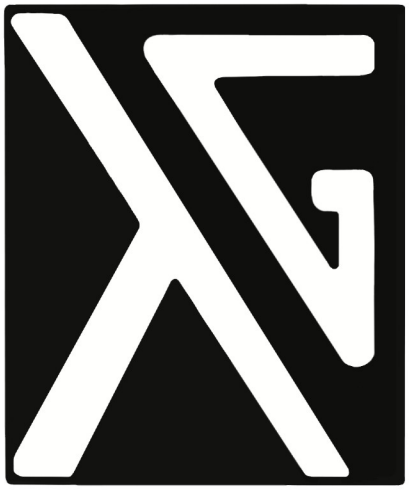

Replace the contents of this file with official assignment.
Místo tohoto souboru sem patří list se zadáním závěrečné práce.

Master's thesis

IMPLEMENTATION OF A STATICALLY TYPED, LAZY, PURE FUNCTIONAL PROGRAMMING LANGUAGE

Bc. Jan Sliacký

Faculty of Information Technology
Department of Theoretical Computer Science
Supervisor: Ryan Michael Culpepper, Ph.D.
June 23, 2022

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis: Sliacký Jan. *Implementation of a statically typed, lazy, pure functional programming language.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Contents

I would like to thank my supervisor, Ryan Michael Culpepper, PhD. for all his invaluable guidance, patience, and advice.

I also want to express my gratitude to Bára, my girlfriend, for her constant and boundless emotional support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on June 23, 2022 .....

Abstract

This thesis presents an implementation of a functional, statically typed programming language inspired by Haskell. It mainly focuses on the challenges of implementing the type system for such an language. The implementation is based on multiple resources covering implementations of different type system features. The thesis also covers other aspects of the language implementation—lexical and syntactic analysis, translation into a smaller functional core language, and non-strict evaluation.

Keywords pure functional programming language, type system, Haskell, type inference, bidirectional type analysis, higher-rank types, type classes, qualified types, laziness, higher kinded types, dictionary passing

Abstrakt

Tato práce se věnuje implementaci funkcionálního, staticky typovaného programovacího jazyka inspirovaného jazykem Haskell. Hlavním bodem zájmu práce je implementace typového systému pro tento jazyk. Implementace vychází z několika zdrojů zabývajících se implementací konkrétních aspektů typového systému. Tato práce se také zabývá dalšími aspekty implementace programovacího jazyka—lexikální a syntaktickou analýzou, překladem do menšího funkcionálního jazyka a tzv. non-strict evaluací.

Klíčová slova funkcionální programovací jazyk, typový systém, Haskell, typová inference, obousměrná typová analýza, typy vyššího stupně, typové třídy, kvalifikované typy, laziness, typy vyššího řádu

List of Acronyms

AST	Abstract Syntax Tree
DHM	Damas-Hindley-Milner
EBNF	Extended Backus-Naur Form
GHC	Glasgow Haskell Compiler
HRT	Higher-rank Types
REPL	Read Eval Print Loop
SYA	Shunting Yard Algorithm
WHNF	Weak Head Normal Form

Introduction

This thesis explores a few aspects of implementation of a typed functional language. It does so by implementing a statically typed, lazy, pure functional programming language called Glask. The language is heavily inspired by the Haskell programming language in all the important aspects of its design.

The produced implementation is not intended to be an industry-strength piece of software. Instead, both the implementation and the description given in the thesis aim to explore the fundamentals of the concepts involved. The goal is to draw from multiple different sources and present a work unifying all those sources into a single implementation.

In contrast to Haskell, Glask offers a bit richer syntax in the space of custom operators. The language allows the definition of custom prefix and postfix operators and it does not require wrapping them in so-called sections at call sites. This thesis covers a specific modification to the *Shunting Yard Algorithm* [1], first described by Edsger Dijsktra, that allows for this flexibility.

The main focus is put on the type system of the language. It represents the largest of all covered topics. The implementation of the type system is composed of multiple different sources each one describing a different concept isolated within a small language. The thesis adopts all those sources in their full extent, omitting no parts—*Typing Haskell in Haskell* [2], *Practical Type Inference for Arbitrary-rank Types* [3], *How to Make Ad Hoc Polymorphism Less Add-hoc* [4], and *Implementing Type Classes* [5].

The implemented language features a version of a non-strict evaluation strategy known generally as laziness.

