An In-Depth Analysis of Haskell

K31 Compilers, Spring 2023

Konstantinos Chousos*, Konstantinos Kordolaimis†, Anastasios-Phaedon Seitanidis‡, and Aggelos Tsitsoli§

Department of Informatics and Telecommunications,
National & Kapodistrian University of Athens

May 26th, 2023

*Student ID: 1115202000215
†Student ID: 1115202000091
‡Student ID: 1115202000179
§Student ID: 1115202000200

Contents

List of Listings

List of Tables

Abstract

This report provides an in-depth look at the functional programming language Haskell. The first part covers the background of the language including its history and popularity, as well as its general position in the history of functional programming. It also delves into the various tools written in Haskell and its use cases in the tech industry. In the second part, control structures, laziness, identifiers, keywords, operators, lambdas, monads, memory management, namespaces, parallel programming, metaprogramming, and object-oriented properties of Haskell are discussed. The report concludes with several examples and code snippets, including a "Hello, World!" program and a factorial implementation.

1 Introduction

1.1 History of Haskell

Haskell was born in the 1980's. It was a child of necessity. Its purpose was to be the answer to an ever-growing problem: The plethora of functional programming languages, among which there were enough differences to distinguish them, but not enough for any one to be unique enough [Mar10]. This led to functional programming being discarded all together by many, as there didn't exist any kind of common ground between the different languages and implementations. Haskell's goal was — and still is — to be the pinnacle of the functional programming space. A language that would embody the functional programming principles and with capabilities suitable for real-world applications.

1.1.1 Prehistory

Functional programming first started gaining popularity around 1978, when John Backus — the leader of the team that developed Fortran [Bac+57] and the inventor of the Backus Naur Form (BNF) — published the famous *"Can Programming Be Liberated from the von Neumann Style?"* lecture [Bac78]. It presented functional programming as something useful in real world situations and applicable to tangible problems, as opposed to simply an academic and theoretical tool.

That's not to say functional programming was new. In fact, it had been around for at least 30 years at the time. Some important milestones in its history are the following:

- 1950's

  John McCarthy created Lisp [McC60], which was the pioneering high-level functional programming language at the time, during his time at the Massachusetts Institute of Technology in the 1950s. The definition of Lisp functions involved the use of Alonzo Church's lambda notation (see Functional programming and lambda calculus), which was expanded to enable the creation of recursive functions. Lisp was the first functional programming language to present several new exemplary features in the functional programming world. Lisp merged them, and as a result enabled the support of multiple programming styles. Subsequent dialects of LISP — e.g Common Lisp, Racket, Scheme and Clojure [Ste84; Fel+15; Abe+98; Hic20] — aimed to streamline LISP and make an easier version of it by emphasizing a purely functional foundation.

  At the same time period another functional programming language was created with the name Information Processing Language (IPL) from Allen Newell, Cliff Shaw and Herbert A. Simon [NT60]. IPL was created with the primary purpose of handling lists of symbols, enabling dynamic memory allocation and multitasking. It was the inventor of the concept of list processing.

- 1960's

  During the early 1960's, Kenneth E. Iverson was responsible for developing APL [Ive62]. APL primarily revolved around arrays that are multidimensional. By utilizing a wide variety of unique graphic symbols, APL enabled the creation of incredibly succinct code. APL and its creation was the inspiration for the development of other

languages in later years e.g. the programming language K which was created in the 1990's by Arthur Whitney.

During the mid-1960's, Peter Landin played a key role in the development of functional programming languages. He was responsible for the creation of the SECD machine [Lan64]. In addition to creating the SECD machine, Peter Landin recommended the ISWIM programming language [Lan66].

- 1970's

  During the 1970's, several functional programming languages were developed in Scotland. Robin Milner was the creator of ML. David Turner was responsible for the creation of SASL. Also at that period in Edinburgh, Burstall and Darlington created the NPL which was a functional programming language that utilized Kleene Recursion Equations [Dar77]. Later, the previous two developers with Sannella implemented polymorphic type checking from ML into Hope, a language inspired by NPL. As time passed, ML evolved into some new idioms, with OCaml being one of the most popular.

  Another functional programming language that was developed at that time was Scheme by Gerald Jay Sussman and Guy L. Steele [Abe+98]. Scheme was the initial variation of Lisp that mandated optimization in functions calls, which were recursive and happened in the rear position, also implemented lexical scoping.

  It was at this time that John Backus's lecture was published [Bac78]. He described functional programs as being constructed hierarchically using "combining forms". This type of construction was inspired from the principle of compositionality.

- 1980's

  During the 1980's, the intuitionistic type theory, developed by Per Martin-Löf, established a connection between functional programs and constructive proofs (these were expressed through dependent types).That opened up new avenues for collaborative theorem proving and had a significant impact on the evolution of later functional programming languages.

  David Turner created the lazy functional language Miranda [Tur85] in the mid 1980's. This language in particular had a significant impact on the development of Haskell (see Haskell and Miranda).

Functional programming and lambda calculus In 1932 and 1933, mathematician Alonzo Church published two papers both titled "A Set of Postulates for the Foundation of Logic" [Chu32; Chu33]. In those papers a new formal system of mathematical logic was introduced that centered around the notion of functions, with the name of *lambda calculus*. The purpose of using this framework was to represent calculation through function generalization and application, using binding between variables and variable substitution. Functional programming was invented as a practical application of lambda calculus, in order to benefit from its many uses.

It is known that lambda calculus was the basis for a lot of functional programming languages. Specifically, it gives a foundation for comprehending the nature of functions and it also provides the ability of combining many functions together to create a new function

(also known as function composition). Functional programming employs functions as the essential units for constructing programs, and lambda calculus furnishes a means to precisely declare and contemplate these functions [pan23].

Also, lambda calculus characterizes a function as a statement that accepts an input and generates an output. Their definition is based on lambda abstractions. Lambda abstractions are nameless functions that connect a variable to a calculation. Also they are employed to represent functions.

Many languages such as Haskell were directly influenced by lambda calculus. The former offers tangible realizations of the concepts introduced by the latter. For instance, using lambda expressions you can create functions and you can combine functions through higher-order functions.

The *lazy* craze 1970–1980 was a time period in which *lazy evaluation* (see 2.2) was getting more and more attention. It had been independently invented at least three times and there were many tries in implementing programming languages that supported it as a feature [Hud+07].

This "movement" was one of the reasons the *Functional Programming Languages and Computer Architecture* (FPCA) conference was established, which became a major conference in this field, held every other year.

FPCA was only one out of many conferences held with focus on functional programming and laziness in it. Another notable conference was the *Lisp and Functional Programming* (LFP) conference. In general, the field gained a lot of traction with major research papers being published and a lot of new ideas and breakthroughs being developed.

1.1.2 Origins

The year is 1987. The biennial FPCA conference is being held in Portland, Oregon. A meeting is organized during the conference — initiated by Peyton Jones and Paul Hudak — with the goal of gaining interest for creating a new functional programming language. This language would be the common ground among all of functional programming [Hud+07].

Haskell and Miranda At the time the committee hadn't yet given a name to the language. Also, not many talks for the structure of the language had happened. The result of this gathering was that the most feasible approach to get started with the creation of a language was to modify a language that already existed according to the committee's needs .

The programming language Miranda [Tur85] which was created by David Turner, came out as the most suitable option due to its efficiency. As Turner was absent from the conference, the committee decided that it would be fair to ask for his permission to use his language as the foundation of the language they were about to create. Although many talks were held for that reason the creator of Miranda did not give his permission.

The committee's objective was to develop a language that could serve various purposes, including researching language features, allowing unrestricted extension, modification, implementation, and distribution of the language. On the other hand, Turner had a strong dedication to upholding a unified language standard and didn't want multiple flavors of Miranda to exist [Hud+07]. He expressed his preference for avoiding the creation of another

Miranda and asked the committee to make a new language different enough to avoid confusion. So the committee had to make a new language from scratch. Although, the fact that the committee wasn't allowed to use features from Miranda made them take another approach in order to design the language. Except from the different ways Haskell and Miranda were created, the committee was greatly influenced by the latter.

After the previous incident, the committee used the mailing list `fplangc@cs.ucl.ac.uk` to discuss what they would do next and used the title "FPlang Committee" as a placeholder till a name wad found [Hud+07].

Figure 1: A timeline of the history of Haskell, its creation and its development [Hud+07].

The committee meetings The first meeting was held at Yale during 1988. There, a clear list of Haskell's goals was defined (Haskell's goals) and a name was decided, that was although later changed (Haskell's name).

Haskell's name During the meeting, the members of the development team were encouraged to suggest names. The list included names such as "Semla", "Vivaldi", "Mozart", and more. After conversation each member was given the opportunity to mark a name they didn't prefer. Eventually, only one name remained, and it was decided that the language would be named after the mathematician Haskell B. Curry. Specifically the creators decided to keep the name of "Haskell" as the final name [Hud+07].

At first the chosen name was "Curry". But after a while, the committee came to the realization that they would have an overabundance of comical references regarding the name [Hud+07]. Although there were various amusing jokes that could be made such as the spice carry, that is used in cooking, they were genuinely unsettled by the possibility of using Tim Curry as a way to make fun of the language. As Jon Fairbairn's abstract machine was referred to as TIM and Tim Curry was well-known for his role in the *"Rocky Horror Picture Show"*, they opted to steer clear of this direction. After additional deliberation the following day, they ultimately opted for the moniker "Haskell" for their novel programming language.

Haskell's goals At first the committee had to clearly set the purposes of the language. January 9th through 12th marked one very important gathering in which the following were decided [Hud+07]:

1. The language should be appropriate for educating, doing inquiry, and lastly implementations, such as constructing big structures .
2. Analytic syntax and significations should be presented in order to provide the complete description of the language.
3. The language must be accessible to all, and everyone should have the freedom to utilize and share it without any restrictions.
4. It should be applicable as a foundation for additional language exploration and investigation.
5. The language's concepts should be founded upon commonly accepted ideas and principles.
6. The language should aim to minimize superfluous discrepancies among functional programming languages. To achieve this, the committee's preliminary decision was to derive it from an already established language, namely OL.

The last two goals were proof that the original vision of Haskell was to rely on battle-tested methods and patterns and only unite the separated functional programming world. But this vision was short-lived, since neither OL was used as a base and Haskell introduced a myriad of new concepts in programming.

The Haskell Report In a meeting of the committee in Glasgow during 1988, it was decided that Hudak and Wadler would be responsible for editing the *first Haskell report*. This report would serve as a specification for the language. Its name was "Report on the Programming Language Haskell, A Non-strict, Purely Functional Language". It was finally published in April 1st of 1990, with version number 1.0.

Its next 1.1 revision was published in August of 1991 and 1.2 in March of the following year. 1.3 was published in 1996, with an added *Library report*. 1.3 was the version which introduced *monads* and monadic I/O. 1.4 was published in 1997, and two years after the *Haskell 98 report* was released. This version was a big milestone for Haskell. It coincided with the disbandment of the Haskell committee. It was later revised and republished as a book in December of 2002, also being freely accessed online [Hud+07].

The latest Haskell report was published in 2010 and is considered the current definition of the Haskell language [Mar10; Has23].

1.2 Haskell's popularity

Haskell's popularity has been recorded by two major programming languages popularity indexes, the TIOBE and PYPL indexes [Car23; TIO23].

According to the TIOBE index, as of May of 2023 Haskell takes the 36th spot in programming language popularity [TIO23]. The TIOBE index is created and maintained by TIOBE Software BV, a company based in Eindhoven, the Netherlands. According to their *About Us* page, they specialize in measuring code quality in software. The TIOBE index's ratings are based on the number of courses, jobs and users a given programming language has, and also many popular search engines such as Google and Bing.

The PYPL index ranks Haskell in the 28th spot, as of May of 2023 [Car23]. Haskell's evolution in popularity from 2005 until today is shown in figure 2. The PYPL (PopularitY of Programming Language) index is based on the number of programming language specific tutorials are searched on Google [Car23].

Also of note is Haskell's subreddit[1], which according to `https://subredditstats.com/r/haskell` has around 74,500 subscribers and an all around upward trend.

