlang':'make_fun' ('Prim', 'curry', 1)('isEmpty'/1)
'leftmostElement'/1 =
  fun(X@1) ->
    case {X@1} of
      {P_arg1} when 'true' ->
        case {P_arg1} of
          {{'tree', Pv, 'empty', _}} when 'true' ->
            Pv
          {{'tree', Pv, Pt1, _}} when 'true' ->
            apply apply '__leftmostElement'/0()(Pt1)
          ({_cor1} when 'true' ->
             primop 'match_fail'({'case_clause', _cor1})
           -| ['compiler_generated'])
        end
      ({_cor1} when 'true' ->
         primop 'match_fail'({'case_clause', _cor1})
       -| ['compiler_generated'])
    end
'__leftmostElement'/0 =
  fun() ->
    apply call 'erlang':'make_fun'
        ('Prim', 'curry', 1)('leftmostElement'/1)
'module_info'/0 =
  fun() -> call 'erlang':'get_module_info' ('BST')
'module_info'/1 =
  fun(_cor0) -> call 'erlang':'get_module_info' ('BST', _cor0)
end
```