The thesis also covers the transformation of the large surface-level representation to a simpler core featuring only a few concepts. The main focus is put on desugaring of type class-based overloading. In other words, type classes and their instances are lowered down into a simpler representation during a compile-time.

The evaluator is a simple AST walking interpreter operating on the core. Because the goal of the work is not to build a practical programming language, parts of the implementation involved in the evaluation are implemented in a very straightforward and simplified way. This means the implementation does not cover compilation or optimization or other related concepts.

The implementation is presented as a simple REPL that offers a set of features for interacting with the language. It can check kinds and types of corresponding terms. It can also print out the core representation of an inputted expression. Finally, it can load a file containing a program and evaluate expressions within its context.

Notation in Examples

This thesis features examples in various languages. Examples in Haskell or Glask are usually syntax highlighted and easy to identify.

EBNF

On the other hand, examples presenting the grammar of the language are written in *EBNF*. Sometimes the limitations of *EBNF* make expressing some forms an invariants hard or impossible. In those cases, the *EBNF*'s escape sequence is used. An example of that looks like this:

```
Prime  =  ? a prime number ?
```

The pair of question marks offers a way to escape into a "natural language" for a simpler description.

Extending Existing Definitions

Sometimes, when a definition of some already defined non-terminal needs to be extended it is better to not repeat the whole definition. Instead, giving just the extending part is much more concise. In those cases, the symbol for elipses is used at the beginning of the new definition. In practice, it might look like this:

```
Expression  =  ...
            |  Expression '::' Type
```

> The same symbol (**...**) is also sometimes used in the source code examples to symbolize an eluded term. However, it is always clear from the context, what meaning it has.

Outline of the Thesis

The thesis is split into four chapters. The first three chapters cover the language and its implementation. The last one summarizes the extent of the implementation and states exactly what has been implemented and to what level. It serves as an evaluation of the work done.

The chapter 1 starts with a small functional language—lambda calculus. It then extends the language by adding new concepts in multiple steps. The chapter is written as an informal specification for the language. It explains how the concepts in the language work from point of view of the user.

It covers the syntax of the language, notion of types, type inference, custom data types, higher-rank polymorphism, various fixity operators, type classes, typed holes, and laziness.

The chapter 2 offers a high-level strategy for implementing non-trivial features of the language. It focuses on the representation of those problems, leaving the specific algorithms to the next chapter. Its structure closely follows the structure of the implementation pipeline.

It starts with the lexical and syntactic analysis and tools used for their implementation. It covers the specific extension to the well-known algorithm that this thesis introduces to support various fixity operators. Then it follows with the implementation of the type system.

The explanation starts with a small *Damas-Hindley-Milner* type system, covering the concept of type inference and unification. It covers the challenges of type analysis for programs composed of a mix of unannotated and annotated bindings. It also covers the topic of type analysis for (mutually) recursive definitions.

The topic of custom data types and type analysis of them is also covered in a detailed way as well as the kind inference which becomes necessary at that point.

The following extension adds qualified types and type classes. The corresponding section also covers topics of ambiguity and type defaulting. It also discusses a strategy for type-checking class members.

The final part of the type analysis extends the language with higher-rank types. This changes the type system from being inference-based to being implemented as a bidirectional analyzer and extends its expressive power.

The strategy chapter also covers the topic of desugaring class-based overloading. It explains the so-called *placeholder approach* and how it fits into the system.

The final part of that chapter covers the evaluation. It explains how the language is lowered down into a smaller core representation and describes the fundamental aspect of our lazy evaluation.

The chapter 3 contains examples of specific and the most interesting aspects of the implementation. It features examples of parts of the implementation when implementations from different sources interacted in interesting ways and a few examples of more involved parts of the implementation.

The final chapter 4 covers the implementation status of various features. It also presents how the result of the implementation can be evaluated in an automated and simple way.

Chapter 1
Language Specification

Functional Language

Glask is a functional language. It resembles *lambda calculus* closely, with two sets of extensions [6]. The first set can be characterized as mere syntactic extensions, the second one as extensions to the model of the language. In this chapter, we are going to build the language from the concept of lambda calculus extending it one concept at a time until we reach the complete form.