Figure 2: Haskell's worldwide popularity since 2005 in logarithmic scale [Car23]

1.3 Haskell's use cases

1.3.1 Spam Filtering at Facebook

Meta's secret weapon against spam and malware is a system called Sigma. Its job is to identify malicious actions on Facebook such as spam, phishing attacks, posting links to malware, etc. Threats detected by Sigma are removed automatically so they do not show up in the user's feed.

[1] `https://www.reddit.com/r/haskell/`

Meta recently completed a full redesign of Sigma, which involved replacing the in-house(custom) FXL language formerly used to power Sigma with Haskell. The Haskell-powered Sigma can now serve up to one million requests per second [Mar15]. It is also important to note that the team behind Sigma's development also made several improvements to `ghc`, making it possible for Sigma to achieve better performance from Haskell compared to the previous implementation.

How does Sigma work? Sigma is a rule engine, which means it runs a set of rules, called *policies*. Every interaction on Facebook from posting a status update to clicking "like" results in Sigma evaluating this action with a set of policies specific to that type of action. These policies make it possible for Meta to identify and block malicious interactions.

Why Haskell? The original language that Meta designed for writing its policies, FXL, was not ideal for expressing the growing scale and complexity of Facebook policies. It lacked certain abstraction capabilities, such as user-defined data types and modules, and its implementation, based on an interpreter instead of a compiler, was rather slow. This is why they decided to migrate to an existing language [Mar15].

These are the features that were at the top of the team's list when choosing a replacement language:

1. Purely functional and strongly typed.
2. Automatic batching and ability to overlap of data fetches.
3. Push code changes to production in minutes. The fleet must get the updated-compiled version of Sigma as fast as possible.
4. Performance.

Haskell measured up quite well for the specific demands.

1.3.2 Cardano

Cardano is a blockchain platform that aspires to provide a secure and scalable infrastructure for creating decentralized applications and executing *smart contracts*. It was founded by a group of researchers, engineers, and academics, including experts in functional programming languages like Haskell. The core software components of Cardano, including the blockchain protocol and smart contract infrastructure, are all written in Haskell [Hei21].

Haskell plays a key role in the development of Cardano, powering various aspects of its *architecture* and *functionality*. Here are some key areas where Haskell is utilized in Cardano:

- Blockchain Protocol: The central blockchain protocol, also known as **Ouroboros**, is a proof-of-stake (PoS) consensus algorithm that guarantees the security and integrity of the blockchain. Haskell's ability to specify complex protocols make it well-suited for implementing the *consensus logic*.
- Smart Contract Platform: The smart contract platform, called **Plutus**, allows developers to write smart contracts using functional programming techniques. Another popular choice for writing smart contracts is the Solidity language.

- Daedalus Wallet: **Daedalus** is Cardano's official wallet. Daedalus provides users with a secure and user-friendly UI to manage their **ADA** (Cardano's native cryptocurrency) and use the blockchain.
- Formal Verification: Haskell's support for formal methods and formal verification techniques. Formal methods involve mathematically proving the correctness of algorithms and protocols. Cardano wields formal methods to enhance its blockchain implementation.

Cardano benefits from the wide range of Haskell libraries and tools that are open-source. These libraries provide support for cryptography, networking, parsing, and other essential functionalities. The use of Haskell in Cardano reflects the language's suitability for building secure, high-assurance-availability systems.

1.3.3 Hasura

Hasura is an open-source, real-time GraphQL engine that allows developers to quickly and easily build scalable, performant APIs for their applications [Has]. Developers can easily connect their databases to GraphQL, allowing them to access their data through a single API. Of course, it supports a variety of databases, including PostgreSQL, MySQL, Microsoft SQL Server, and Oracle. It provides a powerful set of features that make it easy to manage complex database schemas, including automatic schema stitching, real-time data synchronization, and granular access control.

One of the key benefits of using Hasura is its ability to generate a complete GraphQL API from an existing **database schema**. This means that developers can get up and running quickly without having to write any code, and they can easily update their APIs as their data schemas change over time. A rich set of tools for monitoring and debugging GraphQL APIs are provided, including query performance analysis and real-time tracing. It is highly customizable and extensible, with a large ecosystem of plugins and integrations that make it easy to integrate with other tools and services.

It has quickly become a popular choice for developers looking to leverage the power of GraphQL.

Haskell is a key component of Hasura's architecture. Hasura is built using Haskell, a language known for its strong type system, which helps catch errors at compile-time, and its support for lazy evaluation, which allows for efficient handling of large datasets. In general, Haskell is well-suited for building high-performance, scalable systems. For example, Hasura uses the Postgres database library for Haskell to provide seamless integration with a variety of databases.

1.3.4 Hledger

Hledger is a command-line tool for double-entry accounting, written in Haskell. It is part of the larger Plain Text Accounting paradigm[2]. It is cross platform. It can import from and export to various data formats such as CSV or TSV. Also, it has multiple choices for a User Interface: It can be used from the command-line, a web browser, on mobile, and has various editor/IDE plugins [Mic23].

[2] https://plaintextaccounting.org/

1.3.5 Pandoc

Pandoc is a Haskell library used for markup conversion. It is a powerful command-line tool that can inter-convert various file formats, e.g. DocX to PDF, LaTeX to Markdown, Jupyter Notebook to HTML etc. An interesting feature is its Pandoc *filters*, which can be written directly in Haskell and provide the ability to modify the intermediate Abstract Syntax Tree (AST) of the conversion [MKR23][3].

1.4 Tools and frameworks for developing in Haskell

1.4.1 Hackage

Hackage is a key part of Haskell's ecosystem and provides a convenient way for Haskell developers to find and share code [Hac23a]. Hackage is a central repository of open-source Haskell packages, similar to other package managers such as PyPi for Python or npm for JavaScript.

Hackage contains thousands of Haskell packages, each one provides a set of modules and functions that can be used in other projects. Packages are identified by a unique name and version number, and are organized into categories such as "Web", "Database", "Parsing", and so on. It provides a way to effortlessly download and install Haskell packages, including their dependencies. Developers can upload their own packages, making them available to everybody. Uploading a package to Hackage involves creating a Cabal (see 1.4.8) file that describes the package and its dependencies, and then using Cabal to upload the package. Hackage uses a versioning system that allows multiple versions of a package to coexist. Thanks to the versioning system, developers can update their packages without breaking existing projects that depend on them.

1.4.2 Haddock

Haddock is a tool used for generating documentation from specially formatted comments in Haskell code [Had23]. It is named after William Haddock, a British naval officer. Haddock markup is the syntax used for writing these comments, which is similar to HTML but specifically designed for documenting Haskell code. The generated documentation provides information about the functions, types, and modules defined in the code, as well as usage examples. The documentation can be generated in several formats, including HTML, LaTeX, and Hoogle, a search engine for Haskell libraries.

Haddock is included in the Haskell Platform, a collection of tools and libraries for Haskell programming. It is also available as a standalone tool for use with other Haskell compilers and build systems.

Using Haddock for documentation is essential for ensuring that the codebase is well-documented and easily understandable. It helps to maintain code quality and encourages best practices in code development. Haddock is widely used in the Haskell community.

[3] Pandoc was also used extensively in the writing of this document for converting Markdown to Org-Mode, which was later exported to LaTeX and then compiled to this PDF document.

1.4.3 HLint

HLint is a useful tool for analyzing and optimizing Haskell code, designed by Neil Mitchell in 2006 [Mit23]. It is an open-source project available under the BSD-3-Clause license. The main objective of HLint is to help Haskell developers write better code that is more efficient, readable, and maintainable. It achieves this by analyzing the code and identifying patterns that can be simplified or improved. Consequently it produces suggestions for refactoring the code, in order to enhance its quality and readability. Developers can use HLint to analyze individual Haskell files or complete projects. It can be easily integrated into their workflow by incorporating it with a range of tools, such as IDEs and text editors. Furthermore, it provides a web interface that can be used to search and browse through the suggestions offered. It also supports customizable rules and configurations that can cater to various coding styles and preferences.

1.4.4 HUnit

HUnit is a unit testing framework and a key part of the broader Haskell ecosystem [23e]. With HUnit, you can define test cases as functions that assert expected values against actual results. These test cases can be grouped together into test suites, which make it easy to organize and run multiple tests at once. HUnit provides a set of assertion functions that allow you to check conditions such as equality, inequality, and expected exceptions.

```haskell
import Test.HUnit

-- Define the test cases
testAddition :: Test
testAddition = TestCase assertEqual "Addition test" 6 (2 + 4)

testSubtraction :: Test
testSubtraction = TestCase assertEqual "Subtraction test" 10 (30 - 20)

-- Create a test suite and include the test cases
tests :: Test
tests = TestList [testAddition, testSubtraction]

-- Run the tests using the runTestTT function
main :: IO ()
main = runTestTT tests
```

Listing 1: Simple example of a test case using HUnit

In listing 1, `testAddition` is a test case that asserts the expected value 6 against the result of the addition 2 + 4. The `assertEqual` function is used to perform the equality check. As you can see, the `testSubtraction` test case performs a similar check. The `tests` function defines a test suite that includes the `testAddition` and `testSubtraction` test

cases. Finally, the main function runs the tests using the `runTestTT` function, which prints the results to the console.

By leveraging HUnit, Haskell developers can automate the testing process and easily detect bugs and errors in their code. Unit tests help ensure the reliability and correctness of software, providing confidence.

1.4.5 QuickCheck

QuickCheck is a library for automated testing of Haskell programs [CH]. It was developed by Koen Claessen and John Hughes and is based upon the concept of property-based testing. Traditional testing involves writing test cases that check for expected outputs given specific inputs. In contrast, QuickCheck focuses on defining properties that must always hold true for your program, the input. It then generates random inputs and checks that these properties hold true for each one. For example, if you're testing a function that sorts a list of integers, you could define a property like "sorting a list should produce a list in ascending order". QuickCheck will generate random lists of integers and check that this property holds for each one.

To use QuickCheck, you define properties using the property function, which takes a `Boolean` expression that should hold true for all possible inputs. You can also use generators to define the types of inputs that it will generate. For instance, you can define a generator for lists of integers or for strings. Once you have defined your properties and generators, you can run it using the quickCheck function. QuickCheck will report any failures and reduce input values to find the minimal counterexample.

QuickCheck is a powerful testing tool that can help you find bugs and edge cases that you might miss with traditional testing. By generating random inputs, it can help you test your program more thoroughly and find unexpected issues that you may not have considered. It's a great addition to any Haskell developer's testing toolkit.

1.4.6 Yesod

Yesod is a powerful and flexible web framework for Haskell that is designed to be high-level, type-safe, and scalable [Yes23]. It provides a powerful set of tools and abstractions for building web applications. It also provides a type-safe DSL (Domain-specific language) for defining routes in your application. With its help you can be sure (at compile-time) that the URLs of your application are valid and that the provided parameters are correct. There is an ORM (Object-Relational Mapping) library included called **Persistent**, which allows you to interact with a database in a type-safe and composable way. Persistent supports several different database backends, including PostgreSQL, MySQL, and SQLite.

Yesod has a type-safe library for defining and rendering HTML. Additionally, it has built-in support for user authentication and session management. Its libraries support automatic validation, error reporting, password hashing, CSRF protection, and other security-related tasks.

1.4.7 Happy *and* Alex

Happy and Alex are two popular tools used in Haskell programming for generating parsers and lexers respectively. They greatly simplify the process of handling complex input formats by automatically generating the necessary code based on provided specifications of a grammar.

They are two different tools that are usually used together. Happy, also known as the Happy Parser Generator, is designed for generating parsers in Haskell [23d]. It takes a specification file and generates a Haskell module able to parse input according to the specified grammar. Happy has support for both LALR(1) and GLR parsing algorithms, providing flexibility in handling different grammatical constructs. Additionally, it allows programmers to specify semantic actions to be executed during the parsing process, enabling the manipulation and interpretation of parsed data.

Alex, or the Alex Lexer Generator, is used for generating lexical analyzers in Haskell [23a]. It takes a specification file containing regular expressions and associated actions and generates a Haskell module able to perform lexical analysis on input text. Alex employs efficient algorithms, such as finite automata and lazy evaluation, to generate high-performance lexers. Lexical analysis amounts to breaking the input into tokens, which are subsequently consumed by a parser generated by Happy.

By separating the concerns of parsing and lexing from the rest of the code, Happy and Alex allow developers to focus on the core logic of their programs. They automate the generation of code that can handle complex input structures, saving developers time and effort. Moreover, the generated parsers and lexers are highly optimized, contributing to the overall performance of the Haskell programs.

1.4.8 Cabal

Cabal is a build system and package manager for Haskell [23b]. It provides a way to manage dependencies and build Haskell projects, making it easier to develop and distribute Haskell software. In Cabal, a package is a collection of Haskell *modules* and other files that can be compiled into a library or executable. Each package is identified by a unique name and version number. Dependencies are declared in a package's Cabal file, which lists the names and version ranges of the required packages. There is a build system that can compile Haskell code into executables and libraries. It uses the GHC compiler by default, but can also work with other compilers such as Hugs or JHC. Cabal also integrates with Hackage (see 1.4.1), the central repository of open-source Haskell packages.

Cabal supports sandboxing, which allows you to create isolated environments for your projects. This can be useful for testing and development, since it ensures that dependencies are not shared between projects and avoids version conflicts. It is similar to Anaconda for Python.

Cabal is widely used in the Haskell community and is a key part of the Haskell development toolchain.

1.4.9 The Haskell Tool Stack (Stack)

Stack is a popular build tool for Haskell that was originally created to solve problems that developers were facing with Cabal (see 1.4.8), another Haskell build tool [23c].

The main focus of Stack's design point is reproducible builds. If you run `stack build` today, you must get the same result as running `stack build` tomorrow. To simplify this process, Stack uses curated package sets called snapshots.

One of the biggest advantages of Stack is that it provides deterministic builds. This means that if you build your project with Stack, you should get the same results every time, regardless of the environment you're building in. This makes it easier to reproduce builds across different machines and ensure that your code is consistent. Stack manages dependencies using snapshots, which are collections of packages that have been tested together and are sure to work together. This means that the dependencies you're using are compatible with each other, reducing the risk of conflicts. It is important to note that Stack has built-in support for caching compiled packages and dependencies. Stack supports multi-package projects. This can be useful if you're working on a large project that is split into multiple packages.

Stack is relatively easy to set up and use. It is easy to get started with a new project using the `stack new <project-name>` command. Finally, Stack integrates well with other Haskell tools such as Haddock and GHCi.

2 Analyzing Haskell

2.1 Control Structures

Usually functional languages like Haskell do not have constructs to change the execution path or to provide loops. Languages of this category try to follow mathematical notation for function definition and as a result they provide multiple definitions of the same function to cover different input cases. This approach requires that the language has pattern matching capabilities. However, some functional languages like Haskell provide some control constructs to facilitate programmers' work.

2.1.1 Pattern Matching

Pattern matching is a mechanism used to distinguish a structure of variables and constructors (either predefined or user-defined), called *pattern* [Mar10, chapter 3.17].

A given value fulfills a pattern if there is a substitution of its variables so that the instantiated pattern evaluates to the given value. Pattern matching can be used to provide multiple definitions of a function or within a case expression

Case Expression In Haskell, the `case` expression is a control structure used for pattern matching and branching based on the structure of a value. It helps the language to handle pattern matching of complex expressions or to express multiple possible outcomes in a single expression [Kre15].

The syntax for the `case` clause is shown in listing 2, and in listing 3 the `if-then-else` clause implemented with `case` is shown.

```
case expression of
    pattern1 -> result1
    pattern2 -> result2
    ...
    patternN -> resultN
```

Listing 2: `case` clause

```
case expression of
    True  -> result1
    False -> result2
```

Listing 3: `if-then-else` clause using `case`

2.1.2 Guards

Guards is a control construct that is used to specify conditions to be satisfied for a portion of a program. Guards are typically used with function declarations as follows and they are represented by a pipe (`|`) [Kre15]. An example is shown in listing 4 (`otherwise` is just an alias to `True`).

```
factorial :: Int -> Int
factorial n
    | n < 0     = error "Given number: negative -> factorial: undefined."
    | n == 0    = 1
    | otherwise = n * factorial (n - 1)
```

Listing 4: Example of a function declaration using guards

2.1.3 Monads

Definition A *Monad* is a subclass of Applicative class that is defined by the following three things [Wik21]:

1. A type constructor
2. A return function
3. A "bind" operator (`>>=`).

Monads provide a way to handle effects and control flow in a pure functional language like Haskell. Monads facilitates the encapsulation of computations that involve side effects or non-determinism and give the ability to control their sequencing and interaction.

```haskell
class Monad m where
    (>>=)  :: m a -> (a -> m b) -> m b
    (>>)   :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a

    m >> k = m >>= \_ -> k
    fail s = error s
```

Listing 5: Monad class according to Haskell 2010 Language Report [Mar10, chapter 6.3.6]

```haskell
class (Functor f) => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Listing 6: Applicative class [Wik22, chapter 29.1.1]

```haskell
do
  statement1
  statement2
  ...
  statementN
```

Listing 7: General syntax of `do` notation

`do` Notation The usage of `do` expressions is a convenient way to compose monadic action in Haskell and gives the user the ability to write imperative-like code inside a monad (e.g. I/O monad). Furthermore, `do` expressions make the code more readable and as a result maintainable by making the sequence of actions and the handling of produced results simpler [Kre15].

```
sumTwoNumbers :: IO ()
sumTwoNumbers = do
  putStrLn("Enter a number: ")
  num1 <- readLn
  putStrLn("Enter a second number: ")
  num2 <- readLn
  let result = num1 + num2
  putStrLn ("The sum of the 2 given numbers is: " ++ show result)
```

Listing 8: Haskell program to sum 2 numbers using do notation [Wik22, chapter 10.2]

2.1.4 Recursion

Haskell does not have a control structure to achieve repetition. This feature is provided using recursion, which makes it a fundamental and very powerful tool for working with lists and other similar structures. The repetition is achieved using recursive functions that call themselves with modified arguments, until they reach the defined base case [Kre15].

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Listing 9: Haskell program to find the factorial of a number using recursion

2.2 Laziness

Lazy evaluation is an evaluation approach that postpones the computation of a statement until the value of it is wanted. This has as the following result: The evaluation doesn't spend more time for excessive and useless calculations. This feature caught the attention of many experts. Many scientists gathered in order to contribute to the design of the language.

In Haskell, expressions in this context are not assessed when connected to variables. Instead, their evaluation is postponed until the results are necessary for other calculations. As a result, arguments are not assessed before being passed to a function. Their evaluation only takes place when their values are actually utilized. This method is known as *call by name* [Has23].

Although lazy evaluation offers numerous benefits, its primary disadvantage lies in the unpredictability of the usage of the memory. The challenge arises because expressions such as `5+5 :: Int` and `10 :: Int` — which produce the same value of 10 — may not have the same sizes, resulting in problems with the memory such as irregular memory usage.

2.2.1 Lazy evaluation and semantics

Operational semantics in Haskell define how a program is examined, and lazy evaluation is a key aspect of it. Conversely, denotational semantics includes non-strict semantics, which describe what a program calculates. The non-strict semantics approach enables the handling of values that are not defined, and also permits the processing of unending data.

2.3 Identifiers

varid → (*small* {*small* | *large* | *digit* | ' })⟨*reservedid*⟩
conid → *large* {*small* | *large* | *digit* | ' }
reservedid → `case` | `class` | `data` | `default` | `deriving` | `do` | `else`
| `foreign` | `if` | `import` | `in` | `infix` | `infixl`
| `infixr` | `instance` | `let` | `module` | `newtype` | `of`
| `then` | `type` | `where` | `_`

Figure 3: Identifiers in BNF notation [Mar10, section 2.4]

2.3.1 Rules

Identifiers in Haskell start by a letter that can be followed by any combination of zero or more letters, digits, quotes or underscores. Underscore by itself is a reserved identifier and in patterns can act as a wildcard. Identifiers are also case sensitive and keywords such as if, infix, else etc cannot be used as identifiers.

2.3.2 Purpose

Identifiers are used to name program entities in Haskell programs. These entities contain variables, functions and data types. Every used identifier must obey the same strict name restrictions and they would better follow some naming conventions that will be described later. Worth mentioning that Haskell is a statically typed language which means that the type of an identifier is determined at compile time.

2.3.3 Scope

The scope of an identifier indicates the part of the program that can access it. In Haskell there are 3 possible scopes for each identifier:

- Global scope: The identifiers that are declared at the top level of the module can be accessed by the entire module.

- Local scope: The identifiers that are declared in a block of code can only be accessed within this block. Every use of them outside the specific block is invalid.
- Parameter scope: The identifiers that are declared as parameters of a function can only be accessed within this specific function.

2.3.4 Naming conventions

The name of an identifier is important to be meaningful and descriptive so the code gets more readable, understandable and maintainable. Also, a good practice is to follow the following conventions [Has22]:

- Type names should start with capital whereas function names with a lowercase letter, so only avoid infix identifiers.
- Laconic and descriptive names are preferred but if longer names are needed they should be in `lowerCamelCase`.
- Type, type class, and constructor names should be written using `UpperCamelCase`.
- In the standard libraries, some parts of the code are written in `snake_case` for long identifiers to better reflect names given with hyphens in the required documentation. Such names should be transliterated to `camelCase` identifiers if they get used out of the libraries by possibly adding a suffix or prefix to avoid conflicts with keywords.

2.4 Keywords

2.4.1 `data`

The keyword `data` is often used in many cases. One very popular case is the creation of an algebraic data type. These kind of data types involve the combination of types through many different ways. The result is the creation of complex types. An example of it's use can be the following:

```
data Mammal a = Cat a
              | Animal (Mammal a) (Mammal a)
```

2.4.2 `data family`

Type families offer a method to declare functions. The environment they are created in is called type level. These functions can be used to connect different presentations. This way a more polymorphic type system can be accomplished. Specifically, data families illustrate data and newtype declarations in a method which is indexed. For example, the definition of a data family in the form of list is:

```
data family ANOTHERLIST l
```

2.4.3 `data instance`

This keyword relates to paragraph `data family`. Specifically, they enable the declaration of types. So this keyword offers a guide for the execution of the above types.

2.4.4 `default`

In Haskell many obscurities can happen when using the class `Num`, so a way of handling them is by using the `default` declaration.

2.4.5 `deriving`

This keyword provides an automatic creation of often used actions for data types that are declared from the user. For example a data type can be defined as an instance of the `Show` class. This derived instance for `Show` provides an automatic creation of the `show` operation. The `show` operation transforms a value of the data type into a string illustration.

```
data Action = Temp
            deriving (Eq)
```

2.4.6 `deriving instance`

It is used in order to define an instance. It refers to a type class. These are created for data types that already exist. They are created differently from the data type. For example:

```
data MyType = MyType Int
deriving instance Show MyType
```

So from the above commands, instances of `MyType` can be printed with the below commands.