1.1 Lambda Calculus

We start with just three syntactical constructs (and an auxiliary one). Together they represent the entirety of the lambda calculus. The following grammar illustrates that succinctly:

```
Expression  =  identifier
            |  '\' identifier '→' Expression
            |  '(' Expression Expression ')'

            |  '(' Expression ')'
```

The definition above implies that most of the syntactical constructs in the language will be in the form of an expression. That is very common in functional languages. However, from the practical point of view, it is very useful for the language to allow some way of declaring new bindings. In our language, the global declaration is not an expression. It has the following form:

```
Declaration =  identifier '=' Expression
```

Such form allows us to write some simple programs like:

```
id = \ i → i

result = id id
```

A program as a sequence of declarations:

```
Program  =  { Declaration }
```

1.2 Scoping Rules for Declarations

All declarations are defined in one, shared scope. This means, that not only will later declarations see all the previous ones but the other way around too. As for the expression on the right-hand side of the **=**, it is also inside that same scope. Because of that, the body of the declaration can refer not only to all other declarations but to its own declaration too.

This makes it possible to use unrestricted recursion. This is an aspect of the language (and its future type system) that will be preserved throughout the whole implementation. We will certainly give more attention to this topic in future chapters, especially those related to type analysis and interpretation.

In the next step, we extend both parts of the definition with two kinds of addition. We add a few new constructs and extend the existing ones for convenience.

The extended version of the language looks like this:

```
Expression  =  identifier
            |  literal
            |  '\' { identifier } '→' Expression
            |  'let' variable '=' Expression 'in' Expression
            |  Expression operator Expression
            |  Expression Expression { Expression }
            |  '(' Expression ')'
```

```
Declaration =  identifier { identifier } '=' Expression
```

Literals

One of two extensions to the model of the language is the addition of literals and primitive operations. The second one is a let expression. Everything else just extends the syntax. In the following few paragraphs we will give both an explanation of all the new constructs and a demonstration of how they translate back to the pure lambda calculus.

With the first extension, we introduced a notion of simple values. For now, there are just two types of literals—numeric literals and characters. For the time being, we will treat them as built-in primitives. That approach will go well with the same treatment of *operators*. Later in this chapter, we will change that and define both of them more formally.

Lambda Functions with Multiple Parameters

The next one allows us to write lambda functions with many parameters. The way it translates back to lambda calculus is quite straightforward.

```
\ a b c → Expression
```

Is equaivalent to:

```
\ a → \ b → \ c → Expression
```

Let Expressions

The following extension is a new form of binding and is usually called **let** expression. It does not have a translation, because it is a new primitive construct. Later in this chapter, we will

extend **let** with additional sub-constructs. That will explain why it can not be desugared into a simpler language. For now, we define it as a new language primitive.

It works like this: variable binds an expression on the right-hand side of the **=** operator, this binding is then available within the scope of the expression on the right-hand side of the keyword **in**. It is precisely the last part of the whole expression which is the result of the evaluation of this syntactic form.

Our current notion of **let** is very simple and limited. We can, for instance, only bind one identifier at a time and we can not use the *function declaration* syntax from global declarations. That is just for the sake of simplicity and brevity. Later we will extend **let** to allow the binding of many even mutually recursive definitions, possibly in the function declaration form.

Primitive Operations

As we have mentioned above, the introduction of primitive operators qualifies as an extension of the language model. At this point we only have binary, infix operators like `+` or `/` and so on. In the future, we will make them a more standard part of the language as opposed to them being a special case in the grammar.

Applications

To simplify the function application a little, we have allowed applying the function expression to as many values as is possible. It is no longer necessary to group two expressions at a time to make application to many arguments work. We have also added a notion of *infix operation*. An example of that was already shown above in the **let** *desugaring* explanation. Later on, we will extend the concept of the application, be it a function application or an operator application, even more by adding operators with different fixities. We will also unify the two applications together in a way, that will allow us to write sequences of expressions and operators with as few parentheses as possible. We will discuss that in more detail in future chapters. Until we explain the exact details of their implementation and semantics, we will treat operators as kind of built-in operations. Later we will show how they are simply just a syntactic "sugar" for ordinary functions.

Declarations

Finally, the last extension is giving declarations a similar treatment as to lambdas. The idea of that extension is roughly equivalent to that of lambdas with multiple parameters.

```
fun a b = a + b
```

Is equivalent to:

```
fun = \ a b → a + b
```

Which can be further reduced to:

```
fun = \ a → \ b → a + b
```

This new syntax for declarations is called *function binding syntax*.