```
myValue = MyType 35
print (show myValue)
```

Listing 10: When this code block executes, the result will be: “MyType 35”

2.4.7 `do`

This keyword offers an easier way of using monads. Using the `do` notation the creation of code that specifies authoritative programming is possible. For example we have the following:

```
do {a ; b <- c ; process b}
```

In the above example `a` illustrates an operation that is executed but its output is not assigned to a variable. Also `b <- c` is a way of binding that retrieves the value from the monadic computation `c` and binds it with `b`.

2.4.8 `forall`

In Haskell, type variables are considered universally assessed (by default). This means that there's no need for the type variables to be quantified universally. However it is very often in Haskell that a programmer wants to clarify the quantification distinctly. So the `forall` keyword is used in order to achieve that. For example the following syntax:

```
forall b.b -> b
```

2.4.9 `foreign`

This keyword is useful in case a programmer wants to use a function that it is not defined in any Haskell's library or to give permission to another language to use functions that are written in Haskell. The most common commands are `foreign import declaration`, and `foreign export declaration`. The first one gives permission to a programmer that writes a program in Haskell to use a function which is not included in any Haskell library. The second gives permission to non Haskell programming languages to use a function which is created in Haskell.

2.4.10 `hiding`

Haskell allows to include a module and avoid the entities that are useless. The keyword `hiding` handles entities that are useless for the program. Specifically, it excludes them keeping for the program only entities that are useful. The syntax is the following: `import Data.List hiding (sort, isInfixOf, intersperse)`.

2.4.11 `if, then, else`

Haskell — just like other programming languages — offer a way to create conditional expressions. A program evaluating a condition has some options in order to choose between expressions. The syntax is the following:

```
if condition then expression1 else expression2
```

2.4.12 `import`

In Haskell the reference between modules is possible. This can happen with the keyword `import`. This way modules can use entities that belong to other modules. The general form of the keyword is: `import ModuleName (entity1, entity2, ...)`.

2.4.13 `infix, infixl, infixr`

Fixity declaration is a feature which is used to declare the method that the operators can be associated and the priority which they tie in expressions. It gives the ability of defining the behavior of operators when they appear together. These are the following kind of fixity declarations:

1. Infix left-associative
2. Infix right-associative
3. Infix non-associative

The higher the number the higher the priority of evaluating the operator first in an expression. For example: `infixl 6 +`. This state says that addition has left-associativity and the precedence of it is 6.

2.4.14 `instance`

A type can be defined as an instance. It will refer to a class. This method is done using this keyword. Also it allows to provide the implementations for the class methods that are associated with it.

```
instance ClassName Type where
```

2.4.15 `let, in`

These keywords are commonly used in order to define variables in a specific scope. If the `in` keyword is included then it states the scope of the declaration, else the declaration is considered as district. Finally the syntax is the following:

```
let declaration1 ; ... ; declarationn in s
```

2.4.16 `module`

It's a container that holds related definitions, functions, types, entities etc.

2.4.17 `newtype`

This keyword is used as a way to create new types. Specifically those types are the existing algebraic data types, but with different names. This method offers a way to present distinct types that have the same representation as existing ones. The syntax is

```
newtype NewTypeName = Constructor ExistingType
```

The difference between `newtype` and other type declaration keywords is that it forces stricter type-checking.

2.4.18 `proc`

The keyword `proc` is responsible for the definition of a very important feature. It is called *arrow abstractions*. The result of it is the creation of an arrow. An arrow is a generalized function. It is used because of it's expressiveness when it comes to illustrating calculations and control flow. The syntax of this keyword is the following:

```
proc pattern -> do
-- Arrow computations
```

2.4.19 qualified

The keyword `qualified` is used as a way to include a module. However in this case the presentation of its name inside the scope can be avoided. This technique is often used when there are entities of modules with the same names, so the danger of a conflict happening is high. For example:

```
import qualified Data.Text
```

2.4.20 rec

This keyword is used in conjunction with a specific flag (`-XDoRec`). It is used in order to activate connections. Those connections have recursion. This occurs within the `do` block. By default this is not allowed. A `do` block is a way of putting in order operations.

2.4.21 type

`type` can be used as a way of renaming an already existing algebraic data type, and every time this type is referred within the program with the new name it is being executed. This way, it is more convenient to refer to complex types. For example `type PhoneNumber = String`.

This way, whenever a program uses the type `String` it can use the type `PhoneNumber` (which are the same now) as well and if there's a reference of type `PhoneNumber` it can be used wherever the original type (`String`) is anticipated.

2.4.22 type family

In Haskell a `type family` enables type generalization. It is used as a way to declare specific types (family types). They are synonyms and they are related.

2.4.23 type instance

This keyword is used in order to create an instance of the above kind of type.

2.4.24 where

It allows to declare a definition, like a function or a value, that can be seen and used inside a scope. The `where` statement is attached to some construct like a function definition and can be used in relation to that. For example:

```
calculateDiscount :: Double -> Double -> Double
calculateDiscount price discountRate = discountedPrice
  where
    discountedPrice = price - (price * discountRate)
```

The `where` statements declare the variable within the where clause, we define `discountedPrice`, which in this case illustrates the price after the discount.

2.4.25 { and }

This keyword is used in order to define the scope of various statements. The statements are separated from each other with a semicolon (;). For example:

```
doSomething = { x = 10; y = 20; z = x + y }
```

Also this keyword is used in records. For example:

```
record field1 = value1, field2 = value2, ...
```

2.4.26 {- and -}

This is used as a way to write multi-line comments in Haskell. Specifically everything inside the {- and -} is considered as a comment and will be avoided from the compiler during the compilation.

2.4.27 |

A pipe can be used in the following cases:

1. This keyword is used to divide various constructors inside a data type definition. Each one illustrates a value that the data type can take. For example:

   ```
   data Result a = Success a
                 | Failure String
   ```

2. It is used in lists. For example:

   ```
   positiveSquares = [a + a | a -> [2..], a > 0]
   ```

3. It is used in order to separate conditions or guards in a function definition. For example:

   ```
   isPositive :: Int -> Bool
   isPositive x
     | x > 0 = True
     |otherwise = False
   ```

2.4.28 ~

The tilde is used in order to denote a lazy pattern bind. They provide a mechanism for introducing patterns that are evaluated when needed. This type of behavior can be particularly useful when dealing with possibly infinite data structures.

2.4.29 `

This feature allows to use infix notation inside statements [Mar10]. For example backticks can be used to treat add as an infix operator: `result = 5 `add` 3`.

2.4.30 _

It is used as a token that matches a value. It basically represents a value that there's no need to distinctly name. For example:

```
getFirstElement :: [a] -> Maybe a
getFirstElement [] = Nothing
getFirstElement (x:_) = Just x
```

2.4.31 \

The backlash is used as a way to split a string in many lines. For example:

```
s = "hi\
    hello"
```

It is also used as a way to define anonymous functions. For example:

```
subtract = \x -> x - 1
```

2.4.32 [| and |]

It is used in the following cases:

1. It allows the treatment of the quoted expression as a first-class value.
2. To illustrate a declaration.
3. To illustrate a type.
4. To illustrate a pattern.
5. To define a quoting syntax.

2.4.33 @

It offers a method to give a name to a sub-pattern. This allows to refer to the matched value by the given name in the scope. For example:

```
example :: [Int] -> [Int]
example inputList = case inputList of
newList@(first:remaining) -> if first == 0 then
remaining else newList
```

The pattern `newList@(first:remaining)` is an as-pattern. It matches the list if it starts with an element first followed by the remaining elements remaining. The matched list is assigned the name `newList`.

2.4.34 *

In Haskell the form of specific types (boxed) are represented with the symbol `*`. For example the `*` kind indicates that Int is a boxed type.

2.4.35 ?

It is used in cases like the following:

```
ghci> :t ?name ++ " is awesome!"
  ?name ++ " is awesome!" ::
  (?name :: String) => String
```

In this example, `?name` is an implicit parameter of type `String`. The expression `?name ++ " is awesome!"` concatenates the value of `?name` with the string " is awesome!".

2.4.36 =>

It is used to specify constraints on the type variables or type classes that are involved in the function signature.

2.4.37 <-

1. It is used to bind the result of a monadic action to a variable.
2. It is used in lists.
3. It is used to match a pattern and connect values in a guard.

2.4.38 ;

It is used as a way to divide expressions in a sequence of statements that are enclosed in braces. For example:

```
do {
putStrLn "Hello";
putStrLn "World";
}
```

2.4.39 ::

It is used to indicate the type signature of a declaration. For example:

```
add :: Int -> Int -> Int
```

The `::` operator specifies the type signature of the function, indicating that add has the type:

```
type [Int] -> Int -> Int
```

2.4.40 ->

1. It is used to denote the type of a function that takes one or more arguments and returns a result.
2. They are used in functions (lambda).
3. It is used in case expressions.
4. It is used in a specific feature (view patterns) that allows pattern matching
5. It is used in a specific feature (functional dependencies).

2.4.41 -«

It is used as a way to express calculations in a way that uses arrows.

2.4.42 !

Algebraic datatypes in Haskell are defined by constructors, which take one or more arguments. When a new value is created using a constructor, the corresponding values that will be merged to create the final value are generated by evaluating the arguments.

Haskell follows lazy evaluation by default, implying that the arguments supplied to a data constructor are not computed until they are required. This approach can be inefficient at times, especially when the arguments are either *pricey* to calculate or not utilized at all.

Haskell handles this by giving the permission for the definition of strictness for the arguments that were offered to a constructor. In these cases with the use of a strictness flag represented by the symbol `!`, which is included in the algebraic datatype declaration. They are used in order to specify the arguments that must be calculated immediately upon constructor implementation [Mar10; Has23].

If the goal is the definition of a binary tree data type in Haskell, we can depict a binary tree as a node containing a value and two children, where each child can either be another node or an empty leaf. We can establish this data type using two constructors; one for the internal nodes and another for the leaves (listing 11).

```
data BinaryTree a = Node a !(BinaryTree a) !(BinaryTree a)
  | Leaf
```

Listing 11: Definition of the `BinaryTree` datatype.

The `BinaryTree` data type, consisting of two constructors: `Node` and `Leaf`. The `Node` constructor receives three arguments - a value of type “a”, and two `BinaryTree` “a” values

that represent the node's left and right children. The presence of an exclamation point before each child argument specifies that the children must be strictly evaluated. This denotes that their values will be calculated before the `Node` constructor is utilized, ensuring that the entire `BinaryTree` value is wholly computed during its construction.

Also, the exclamation point is further utilized denoting rigor within patterns (see listing 12). The function uses a bang pattern. This happens in order to denote that its arguments `x` and `y` must be calculated strictly prior to the application of the function. This ensures that any possible delay in evaluating the arguments is eliminated, which can improve performance in certain scenarios.