1.3 Types

Even though the current scope of the language is quite small, we have already made it possible to write expressions, and transitively, programs, which do not make sense. Consider for example the following expression:

```
let fn = 'c' in fn 23
```

Running it would result in an error because `'c'` is not a function, therefore it can not be applied to a numeric literal. This kind of problem is only a single example representing a whole class of problems. Statically typed languages, like Glask, deal with them by using types. Having a static type system allows to rule out a large set of erroneous programs during the compilation—even before the program is executed.

In Glask, every expression has a (single) type. There are *primitive types* like **`Int`** or **`Char`**, and there are *function types*. The function type is a type of function which takes some value on the input and produces some value on the output. The type then consists of those two values denoting the intended usage of that function. Here is an example illustrating the idea:

```
23  :: Int

'c' :: Char

(\ n → n)  :: Int → Int
```

We use **`::`** to denote the fact that the *term* on the left, has a *type* on the right. To denote a function type we use a symbol **`→`** positioned in between the *input type* and the *output type*. It is worth mentioning that the arrow symbol **`→`** is right associative. Therefore type terms like **`Int → Int → Int`** are equivalent to **`Int → (Int → Int)`**.

However, a type system built just on the notion of primitive types and functions between them is not expressive enough. It does not let us reuse the same implementation for values of different types. The following, for example, is prohibited:

```
id x = x

a = id 23

b = id 'c'
```

Even though the `id`—or *identity* function—is obviously a good fit in both cases. The reason for this limitation is exactly the fact, that every term in the language has a *single* type. In the example above, once the `id` was applied to the numeric literal, its type was assumed to be **`Int → Int`** and as such, it can not be applied to the character literal too. This might seemingly imply that the assumption was wrong, but that is not really the case. The actual problem is the simplicity of the *type language*. There is no notion of *polymorphism*, in other words, there is no way to express a type of a function, which is flexible on the type of its argument. To obtain that notion, and make it possible to type the example above, we extend the language of types with two more constructs—*universal types* and *type variables*. Those two are closely related, and immediately make it possible to denote a type of a polymorphic function. It is best to demonstrate with an example:

```
id :: forall i . i → i
```

Here the `id` is the same function as in the invalid example. The `forall` keyword represents the universal quantifier ∀ and is followed by a sequence of identifiers—those are called *type parameters*—they represent many concrete types at the same time, making it possible to abstract

over specific parts of the whole type. What follows after the `.` is the type of the value with all "flexible" parts abstracted away. Our example then means that `id` is a function, taking a single value of any type and returning a value of that same type.

Now we can give the full language of types in the following figure:

```
Type  =  ? primitive type like Char or Int ?
      |  identifier
      |  Type '→' Type
      |  'forall' { identifier } '.' Type
      |  '(' Type ')'
```

The figure above implies that one could write some rather unusually looking types like:

```
(forall x . x → x) → Int → Int
```

That is indeed true. The ability to put universal quantifiers anywhere within the type term significantly widens the expressive power of the type language. This idea is called *higher-rank polymorphism* and we will cover it in the future section of this chapter, for now, we limit ourselves to working with types with only a single, left-most, outer-most universal quantifier—like in the case of the `id` function.

1.4 Type Inference

One of the most notable features of our type system so far is the ability to type the whole program (assuming it is well-formed) without forcing the programmer to explicitly mention types anywhere. This notion can serve as a foundation of our understanding of what the *program* is. We will say that any sequence of declarations is a (valid) program if and only if our type system can properly type it. Being able to type some code means that it can reconstruct the types for all parts of code. The way that it is done is using a concept known as a *type inference*. The type inference is a discipline based on finding types of terms of the language with no, to very little help. In the following sections, we will explore this topic in more detail, talking about the specific amount of "help" our type system needs to validate our programs in terms of them being well-formed.

1.4.1 Damas-Hindley-Milner Type Inference

The specific type system and the corresponding algorithm for type inference, that will be the basis of our type system is known as *Damas-Hindley-Milner* [7] [8] [9].

One of the most important properties of the *DHM* type system is the (global) inference. This allows for having a fully statically type-checked language with no type annotation whatsoever. That alone is an interesting and desirable property in practical and statically typed programming languages. Some of our future extensions will bring more expressive power to the type system and from the user standpoint, it will usually be paid for by making the type annotations mandatory at some places throughout the code. However, that is a trade-off that we are willing to make. It has been proven by Haskell authors and users that those trade-offs are advantageous in the end, especially since most of the time it is between not being able to express something at all and being able to, but having to write a small number of type annotations. In that regard, we follow the footsteps of Haskell, specifically *GHC* [10].

1.5 Type Annotations

One of the previous sections introduced types as a standalone concept. What is currently missing, however, is the formalism for using types inside type annotations. As a side note—one of the advantages of Glask is that it does not require writing type annotations in most places. With only a few exceptions it utilizes global type inference to fill in the types for the programmer. Some of those exceptions have to do with *type classes* and higher-rank polymorphism—concepts that will be discussed in future chapters. With that being said, it is very useful for the language to allow the use of type annotations for bindings. Programmers might utilize them for documentation purposes. They might be also useful in the context of type error messages. Hence we extend the grammar for expressions and declarations as follows:

```
Expression  =  ...
            |  Expression '::' Type
```

```
Declaration  =  ...
             |  identifier '::' Type
```

We also impose an invariant on the types inside type annotations—they are always required to be closed. In other words, if there is a left-most, outer-most `forall` it must quantify all type variables from the type. However, similarly to Haskell, there is an option to not write the left-most, outer-most `forall` and let it be *implied*—in those cases, all free type variables within the type are quantified by the implicit `forall`. This rule is known to Haskell programmers as the `forall`-*or-nothing rule* [11]. The following example illustrates both use-cases:

```
const :: a → b → a
```

is equivalent to the "explicit" variant:

```
const :: forall a b . a → b → a
```

And in case of a little bit more complicated example:

```
const :: a → (forall b . b → a)
```

is equivalent to:

```
const :: forall a . a → (forall b . b → a)
```

From now on we will mostly omit the explicit `forall` for the sake of simplicity. It is the custom in Haskell and we adopt it for Glask too.

1.6 Custom Data Types

So far the only data types in the language are primitive values, like numbers and characters. To make the language a little bit more practical we introduce user-made data types. We first give an example of a simple declaration and follow with an explanation after that.

```
data Bool = True | False
```

This is how we define the well-known **`Boolean`** type together with its two possible values. The single **`data`** declaration introduces both type-level constant **`Bool`** and expression level constants

True and **False**. The latter two are called *constructors* and they are used to do just that—to construct—or in this case to represent values of their corresponding type, **Bool** in this case.

As mentioned above, constructors do not have to be just constants, they can take arbitrary numbers of arguments. To make them do that, we need to write types of those arguments after the corresponding name of the constructor within the data declaration. Like so:

```
data Answer = None | Number Int | Letter Char | Both Int Char
```

Constructors that take arguments are generally treated the same way as ordinary functions.

The addition of data declarations illustrates an important point. There are two namespaces: one for expressions, one for types. Each namespace accommodates two kinds of inhabitants. Members of the namespace whose name starts with a capital letter are considered *constants* whereas those starting with lower case letter are considered *variable*.

This means, that data constructors are always denoted with a capital letter at the beginning, whereas ordinary functions and variables start with a lower case letter. A similar is true for the language of types—type constants like **Bool** or **Int** start with a capital letter and type variables always start with a lower case letter.

Our current grammar for both expressions and types is still correct. We only need to acknowledge the changed notion of the *identifier*—from now on, it no longer represents only a variable, but also a constant—be it a type level constant or an expression level constant.

1.6.1 Working with Data Structures

In the previous section, we introduced the concept of custom data types and constructors for creating *data values*. To make the explanation of custom data complete we now describe the discipline necessary to use them in any other way than just passing around. The discipline in question is called *case analysis* and in Glask it is done by *pattern matching*. The main idea of pattern matching will be explained a bit later in this chapter. For now, we will focus our attention on the concept of the case analysis. We first need to extend our grammar with a new construct called *case expression*. Here we give its syntax as an extension to the `Expression` form:

```
Expression  =  ...
            |  'case' Expression 'of' { Pattern '→' Expression }
```