```
g :: Int -> Int -> Int
g !! y = x * y
```

Listing 12: Pattern matching

2.5 Haskell's Prelude

`Prelude` is a module that contains a small set of standard definitions and is included automatically into all Haskell modules [Has23]. It can be thought of as Haskell's default API.

The Prelude provides the user with a wide range of function, data types and type classes, which make it very useful for the programmers.

The following can be found in `https://hackage.haskell.org` [Hac23b].

2.5.1 Basic Functions

Some of the most used functions are the following:

- `map`: applies a given function to all the elements of a given list
- `filter`: applied to a predicate and a list, returns the list of those elements that satisfy the predicate
- `read`: transforms a string to another datatype
- `length`: returns the length of a given list

2.5.2 Basic Data Types

Some of the most used data types are the following:

- `Int`: Integer
- `Double`: Floating point (double precision)
- `Bool`: Boolean (True or False)
- `Char`: Character
- `String`: String

2.5.3 Basic Type Classes

- `Eq`: defines equality and inequality
- `Read`: parses a given string and produces values
- `Num`: numeric class

2.5.4 Standard Output Functions

- `putChar`: writing a char
- `putStr`: writing strings
- `putStrLn`:writing a string and adding a newline.
- `print`: outputs any kind of printable type

2.5.5 Standard Input Functions

- `getChar`: reading a char
- `getLine`: reading lines
- `getContents`: returning the input, that was provided to a string

2.5.6 Error Functions

- `error`: stops running program due to error
- `errorWithoutStackTrace`: stops running program without a stack trace
- `undefined`: explicit type of error

2.5.7 Filepath Functions

- `readFile`: reading a file
- `writeFile`: writing in a file
- `appendFile`: appending in a file

2.5.8 Shows Functions

- `shows`: transforms a value to a String type
- `showChar`: tranfrom a char to a ShowS type function
- `showString`: tranfrom a string to a ShowS type function
- `showParen`: offers parentheses to inside function (show)

$$
\begin{array}{ll}
\text{varsym} & \rightarrow \Big(\text{ symbol } {}_{\langle : \rangle}\{\text{symbol }\}\Big)_{\langle \text{reservedop} \mid \text{dashes} \rangle} \\
\text{consym} & \rightarrow (: \{ \text{ symbol } \})_{\langle \text{reservedop} \rangle} \\
\text{reservedop} & \rightarrow \ldots \mid : \mid :: \mid = \mid \backslash \mid \mid < - \mid - > \mid @ \mid \sim \mid =>
\end{array}
$$

Figure 4: Haskell operators in BNF form

2.6 Operators

2.6.1 In general

Haskell has a rich set of built-in operators, which is included in the prelude [lib23]. Therefore, it supports operator overload, which means that the user can define his own operators and define their functionality.

In Haskell, operators are defined as functions that take one or more arguments. There are two types of operators [Mar10; Has23]:

- Infix operators (the operator goes *between* 2 arguments).
- prefix operators (the operator goes *before* the arguments), but the vast majority of them is infix.

Table 1: Operators and their types in Haskell's prelude [Fag]

Operator type	Example
Arithmetic	`+`, `-`, ...
Logical	`&&`, `not`, ...
Comparison	`<`, `<=`, `>`, `>=`, ...
Bitwise	`.&.`, `xor`, ...
List	`:`, `++`, ...
Tuple	`fst`, `snd`, ...
Function	`$`, `.`, ...

The operators based on their syntax are distinguished in two namespaces:

- An operator starting with a colon is a constructor.
- An operator starting with any other character is an ordinary identifier.

The colon operator solely is reserved by Haskell as the default list constructor.

2.6.2 Fixity Declarations

Like in every other programming language, each Haskell operator has a precedence. In Haskell, this characteristic is expressed by an integer value in the range of 0 to 9.

In case that two or more operators have the same precedence, there is another property to act as a "tie-breaker" and determine how the operators will be grouped without using parentheses. This property is called fixity.

There are 3 kinds of fixity:

$$
\begin{array}{lll}
\text{gendecl} & \rightarrow & \text{fixity [integer] ops} \\
\text{fixity} & \rightarrow & \text{infixl|infixr|infix} \\
\text{ops} & \rightarrow & op_1, \ldots, op_n \quad (n \geq 1) \\
\text{op} & \rightarrow & \text{varop} \mid \text{conop}
\end{array}
$$

Figure 5: The *integer* in this rule is the precedence of the operator.

- non-associativity (infix)
- left-associativity (infixl)
- right-associativity (infixr)

For every operator the precedence and the fixity have to be declared. This declaration in BNF is shown in figure 5.

2.7 Lambdas

A lambda expression is a way of defining an **anonymous function**. Lambdas help developers define functions "on the fly" and pass them directly as arguments to another function, making your code concise and expressive. They are especially useful for defining functions that are only used once or so simple they do not need a name. Lambdas are a core feature of functional programming, naturally they are used extensively in Haskell.

A lambda expression is defined using the backslash character `\` followed by the arguments, which are separated by spaces, then an arrow `->` that separates the arguments from the body. The body is a Haskell expression that is evaluated when the lambda is called. For instance, the lambda `\x -> x + 1` takes an argument `x` and returns the value of `x` incremented by 1. The syntax of a lambda expression in Haskell is shown in listing 13.

```
\arg1 arg2 ... argN -> body
```

Listing 13: Lambda syntax

```
\x y -> x * y
```

Listing 14: A lambda that multiplies two variables

The code in listing 15 uses the `map` function to apply the lambda to each one of the elements as it iterates the list. The resulting list contains the result of each addition.

The `filter` function selects the elements of a list that satisfy a given condition. Here's an example that demonstrates the use of a lambda that is given to filter to select only the even numbers from a list of integers:

Table 2: The operators included in the prelude with their precedence and fixity [Mar10].

Precedence	Left associative operator	Non-associative operator	Right associative operator
9	`!!!`		`.`
8			`^`, `^^`, `**`
7	`*`, `/`, `div`, `mod`, `rem`, `quot`		
6	`+`, `-`		
5			`:`, `++`
4		`=`, `=/=`, `<`, `<=`, `>`, `>=`, `elem`, `notElem`	
3		`&&`	
2		`||`	
1	`>>`, `>>=`		
0			`$`, `$!`, `seq`

```
map (\x y -> x + 1) [1, 2, 3]
-- map (+ 1)  [1, 2, 3]
> [2, 3, 3]
```

Listing 15: This lambda takes one argument and returns its value incremented by 1. It is then passed to a higher-order function `map`

```
list = [1,2,3,4,5,6]
filter (\x -> x `mod` 2 == 0) list
-- filter (\x -> even x) list
-- filter even list
> [2, 4, 6]
```

Lambda expressions can also be composed to create more complex functions using the `.` operator. For example, the expression `((\x -> x + 2) . (\y -> y * 2))` creates a new lambda that first multiplies its argument by 2 and then increments it by 2. This composed lambda function can consequently be applied to any argument.

```
( (\x -> x + 2) . (\y -> y * 2) ) 3
> 8
```

2.8 Monads

Monads are a predominant concept in Haskell that grant developers the ability to represent computations as sequences of steps. A monad is a type class in Haskell that defines two operations: `return` and `>>=` (called "bind").

The `return` operation (also a keyword) takes a value and puts it in a monad. For example, `return 42` creates a monadic value that contains the int 42. This operation lifts a pure value into a monadic context.

The `>>=` operation is employed to connect monadic computations together. It takes a monadic value and a function that returns another monadic value, and applies the function to the value inside the monad. The result is a new monadic value that combines the effects of both computations (listing 16).

```
Just 3 >>= (\x -> Just (x * 3))
> Just 9
```

Listing 16: `>>=` operator

The expression `Just 3 >>= (\x -> Just (x * 3))` applies the function `(\x -> Just (x * 3))` to the monadic value `Just 3`, resulting in the monadic value `Just 9`.

Monads are a way of abstracting away the details of computation, allowing you to write generic code that can work with different types of computations. In Haskell, many types are instances of the Monad type class, including the `Maybe` type and the `IO` type.

The `Maybe` type is used to represent computations that might fail. The `IO` type is used to represent input/output operations that interact with the outside world.

```
main :: IO ()
main = do
    putStrLn "What is your name ?"
    name <- getLine
    putStrLn ("Hello, " ++ name ++ "!")
```

Listing 17: How monads are able to perform I/O in Haskell

In listing 17 firstly the `main` function is defined as an `IO` computation and the `putStrLn` function is used to print a message to the console. After that the `getLine` function is used to read a line of text given from the user as input. The `name` variable is bound to the result of `getLine` using the `<-` operator, which is a shorthand for using `>>=`. Finally, there is a second `putStrLn` call used to print a "Hello <name>" to the console.

Monads are a powerful abstraction that allow us to express complex computations in modular way. By defining new monads and providing instances for the Monad type class developers are able to create their own abstractions as well as reuse them in different parts of their code.

2.9 Memory management

To gain insight into the memory management of Haskell, we analyze the implementation of `ghc` as a point of reference.

2.9.1 Garbage collection

`ghc` uses a "parallel generational-copying garbage collector" [Mar+08] that traverses the live data of the running program. The two generations are Generation 0 and 1 (figure 7). An illustration of the garbage collection steps of `ghc` is shown in figure 6.

Newly generated data When new data are generated, they are stored in the *nursery*, a 512kb memory space [Has23]. In every iteration of the garbage collector, each value in Gen 0 that is still used is promoted in Gen 1[4].

Collecting the garbage The garbage collector initiates a search, starting from the so-called *roots*, meaning the data in the stack and possible global variables. It then moves towards any data that roots point to. These pieces of data are stored in the heap.

[4]The transition to Gen 1 actually involves two steps: Data that are not to be deallocated remain in Gen 0 during the "aging" phase, before being promoted to Gen 1.

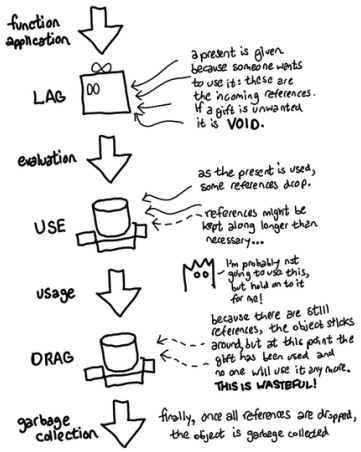

Figure 6: A general overview of the steps of garbage collection [Yan]

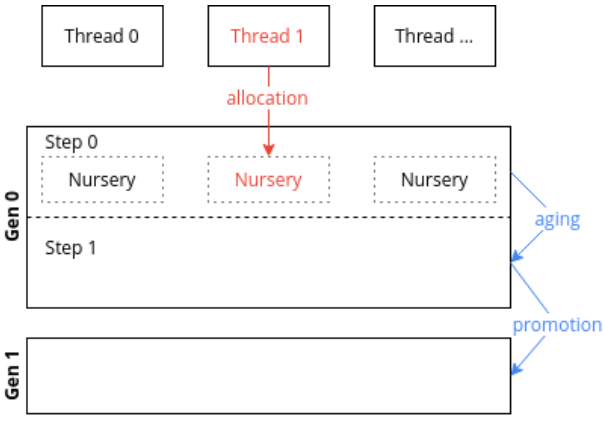

Figure 7: Two-generation GC of `ghc` [Cha]

Copying The visited pieces of data are then copied in parallel to a newly allocated heap [Mar+08] (figure 8). This process is called "evacuation" [Yan].

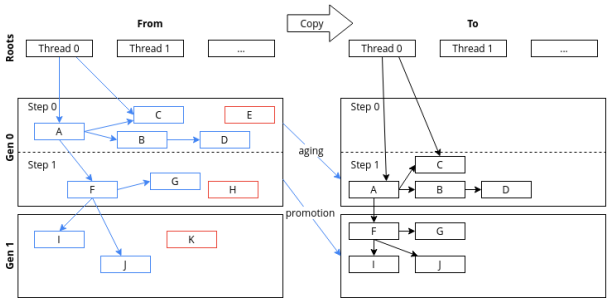

Figure 8: The copy mechanism of the `ghc` GC [Cha]

"Parallel" means that as the GC searches for the data that the roots are pointing to, the roots are "migrated" in the new heap at the same time. The process of finding the data pointed by the roots is called "scavenging" [Yan].

Also, if a piece of data is evaluated and its value is used, it is then considered garbage and collected by the GC [Yan].

The downsides of this copying property of the GC is that the memory needed is double the one of the initial heap. That is, the old heap can't be freed before all of its live data are copied in the new heap.

This problem is solved by the use of a mark-compact algorithm, that reduces memory needs by rearranging the live data in the same heap [Cha]. It is used when there are a lot of live data in the old heap.

Compact regions Another problem of the garbage collection in Haskell is that it is performed successively with the main thread and not in parallel. That means that the process of garbage collection halts the running program and therefore time complexity is afflicted[5] [Cha].

The solution to the previous problem is found in *compact regions* [Yan+15], which were introduced in version 8.2.1 of `ghc`. They constitute a section of memory that is treated as a separate heap and handled separately. If, while searching, GC encounters data that lives in a compact region, then it stops searching for dead data pointed by it. This function is better illustrated in figure 9. By using compact regions to store big amounts of live data, garbage collection time is greatly reduced. Compact regions are pinned in memory and the GC doesn't alter them in any way [Cha].

Mutability and garbage collection Mutability is one of the features of Haskell that negatively influences GC time. Mutability gives data the ability to point to "future" objects, meaning data that have not yet being created. This means that a live object in Gen 0 can depend in an object of Gen 1 [Cha].

[5]From version 8.10 of `ghc` there exists an alternative GC strategy that allows concurrency [Gam19].

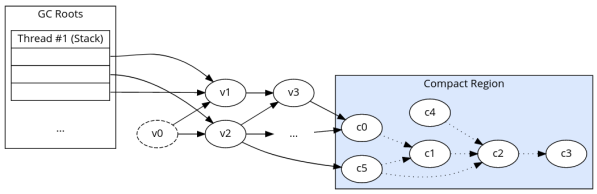

Figure 9: Compact regions in GC [Cha]. When the GC reaches c0, it stops searching for dead data in anything that is pointed by it. As a result, data such as c4 is kept, even though nothing points to it.

2.10 Namespaces

2.10.1 Definition

A namespace is group of related elements that each one of them has a unique name so it can be easily identified[6].

2.10.2 Namespaces in Haskell

In Haskell there are 6 kinds of names that can be grouped in broader categories as follows: Firstly, Values. They consist of variables and constructors. Second, Entities related to file system. They consist of type variables, type constructors and type classes. Third and last, Modules. They consist of module names.

Haskell has just 2 naming constraints: First, names for variables and type variables are the only names that start with underscore or a lowercase letter. The names for the other categories are not allowed to start with these characters. Second, a class and a type constructor cannot use the same identifier as a name in the same scope [Mar10].

2.11 Parallelism and Concurrency

Parallelism and **Concurrency** are two different concepts in Haskell that enable the execution of tasks concurrently or in parallel, allowing for increased performance and more efficient utilization of hardware resources. Haskell provides exceptional support for both of them.

Parallelism indicates to the execution of computations simultaneously on multiple cores, while Concurrency refers to managing multiple tasks concurrently, without regard to the number of physical processors. Haskell's libraries like `Control.Parallel` and `Control.Concurrent` facilitate parallel and concurrent programming.

Haskell provides several mechanisms for parallel programming:

Annotations like `par` and `pseq` allow programmers to explicitly specify parallel or sequential execution. Using the par keyword the developer denotes that an expression can be evaluated in parallel, while `pseq` enforces sequential evaluation. The par annotation only acts as a guide to the runtime system, in order for it to evaluate expressions in parallel whenever possible.

[6] https://techterms.com/definition/namespace

Strategies are high-level abstractions for specifying parallelism. The `Eval` monad along with the `rpar` and `rseq` combinators provided by the `Control.Parallel.Strategies` module allow developers to express parallel computations more flexibly and easily. Strategies also enable you to specify granularity control, something truly important for load balancing and steering clear of excessive overhead.

Furthermore, there are parallel versions of common list functions, like `parMap` and `parList`, which automatically distribute the computation across processors. These functions make it convenient to parallelize operations on lists and apply a function to each one of the elements concurrently.

As it was already stated, Concurrency deals with managing multiple tasks, regardless of the underlying hardware capabilities. It mainly focuses on the organization and coordination of tasks to achieve efficient resource sharing and responsiveness. Haskell provides several powerful abstractions and features for managing concurrency in a safe and expressive manner.

The `IO` monad provides input/output operations, including concurrent I/O. It ensures that I/O actions are sequenced correctly and provides a safe and controlled scheme to perform concurrent operations.

Software Transactional Memory (STM) is a powerful mechanism for managing shared mutable states. It allows multiple threads to perform transactions, which are sequences of operations on shared memory. It ensures that transactions are atomic and isolated, preserving data integrity and simplifying concurrent programming. STM makes concurrent programming easier by automatically handling conflicts (race conditions - false sharing) and providing a consistent view of shared data.

Haskell enables developers to write concurrent programs that are concise, scalable, and robust.

2.12 Metaprogramming

Metaprogramming in Haskell refers to writing programs that manipulate other programs as data. It allows programmers to generate, transform, and analyze Haskell code.

Haskell provides various features for metaprogramming:

Template Haskell is a prominent metaprogramming extension built into `ghc`. It grants developers the capability to define and manipulate Haskell syntax at compile-time. With Template Haskell, dynamic code generation, code transformations and embed domain-specific languages (DSLs) are all possible within the language.

Quasi-quotation is another metaprogramming technique in Haskell. It allows code written in a different language or DSL to be directly embedded within Haskell source code. It provides a simplified way to generate or manipulate code in a non-Haskell language.

Haskell's strong type system enables type-level programming, writing code at the type level. It also allows for performing computations and expressing complex constraints at compile-time. Type families, type-level functions, and type-level literals are the tools that support type-level programming in Haskell.

Metaprogramming in Haskell serves various purposes, such as code generation to automate repetitive or boilerplate code, creating domain-specific languages for concise and expressive abstractions, transforming programs in order to be optimized and adaptive to specific requirements as well as enabling generic programming for code reuse across different data

types. While metaprogramming can be a potent technique, it must be based upon a deep understanding of Haskell. Extreme care is needed, in order to maintain code clarity and readability when using metaprogramming approaches.

2.13 Object-Oriented Properties of Haskell

Haskell is a functional language and as a result it does not provide object oriented functionalities by default. However, there are some tools given by the language that can be used to encode object oriented features in it [Fra13; KL05]. A sample representation of two fundamental entities of Object Oriented Programming follows: Classes and Objects.

2.13.1 Classes

A class can be mapped to an abstract Haskell type. A constructor can be defined as well, as it is supported by Haskell. Lastly, the user can define the needed functions that use that type [Fra13].