The **case** expression allows defining zero or more "branches" to pattern match. In regards to the notion of the single "branch"—it consists of a pattern followed by the symbol for arrow **→** and an expression. Its purpose is to try and match the value using the pattern on the left and if it succeeds, evaluate the expression on the right. In regard to the scoping, each "branch" is separated from others in the sense, that if the pattern binds one or more values, it is only the expression on the right-hand side of the **→** which has the access to those bindings. Another important aspect of the **case** expression is that once any branch successfully matches the value, the whole case is resolved. In this regard, it does not have a "fall-through" semantics as might be the case for similar-looking constructs in other languages. For those reasons, the ordering of the "branches" very much matters and can have an impact on evaluation if patterns in those "branches" overlap. Speaking of overlapping patterns, it might be a good practice to always make sure that at least one of the "branches" will match and execute. If it were not, the whole **case** would raise a runtime error. In other words, **case** expression must always cover every actual run-time value. If no "branch" matches, the evaluation results in an error. Since Glask does not have a notion of exceptions and a way of handling errors, any such error will cause the termination of the whole program.

1.6.1.1 Examples of Case Analysis

The concept of case analysis represents a core concept of a *control flow* in languages that offer it. It can be used to make all sorts of other constructs more flexible and ergonomic. We will discuss most of these opportunities later in this chapter. Before that, we show how **`case`** expression can be used in practice:

```
data Bool = True | False

data Answer = A | B | C

isA :: Answer → Bool
isA an = case an of
           A → True
           B → False
           C → False
```

The example above shows how **`case`** makes it possible to identify the value by the constructor it was constructed with. In the future section on pattern matching we will go into more detail regarding the specific rules of constructing patterns and what are their semantics. The last topic of this short section is how can our **`case`** expression be specialized to introduce another control flow mechanism, an **`if`** expression. Consider the following example:

```
data Bool = True | False

boolToInt :: Bool → Int
boolToInt b = case b of
                True → 1
                False → 0
```

From the example it becomes pretty obvious that we can introduce **`if`** expression as a syntax "sugar" for **`case`** expression. To help with the intuition, here is how such desugaring may work:

```
if <bool expression>
  then <positive branch>
  else <negative branch>
```

is equivalent to:

```
case <bool expression> of
  True → <positive branch>
  False → <negative branch>
```

To conclude the section on **`if`**, we also update the production rule for `Expression`:

```
Expression  =  ...
            |  'if' Expression
                 'then' Expression
                 'else' Expression
```

> Note that the indentation in the production rule is only there for better readability.

It would seem like everything around **`if`** expression is quite simple and straightforward. There is one thing to keep in mind, however. Since **`Bool`** type is not a primitive, built-in construct and

at the same time, the **if** strictly requires the condition to have a **Bool** type (there is no "soft" notion of truthiness like in other programming languages). This means that even though the actual **Bool** type might be defined in the "user space", within some core library, it must always be so. Simply, the type checker very much depends on the **Bool** being defined. As Glask does not have a notion of modules as of right now, we must always make sure that **Bool** is defined correctly within the program.

We now interrupt the explanation of case analysis to first extend our notion of custom data types. Once we describe them to their full extent, we will continue the description of the case analysis through a pattern matching.

1.6.2 Higher Kinded Data Types

With the current state of the language, we should be able to express quite a lot of programs already. Our data types, however, are significantly limited. We have a similar problem with constructors as we had with functions before we introduced polymorphism. We can not, for example, define a "generic" data type that would "consume" value of any type given to it. The solution to this limitation is to introduce the notion of polymorphism to types too. This is done by allowing the data declaration to quantify over type variables similar to how the `forall` quantifier does it. The only difference is that we do not need to introduce any new keywords. We just write one or more type variables behind the name of the type we are defining. All data constructors are then able to refer to them as they define the types of their arguments. One of the simplest examples is the **Maybe** type defined as:

```
data Maybe a = Nothing | Just a
```

Here the declaration of **Maybe** quantifies over a single type variable `a`, this makes it possible for the data constructor—**Just**—to refer to it as the type of its argument. The type of the **Just** constructor is hence `forall a . a → Maybe a`.

Not only does this extension allow us to define parametrized types, but it also introduces a completely new concept of a *type application*. The type application is similar to the ordinary *expression application*. The only difference is it being part of the language of types. Its meaning is quite simple—it denotes a type constructor being applied to one or multiple types—its arguments—to form a complete type. It should be pretty apparent, that values only have types that are "fully applied", in other words, we can not have a value of type **Maybe** the same way we can not have a value of type **List**, that notion does not even make sense. It must always be a *List of something*, like **List Int** and so on.

However, this invariant is not expressible through the grammar of the language of types. We can not easily prohibit writing definitions like:

```
foo :: a → Maybe → a
foo = ...
```

That example does not make sense because the type is not well-formed. We now face a similar issue as with the language of expressions, when we realized that there are some programs, which just do not make sense and would fail during execution. The solution to the original problem was the introduction of the type system as a means to prevent admitting ill-formed programs. The solution now is going to be almost the same—we will introduce kind of a type system for the language of types. We will classify *type terms* depending on whether they represent a concrete type—like **Int** or **Answer** or whether they represent a type constructor which yet needs to be applied to the right number of type arguments—like **Maybe**.

Since the problem we are trying to solve is rather small and simple, compared to the problem we solved by types, the solution will reflect that. We will introduce a notion of *Kinds*, or *types of types* and there will be just two constructs to describe a *kind* of any type term. There is a

kind `*` pronounced *Type* and there is a notion of a *kind function*, denoted `→` which is used to represent a function between the kind on the left of the arrow to the kind on the right of it. For example, our **`Maybe`** has kind `* → *` because it is supposed to be applied to a single type and then it forms a proper *Type*, like **`Maybe Int`**. There can be types with more complicated kinds, like this type:

```
data Foo m = F (m Int)
```

In this example the type variable `m` is applied to **`Int`** to form a *Type*. That means that the `m` must have kind `* → *` implying that the **`Foo`** itself has kind `(* → *) → *`.

This might raise a question of a kind polymorphism. Let us consider the following example:

```
data Poly m x = P (m x)
```

In simple words—whatever the actual two type terms are, they must "fit" together. That is really the only obvious requirement. The `m` could have kind `* → *` and `x` would have to be just a *Type*. Or they could have more complicated kinds. However, whatever the `x` is, the `m` must accept it and upon being applied to it, it must result in a type of kind `*` aka *Type*.

All of that would technically be true and it might work that way in our language, but the topic of *kind polymorphism* is not covered by this thesis. Details related to the kind system and its implementation will be given in future sections.

1.6.3 Record Syntax in Data Constructors

Following the approach of Haskell, we introduce a special syntactic extension for data constructors to support a *record syntax*. Records in Haskell, and therefore Glask, are just a lightweight syntactic "sugar" for ordinary data constructor syntax. The idea is that one can give a unique name to the specific constructor argument and use that name as a getter of that specific field from the constructed value. Here is an example:

```
data Item a = Item { identifier :: Int, designation :: a }

getId item = identifier item

ident = getId (Item{ identifier = 42, designation = 'a' })
```

The simple idea is, that even though we write `identifier :: Int` the actual `identifier` that gets generated, has a type **`Item a → Int`**. It is worth noting that those generated "getter" functions might be partial if the data structure defines multiple constructors and not all of them have a field with that specific name. That can easily happen as there is no requirement on the data declaration to define its constructors using one way or the other—single data declaration can feature both shapes at the same time, depending only on what the programmer needs to achieve.

Construction is then done using those field descriptors on the left side of the **`=`** and the actual value for that field on the right. One thing worth noting is that while in Haskell one might omit one or more fields during the construction and those fields will automatically be bound to the `bottom` value, the same is not allowed in Glask. In other words, Glask requires that records are constructed by supplying all of the required values.

> Since those generated "getters" are inserted into the global scope, once some name is used by a record field in one data declaration, it must not be used in any other, or even as an ordinary function.

Working with (Higher Kinded) Data Structures

With custom data structures extended to their full power, we may now resume our introduction to case analysis and pattern matching. We proceed by giving a description of pattern matching, followed by a section about the pattern matching being used in different parts of the language. Our goal is to utilize it in place of all binders to make dealing with data structures truly ergonomic.