```
class C {
    C(int x) { ... };      /*Constructor*/
    static int s(int x);   /*Static method*/
    int m(bool b, int x);  /*Instance method*/
}
```

Listing 18: A class definition in C#

```
newtype C  -- An abstract type
newC :: Int -> IO C
s    :: Int -> IO Int
m    :: (Bool, Int) -> C -> IO Int
```

Listing 19: The equivalent “class” in Haskell (listing 18)

2.13.2 Objects

The objects of a class can be modeled as containers (data structures) that contain the data of the class. In Haskell the user has to define the data structure in two phases: data declaration and a class declaration for its methods [Fra13].

2.13.3 Conclusion

To conclude, Haskell does not provide native object-oriented functionalities, but most of them can be implemented in the language.

2.14 Classes

Declarations		
top_level_declaration	→	class[a_simple_context_=>]typeclass typevariable [where class_method_declaration]
a_simple_context_	→	a_class
a_simple_context_	→	(a_class$_{first}$, ... ,a_class$_{last}$) (last≥0)
a_class	→	qualified_class type_variable
class_declarations	→	{class_declaration$_{first}$,.....,class_declaration$_{last}$} (last≥0)
class_declaration	→	(function_left_hand_side — variable) right_hand_side

Figure 10: Classes in Haskell

In Haskell a new class can be defined alongside with its operations through the process of class declaration. The syntax `class cx=> MyClass tv where dec` is a way to declare a class. `MyClass` is the name of the new class, and `tv` is a type variable that is used only within the class definition. Also The `cx` part of the declaration specifies any superclasses of `MyClass`. Finally, the `dec` part of the declaration specifies the methods and properties that belong to the class [Mar10, chapter 4.3].

Specifically from the class declarations we have the following: A class declaration results in the creation of class methods, represented b variables$_i$. The class methods defined within the class declaration are not limited to the class declaration itself, but can be accessed and used outside of it. Every class method will have this kind of type, `method :: context => type`.

In Haskell, it is known that types like variables, functions, data types etc. have the same namespace. This basically means that two different top-level names can't have the same name because of the conflicts that may occur.

The same rule also applies for classes and their bindings. A compilation error will happen when there are two or more names with the same identifier in the same namespace. So the following is the signature form of a top level class method: `fn :: ` $\forall x, y.(Cx, cyz) \rightarrow r$.

Type classes provide a way to abstract over container types, allowing class methods to have polymorphic type signatures that can involve additional type variables and constraints beyond the primary type parameter of the class (listing 20).

```
class Container c where insert :: a -> ca -> c a
remove :: a -> ca -> c a
size :: c a -> Int
```

Listing 20: Polymorphic type signatures

Also, fixity declarations are used to specify the precedence and associativity of operators [Mar10]. It is possible that the `dec` declaration from above has fixity declarations for class methods, allowing to specify the precedence and associativity of those methods.

Finally, in Haskell a default class method is a method that provides a default implementation. It is offered for a class method in case it is not defined explicitly in an instance declaration. In Haskell, a class declaration can have a `where` keyword, which contains the

method signatures for the class methods. A class declaration may also not have this `where` keyword. The second case is used in order to merge many classes to one bigger class that inherits the methods of the smaller ones (listing 21).

```
class (Ord a, Num a) => OrderedNum a where
absSquared :: a -> a
```

Listing 21: Class inheritance. In this example, the `OrderedNum` class combines the `Ord` and `Num` classes and defines a new method `absSquared` that is specific to `OrderedNum` instances.

Becoming an instance of a subclass is not guaranteed. Mainly if a type is an instance of the superclasses. To make a type an instance of such a subclass, then an explicit instance definition without a `where` must be given [Mar10].

2.14.1 Instance Declaration

Declarations		
top_declaration	→	instance[a_simple_context_=>]qualified typeclass instance [where instance_declaration]
instance	→	general_type_constructor
instance	→	(general_type_constructor typevariable$_{first}$,....., general_type_constructor typevariable$_{last}$) (last $\geq$ 0 , typevariable discrete)
instance	→	typevariable$_{first}$, ... , typevariable$_{last}$ (last $\geq$ 2, typevariable discrete)
instance	→	[typevariable]
instance	→	(typevariable$_{first}$ → typevariable$_{last}$) (typevariable$_{first}$ and typevariable$_{last}$ discrete)
instance_declaration	→	{instance_declaration$_{first}$; .... ;instance_declaration$_{last}$} (last$\geq$0)
instance_declaration	→	(function_left_hand_side | variable)right_hand_side
	→	(empty)

Figure 11: Instance declarations

An instance declaration, on the other hand, defines an implementation of a class for a specific type.

```
class (constraint) => Aclass parameter where cbody
```

The following instance declarations cannot be used.

```
instance Aclass (typ,typ) where ...
instance Aclass (Int,typ) where ...
instance Aclass [[typ]] where ...
```

From the above examples tuple types which their elements have the same types are not allowed to be used by an instance declaration. Also it is not allowed to have the first element

as a specific type and the second element as a type variable. Finally an instance cannot use a list of lists of type variables.

A class declaration provides type signatures and fixity declarations so they are not allowed in an instance declaration. The method declarations within the instance must be in some kind of forms. These forms must be in the form of a variable or function definition.

When a class method is not explicitly bound in an instance declaration, Haskell looks for a default class method. If a default is found, it is used in place of the missing binding `else`, Haskell assigns the value to `undefined`, which will not trigger a compile-time error [Mar10].

In order to declare an A-B instance that A is an instance of the class B then a number of constraints must be satisfied:

1. In Haskell, you cannot define a type as an instance of a given class many times in a program.
2. Kind inference can be used in order to make the class and type have identical kind. Those two must have the same type.
3. Hypothetically theres an instance context `con 0`, which restrictions are satisfied by the bk of the instance type $A(b_1, \ldots, bk)$. Then the following rules must be satisfied as well:
   - The restrictions provided by the superclass context `con0` of a class declaration should not be violated.
   - In order to get the class method declarations presented the correct way, then the restrictions on the type variables must not be violated as well.

```
class Animal b => Mammal b where ...
instance (Eq b, Show b) => Animal [b] where ...
instance Num b => Mammal [b] where ...
```

In the above example the `Mammal` is a subclass of `Animal` so the second declaration is acceptable if `[b]` is an instance of `Animal`. `Eq` and `S` are superclasses of `Num` so the `[b]` is an instance of `Animal` from the first declaration.

On the other hand, if there are two instance declarations

```
instance Num b => Mammal [b] where ...
instance (Eq b, Show b) => Animal [b] where ...
```

The program wouldn't be acceptable. In order to be acceptable then for the second instance it is required that the `[b]` has to be instance of `Mammal`, including the fact that this should happen including `Eq b` and `Show b`. However in this case it is not true.

2.14.2 Derived Instance

A deriving keyword can be included when defining a data type or a new type. This keyword makes the following difference: basically this move gives permission for a automatic creation

of instance declarations for this type. This happens in specific classes. It must be noted that the generated instances have identical limitations as other instances (user-defined). The derivation of a class can happen when instances for the classs superclass are present, and they actually do exist. It can happen through the definition of a distinct instance declaration or by adding the superclass in the clause that the derivation happens.

There are many benefits that come from a derived instance. One of them is a simple approach to defining operations that are used all the time (referring to custom data types). Instead of defining these operations from scratch, the derived instances for datatypes in the class `Eq` define them for the programmer. This can save time and effort and can help ensure consistency in the implementation of these operations.

Only for a specific set of classes can derived instances be permitted (`Eq`, `Ord`, `Enum`, `Bounded` etc.). The standard libraries defined classes can be eligible for derivation as well.

If a class name is included in a deriving form and the generation of an instance declaration cannot happen, then theres an error [Mar10]. Additionally, if a class is both explicitly declared and derived, this will also result in an error.

2.14.3 Defaults, ambiguous types

top level declaration $\rightarrow$ default(just_a_type$_{\text{first}}$, ..., just_a_type$_{\text{last}}$).

In Haskell, ambiguous types can be the result of an overloading implementation. Supposing theres a typeclass named `MyClass` with a single function `Func` (like in the following listing).

```
class MyClass a where
Func :: a -> Int
```

Also there are definitions of two instances of `MyClass` for `Typea` and `Typeb`:

```
instance MyClass Typea where
Func x = 1

instance MyClass Typeb where
Func x = 2
```

The use of `Func` with an ambiguous type like this here: `let x = Func undefined in ...` will have as a result a type error, because Haskell doesn't know what instance to use, `Typea` or `Typeb` for Func based on the type of undefined.

We have an ambiguous type when a type of an expression contains a type variable that appears in the context but it isn't in the result type. For example, suppose we have a function `Func` with the following type signature:

```
Func :: Num a => a -> a -> a
```

Also theres this use:

```
let x = Func 5 6 in show x
```

Here, the type of `x` could be any type that is an instance of `Num`, which includes `Int`, `Double`, `Float`, and others. Since `show` can work with any type that is an instance of `Show`, the type of the expression `show x` is ambiguous, because it could be any of the possible types that `x` could have.

In order to resolve such a problem we use an expression type like this:

```
let x = Func 5 6 in show (x :: Int)
```

the type of `show` is unambiguous here.

In order to ensure that an ambiguous expression has identical type with a variable, the use of the function `asTypeOf` can be helpful.

To address the most frequent ambiguities arising from the `Num` class in Haskell, the language gives an alternative method known as *default declaration*. The form of it is $\text{type}_{\text{first}}, \ldots, \text{type}_{\text{last}} \text{last} \geq 0$. An ambiguous type is *defaultable* if the following are valid (supposing theres an ambiguous type variable `x`):

1. Supposing we have the form `Class x` where `Class` is a class. If `x` occurs only in those kind of type constraints then it is defaultable.

2. If there is one class among the constraints that is a subclass of or equivalent to the `Num` class, then it is considered as a "numeric class" (the `Num` class is the most common numeric class in Haskell, and it defines basic numeric operations such as addition, subtraction, multiplication, and negation).

3. The classes in which the type variable `x` appears should be either in the Prelude or in a standard library.

A defaultable variable can be substituted by the type that is mentioned as first (noting that it is in the default list) that satisfies all of its classes.

It is also worth noting that each module can have a single default declaration. Also its scope is bounded to that module. The cases that default types are assumed (specifically `Integer`, `Double`) are the cases where a default declaration is not explicitly provided.

2.15 (Trans)portability

The implementation of Haskell, also known as `ghc`, is supported by a variety of platforms. These platforms are divided into three tiers, as indicated by the `ghc` GitLab repository's wiki [Gam23].

- Tier 1

  Tier 1 comprises major computing platforms, such as Windows (MinGW), MacOS X, and Linux, supporting x86/64, x86, and AArch64 architectures. The `GHCi` interpreter, `NGC` native code generator, and dynamic libraries are all supported in these platforms.

To be included in tier 1, a platform must satisfy certain requirements, including the availability of a GitLab CI runner and a dedicated sponsor to provide support for new releases [Gam23].

These platforms constitute the main targets for `ghc` developers. All platforms in tier 1 must be supported to release a new version of `ghc`.

- Tier 2

  Platforms on tier 2 consist largely of various BSD distributions such as FreeBSD and OpenBSD, Solaris, and some less typical architectures of Linux. There is varying support for the `GHCi` interpreter, `NCG`, and dynamic libraries for these platforms. Support for these platforms depends heavily on community support, since `ghc` is open-source software. Therefore, not all platforms in tier 2 may necessarily be supported by new releases of `ghc`.

- Tier 3

  Finally, tier 3 includes platforms such as iOS, AIX, and the RISC-V architecture version of Linux. These platforms are less reliable, with some versions of `ghc` possibly available but generally unsupported.

3 Examples and code

3.1 The "Hello, World!" program in Haskell:

```
main = putStrLn "Hello, World!"
```

Listing 22: "Hello, World!" program

The program in listing 22 defines a main function that uses the `putStrLn` function to print the string `Hello, World!` to the console.

To run this program, you can save it in a file with a `.hs` extension. Then compile and execute it from the command line using the commands shown in listing 23.

```
$ ghc hello.hs
$ ./hello
Hello, World!
$
```

Listing 23: How to compile and run a program with the `ghc` compiler

The *Hello World* program is traditionally the first program we write when learning a new programming language. In the world of functional programming we need something different to showcase the power of Haskell.

3.1.1 Compilation process of the ghc compiler

After you compile a Haskell program there will be 3 new files (listing 24).

```
$ ghc HelloWorld.hs
[1 of 1] Compiling Main ( HelloWorld.hs, HelloWorld.o )
Linking HelloWorld ...
$ ls
HelloWorld HelloWorld.hi HelloWorld.hs HelloWorld.o
```

Listing 24: Files generated by `ghc`

The `HelloWorld` file is the executable and the `HelloWorld.o` is the object file. The `HelloWorld.hi` file is an *interface file*. It contains information about the object file that `ghc` would need if you were to compile other modules, so that it would be able to link against that object file (said information cannot be stored in a standard object file).

You could say that the interface file is the equivalent of C's header files, only these are generated by `ghc` from the original Haskell source. Thus, the interface file is used when `ghc` compiles other modules, and the object file is used when linking all modules together to produce the executable. It is safe to delete the `.hi` and `.o` files once the executable is successfully generated, but keeping them helps in rebuilding and recompilation times after small changes.

3.2 Factorials implementation in Haskell

```haskell
factorial :: Integer -> Integer
factorial 0 = 1
factorial 1 = 1
factorial n = n * factorial (n - 1)

main :: IO ()
main = do
    putStrLn "Enter a number:"
    n <- readLn
    putStrLn ("The factorial of " ++ show n ++
              " is " ++ show (factorial n))
```

Listing 25: The program takes the number `n` as input, calculates and prints `n`!

The program in the listing 25 defines a main function that prompts the user to enter a number, reads the number from the console using the `readLn` function, computes its factorial using the `factorial` function, and prints the result to the console using the `putStrLn` function.

```
$ ghc factorial.hs
[1 of 1] Compiling Main ( factorial.hs, factorial.o )
Linking factorial ...
$ ./factorial
Enter a number:
5
The factorial of 5 is 120
```

Listing 26: Executing listing 25

3.3 Lambda functions

Real use of lambda functions, acting on lists.

As you can see in listing 27, Haskell provides a really easy way to create an new array with integers ranging from `a` to `b` (`b` included). Even high level languages like Python do not provide a similar feature. You can exclude `b` using the `..:` operator. These lambdas were implemented for demonstration purposes, Haskell already provides lambdas like `sum`, `product`, `maximum` and `minimum`.

The main function then creates a list called `myList`, calls the `sumList` function to calculate its sum using `foldr` and then prints out the result using the `putStrLn` function. The same thing happens for the `productList` function.

Note that we use the `show` function to convert the input list and the resulting sum to strings so that we can print them out using `putStrLn`.

3.3.1 The `foldr` function

In Haskell, `foldr` is a higher-order function that takes a binary function and an initial accumulator value and applies the binary function repeatedly to the elements of a list, starting from the rightmost element, and accumulating the result in the accumulator value. The type signature of `foldr` is shown in listing 29.

- The first argument is the binary function that takes an element of type `a` and an accumulator value of type `b`, and returns a new accumulator value of type `b`. The second argument is the initial accumulator value of type `b`. The third argument is the list of elements of type `a`.
- `foldr` works by applying the binary function to the last element of the list and the initial accumulator value, producing a new accumulator value. It then applies the binary function to the second-to-last element of the list and the new accumulator value, producing another new accumulator value. It continues this process until it reaches the first element of the list, at which point it returns the final accumulator value.

3.4 The `GHCi` Interpreter

To start a `GHCi` session use the command `ghci` (listing 30).

```haskell
-- Calculate the sum of a list of integers
sumList :: [Int] -> Int
sumList = foldr (+) 0
-- sumList = sum

-- Calculate the product of a list of integers
productList :: [Int] -> Int
productList = foldr (*) 1
-- productList = product

main :: IO ()
main = do
    -- Create an array of integers ranging from a to b (b included)
    -- let list = [a ..b]

    let list_a = [1.. 5]
    print list_a

    let list_b = [-4 .. 33]
    print list_b

    let myList = [1..10]
    putStrLn $ "list = " ++ show myList
    putStrLn $ "The sum of list is " ++ show (sumList myList)
    putStrLn $ "The product of list is " ++ show (productList myList)
    putStrLn $ "Max element of list is " ++ show (maximum myList)
    putStrLn $ "Min element of list is " ++ show (minimum myList)
```

Listing 27: Sum and product of an integer list

```
[1,2,3,4,5]
[-4,-3,-2,-1,0,1,2,3,4,5,6,7,8,9,10,11,12,
 13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33]
list = [1,2,3,4,5,6,7,8,9,10]
The sum of list is 55
The product of list is 3628800
Max element of list is 10
Min element of list is 1
```

Listing 28: Result of listing 27

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

Listing 29: The `foldr` type signature

```
$ ghci
GHCi, version 9.2.7: https://www.haskell.org/ghc/  :? for help
ghci>
```

Listing 30: Using the `ghci` interpreter.

Any **Haskell expression** can be typed at the prompt. The `ghci` interpreter will output the result.

```
ghci> 1+1
2
ghci> let x = 1 in x / 3
0.3333333333333333
ghci>
```

You can also bind values (or expressions) and functions to names.

```
ghci> x = 10
ghci> x
10
ghci> x = 3
ghci> x
3
ghci>
```

This is only supported in newer versions, since `GHC` 8.0.1. In older versions you are obliged to use `let` statement. Take note of the ability to overwrite `x`'s value.

Of course, you can also define new lambdas. Here is an example:

```
ghci> x = 10
ghci> addx y = y + x
ghci> addx 1
11
ghci> x= 11
ghci> addx 1
11
ghci> addx y = y + x
ghci> addx 1
```

```
12
ghci>
```

As you can see, even though we overwritten the value x, the `addx` lambda seems to hold the previous value of `x`. Even reassignments are a complex topic in Haskell.

Here is how someone would define the factorial function as a lambda using `ghci`:

```
ghci> factorial n = product [1..n]
ghci> factorial 10
3628800
ghci>
```

4 Conclusion

Haskell is a computer programming language renowned for its functional paradigm, type system, and purity, which distinguishes it from other programming languages [Mar13]. Haskell has a rich and storied history that reflects an ambition to create a smarter, more efficient, and easier-to-use functional programming language. It has lived up to this ambition, continuing to evolve and gain popularity in various industries, such as finance, web development, and academia, where its distinctive features are especially valued.

One of the essential and most notable Haskell features is its strict type system. This characteristic eliminates some of the most typical errors and catches them early in the development process, ultimately reducing the costs attributed to debugging and streamlining the software development cycle. Such advantage surpasses mere typing as the system allows for better documentation and significantly enhances the readability of your code. Moreover, Haskell's compiler can detect a wide range of errors that would otherwise be challenging to catch [Hut07]. Such characteristics make Haskell especially useful in the development of high-integrity systems where the cost of failure is high.

Another crucial aspect of Haskell is its non-strict evaluation, which makes it different from other programming languages. Non-strict evaluation allows the expressions to remain unevaluated until they are required. It promotes efficiency, reducing unnecessary computations.

One of the most significant advantages of Haskell is its purity, meaning that Haskell functions have no side effects. Such characteristics make code more predictable, easier to understand, and reduce the complexity of debugging and detecting errors commonly encountered in impure languages [Hut07].

Haskell has a highly flexible control structure, which allows for the creation of smaller and more readable codebases. Haskell uses unique identifiers that require less explanation and come naturally to the understanding of developers.

Haskell has a wide variety of operators that provide a wealth of functionality in a natural and mathematical way, which makes them easier to remember and apply to problems [Hut07]. Operators can also be composed to form a single function, an ability that sets Haskell apart from other programming languages. This ability offers a high degree of concision and expressiveness that can aid in the development of intricate software systems.

Lambdas and Monads are two of the most valuable Haskell features. Lambdas are anonymous functions that are declared using a backslash, allowing for flexible, on-the-fly function creations. Monads are a unique Haskell abstraction that facilitates the organization and composition of code for solving complex problems [Mar13].

Finally, the Haskell memory management and runtime are optimized to provide superior performance levels. Haskell uses garbage collection, eliminating memory leaks and other errors typical of languages that do not have garbage collection. This results in improved performance and more reliable software programs.

References

[23a] *Alex: A Lexical Analyser Generator*. Haskell, May 7, 2023. URL: https://github.com/haskell/alex.

[23b] *Cabal*. Haskell, May 22, 2023. URL: https://github.com/haskell/cabal.

[23c] *Commercialhaskell/Stack*. Commercial Haskell SIG, May 22, 2023. URL: https://github.com/commercialhaskell/stack.

[23d] *Happy*. Haskell, May 16, 2023. URL: https://github.com/haskell/happy.

[23e] *HUnit User's Guide*. hspec, May 22, 2023. URL: https://github.com/hspec/HUnit.

[Abe+98] H. Abelson et al. "Revised Report on the Algorithmic Language Scheme". In: *Higher-Order and Symbolic Computation* 11.1 (1998), pp. 7–105. ISSN: 13883690. DOI: 10.1023/A:1010051815785. URL: http://link.springer.com/10.1023/A:1010051815785.

[Bac+57] J. W. Backus et al. "The FORTRAN Automatic Coding System". In: *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability on - IRE-AIEE-ACM '57 (Western)*. Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability. Los Angeles, California: ACM Press, 1957, pp. 188–198. DOI: 10.1145/1455567.1455599. URL: http://portal.acm.org/citation.cfm?doid=1455567.1455599.

[Bac78] John Backus. "Can Programming Be Liberated from the von Neumann Style?: A Functional Style and Its Algebra of Programs". In: *Communications of the ACM* 21.8 (Aug. 1978), pp. 613–641. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/359576.359579. URL: https://dl.acm.org/doi/10.1145/359576.359579.

[Car23] Pierre Carbonnelle. *PYPL PopularitY of Programming Language Index*. 2023. URL: https://pypl.github.io/PYPL.html.

[CH] Koen Claessen and John Hughes. *QuickCheck: An Automatic Testing Tool for Haskell*. URL: https://www.cse.chalmers.se/~rjmh/QuickCheck/.

[Cha] Channable. *Lessons in Managing Haskell Memory*. Channable. URL: https://www.channable.com/tech/lessons-in-managing-haskell-memory.

[Chu32] Alonzo Church. "A Set of Postulates for the Foundation of Logic". In: *The Annals of Mathematics* 33.2 (Apr. 1932), p. 346. ISSN: 0003486X. DOI: 10.2307/1968337. JSTOR: 1968337. URL: https://www.jstor.org/stable/1968337?origin=crossref.

[Chu33] Alonzo Church. "A Set of Postulates For the Foundation of Logic". In: *The Annals of Mathematics* 34.4 (Oct. 1933), p. 839. ISSN: 0003486X. DOI: 10.2307/1968702. JSTOR: 1968702. URL: https://www.jstor.org/stable/1968702?origin=crossref.

[Dar77] John Darlington. *Program Transformation and Synthesis: Present Capabilities*. Imperial College of Science and Technology. Department of Computing and Control, 1977.

[Fag] Rolf Fagerberg. *DM22 Programming Languages*. URL: https://imada.sdu.dk/u/rolf/Edu/DM22/F06/haskell-operatorer.pdf.

[Fel+15] Matthias Felleisen et al. "The Racket Manifesto". In: (2015). Ed. by Marc Herbstritt, 16 pages. DOI: 10.4230/LIPICS.SNAPL.2015.113. URL: http://drops.dagstuhl.de/opus/volltexte/2015/5021/.

[Fra13] Andrew Frank. "Object-Orientation in Haskell: Exploring the Design Space". In: (Apr. 1, 2013).

[Gam19] Ben Gamari. *!972: A Concurrent Garbage Collector for the Old Generation ů Merge Requests ů Glasgow Haskell Compiler / GHC ů GitLab*. GitLab. May 17, 2019. URL: https://gitlab.haskell.org/ghc/ghc/-/merge_requests/972.

[Gam23] Ben Gamari. *Platforms - Glasgow Haskell Compiler / GHC*. GitLab. Jan. 18, 2023. URL: https://gitlab.haskell.org/ghc/ghc/-/wikis/platforms.

[Hac23a] Hackage. *Hackage*. 2023. URL: https://hackage.haskell.org/.

[Hac23b] Hackage. *Prelude*. 2023. URL: https://hackage.haskell.org/package/Prelude-0.1.0.1/docs/Prelude.html.

[Had23] Haddock. *Haddock 1.0 Documentation*. 2023. URL: https://haskell-haddock.readthedocs.io/en/latest/.

[Has] Hasura. *Instant GraphQL APIs on Your Data | Built-in Authz & Caching*. URL: https://hasura.io/.

[Has22] HaskellWiki. *Programming Guidelines HaskellWiki*, 2022. URL: https://wiki.haskell.org/index.php?title=Programming_guidelines&oldid=65285.

[Has23] HaskellWiki. *Haskell HaskellWiki*, 2023. URL: https://wiki.haskell.org/index.php?title=Haskell&oldid=65612.

[Hei21] Heisenbird - EOK, director. *Charles Hoskinson - Why Do Cardano Use the Programming Language Haskell? - ADA*. Apr. 25, 2021. URL: https://www.youtube.com/watch?v=1HGUu-EIb5g.

[Hic20] Rich Hickey. "A History of Clojure". In: *Proceedings of the ACM on Programming Languages* 4 (HOPL June 14, 2020), pp. 1–46. ISSN: 2475-1421. DOI: 10.1145/3386321. URL: https://dl.acm.org/doi/10.1145/3386321.

[Hud+07] Paul Hudak et al. "A History of Haskell: Being Lazy with Class". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL-III '07: ACM SIGPLAN History of Programming Languages Conference III. San Diego California: ACM, June 9, 2007. ISBN: 978-1-59593-766-7. DOI: 10.1145/1238844.1238856. URL: https://dl.acm.org/doi/10.1145/1238844.1238856.

[Hut07] Graham Hutton. *Programming in Haskell*. Cambridge, UK ; New York: Cambridge University Press, 2007. ISBN: 978-0-521-87172-3 978-0-521-69269-4.

[Ive62] Kenneth E. Iverson. "A Programming Language". In: *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference on - AIEE-IRE '62 (Spring)*. The May 1-3, 1962, Spring Joint Computer Conference. San Francisco, California: ACM Press, 1962, p. 345. DOI: 10.1145/1460833.1460872. URL: http://portal.acm.org/citation.cfm?doid=1460833.1460872.

[KL05] Oleg Kiselyov and Ralf Lammel. "Haskells Overlooked Object System". In: (2005).

[Kre15] Rob Kremer. *CPSC 449: Programming Paradigms*. Win. 2015. URL: https://kremer.cpsc.ucalgary.ca/courses/cpsc449/W2015x/.

[Lan64] P. J. Landin. "The Mechanical Evaluation of Expressions". In: *The Computer Journal* 6.4 (Jan. 1, 1964), pp. 308–320. ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/6.4.308. URL: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/6.4.308.

[Lan66] P. J. Landin. "The next 700 Programming Languages". In: *Communications of the ACM* 9.3 (Mar. 1966), pp. 157–166. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/365230.365257. URL: https://dl.acm.org/doi/10.1145/365230.365257.

[lib23] libraries at haskell dot org. *Haskell Prelude Documentation*. 2023. URL: https://hackage.haskell.org/package/base-4.15.0.0/docs/Prelude.html#g:12.

[Mar+08] Simon Marlow et al. "Parallel Generational-Copying Garbage Collection with a Block-Structured Heap". In: *Proceedings of the 7th International Symposium on Memory Management*. ISMM '08: International Symposium on Memory Management {co-Located with PLDI 2008}. Tucson AZ USA: ACM, June 7, 2008, pp. 11–20. ISBN: 978-1-60558-134-7. DOI: 10.1145/1375634.1375637. URL: https://dl.acm.org/doi/10.1145/1375634.1375637.

[Mar10] Simon Marlow, ed. *Haskell 2010 Language Report*. 2010. URL: https://www.haskell.org/definition/haskell2010.pdf.

[Mar13] Simon Marlow. *Parallel and Concurrent Programming in Haskell*. Sebastopol, CA: O'Reilly, 2013. ISBN: 978-1-4493-3594-6.

[Mar15] Simon Marlow. *Fighting Spam with Haskell*. Engineering at Meta. June 26, 2015. URL: https://engineering.fb.com/2015/06/26/security/fighting-spam-with-haskell/.

[McC60] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Communications of the ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/367177.367199. URL: https://dl.acm.org/doi/10.1145/367177.367199.

[Mic23] Simon Michael. *Hledger*. May 17, 2023. URL: https://github.com/simonmichael/hledger.

[Mit23] Neil Mitchell. *HLint*. May 21, 2023. URL: https://github.com/ndmitchell/hlint.

[MKR23] John MacFarlane, Albert Krewinkel, and Jesse Rosenthal. *Pandoc*. May 18, 2023. URL: https://github.com/jgm/pandoc.

[NT60] A. Newell and F. M. Tonge. "An Introduction to Information Processing Language V". In: *Communications of the ACM* 3.4 (Apr. 1960), pp. 205–211. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/367177.367205. URL: https://dl.acm.org/doi/10.1145/367177.367205.

[pan23] pandaquests. *Relationship between Lambda Calculus and Functional Programing*. Feb. 26, 2023. URL: https://levelup.gitconnected.com/relationship-between-lambda-calculus-and-functional-programing-5b5b44161c34.

[Ste84] Guy L. Steele. *COMMON LISP: The Language*. Burlington, MA: Digital Press, 1984. ISBN: 978-0-932376-41-1.

[TIO23] TIOBE. *TIOBE Index*. May 2023. URL: https://www.tiobe.com/tiobe-index/.

[Tur85] D. A. Turner. "Miranda: A Non-Strict Functional Language with Polymorphic Types". In: *Functional Programming Languages and Computer Architecture*. Ed. by Jean-Pierre Jouannaud. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1985, pp. 1–16. ISBN: 978-3-540-39677-2. DOI: 10.1007/3-540-15975-4_26.

[Wik21] Wikibooks. *Haskell/Understanding Monads — Wikibooks, the Free Textbook Project*. 2021. URL: https://en.wikibooks.org/w/index.php?title=Haskell/Understanding_monads&oldid=3833254.

[Wik22] Wikibooks. *Haskell — Wikibooks, the Free Textbook Project*. 2022. URL: https://en.wikibooks.org/w/index.php?title=Haskell&oldid=4194261.

[Yan] Edward Yang. *How the Grinch Stole the Haskell Heap : Ezyangs Blog*. Ezyangs blog. URL: http://blog.ezyang.com/2011/04/how-the-grinch-stole-the-haskell-heap/.

[Yan+15] Edward Z. Yang et al. "Efficient Communication and Collection with Compact Normal Forms". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. New York, NY, USA: Association for Computing Machinery, Aug. 29, 2015, pp. 362–374. ISBN: 978-1-4503-3669-7. DOI: 10.1145/2784731.2784735. URL: https://doi.org/10.1145/2784731.2784735.

[Yes23] Yesod. *Yesod Web Framework for Haskell*. 2023. URL: https://www.yesodweb.com/.