1.6.4 Pattern Matching

The main idea of pattern matching is quite straightforward. It syntactically mirrors closely the syntax of the way the value *might have been created*—by applying one of the constructors to the correct amount of arguments. So to *deconstruct* or *pattern match* on such a value, one needs to write a *pattern term* which consists of a specific constructor being applied to just the right number of *pattern terms* representing the arguments. We now give the full grammar of patterns to serve as a visual clue in the later parts of this section:

```
Pattern  =  variable
         |  '_'
         |  constant { Pattern }
         |  literal
         |  '(' Pattern ')'
         |  Pattern '::' Type
```

Coming back to our explanation of the case analysis on user-defined data structures—we pose an important invariant upon patterns—constructors inside patterns must always be fully applied. In other words, they must be followed by as many patterns as they accept arguments. The reasoning behind this is simple, while it is possible to partially apply the constructor, and obtain a function that expects to be given some more values before it returns an actual data structure, it simply does not make sense to attempt the same when deconstructing the value. This, for one, is the reason why we must be careful when parenthesizing patterns, we must never put a pair of parentheses in such a way that it would isolate one or more pattern arguments from their constructor. For instance, given this data declaration `data Pair a b = Pair a b`, the following pattern would be illegal `(Pair 23) 42`. In this regard, patterns are much more sensitive to parentheses than any other part of the syntax.

1.6.4.1 Rules of Pattern Matching

In this short section, we will give a description of the semantics of pattern matching from the evaluation perspective. In general, any pattern can either match or fail to match. If the pattern matches, it binds none or some values from within the value being matched on. For example, suppose that we constructed a value of the type `Maybe Int` by applying the `Just` constructor to the value `23`, pattern like `Just num` would not only succeed but would also bind the value `23` to the variable named `num`, which would make it available in the current scope. This makes patterns another kind of *binders* together with *lambdas* and `let` expressions.

Variable Pattern

The *variable* within a pattern matches any value at all. As was mentioned above, while doing so, it will bind that value to its name and make it available to the corresponding scope (depending on how the pattern is used). Here is one example of that:

```
case 2 + 3 of
  number → number * 2
```

This expression evaluates to `10`.

Wildcard Pattern

The *wildcard* is a special kind of pattern. Similarly to variables it also matches anything at all, but it does not introduce any new bindings into the scope. The meaning of the wildcard is to simply acknowledge there is a value. At the same time we do not care about the value even a little bit. The example of that might look like this:

```
case 2 + 3 of
  _ → True
```

> Because wildcard does not introduce any new bindings we do not get access to the expression (unless it was already bound to some local variable).

Data Pattern

All of the details about data patterns were already given in the text above. Here we give some simple example to complete the explanation:

```
data Maybe a = Nothing | Just a

fromMaybe :: a → Maybe a → a
fromMaybe defaultVal m = case m of
                           Nothing → defaultVal
                           Just a → a
```

Literal Pattern

This simple pattern is used to match on primitive values, like numeric literals, characters and possibly other built-in primitives. The following example shows how:

```
num :: Int
num = ...

... case num of
      1 → '1'
      2 → '2'
      3 → '3'
      _ → '_'
```

1.6.5 Using Patterns

In this section we are going to describe how can patterns be used to make the handling of values in our language even more natural. Glask, as a language inspired by Haskell, aims to give the same level of flexibility when working with (not only) data structure. In Haskell, almost every binding site allows the use of patterns. Beside the **`case`** expression, they can be used in **`let`** expression, or inside lambdas and global function declarations to deconstruct/restrict its arguments. It is

worth noting that all of those extra constructs where we are going to utilize pattern matching are just syntactical extensions to the same underlying model. The last extension to the model was the addition of the **`case`** expression together with the notion of patterns, but all of the following extensions to the language will be purely syntactical.

1.6.5.1 Patterns in Binders

As mentioned above, Glask allows patterns to appear at all binding sites. This extends the grammar by a fair bit. However, all previously syntactically correct programs are still correct under the new extension—simply because where there was originally just a *variable*, it now can be any valid pattern—and a variable is a one. In the following section, we will go over all newly extended constructs and explain both how they change syntactically and what is the new form equivalent to, in terms of "desugaring".

1.6.5.2 Pattern Matching in Function Declarations

When it comes to global declarations, they can utilize the concept of pattern matching even more than just matching on function arguments. We can take inspiration from the **`case`** expression and introduce a new form of syntax, allowing us to split the single function declaration into multiple parts. The idea behind that is quite simple—if we want to split the function definition into two parts, where each one only handles a specific subset of the inputs, we can employ the same behavior as for the **`case`** expression. That is, the pattern matching goes one option at a time and stops when it succeeds. In other words—if a function with multiple equations is called, the evaluation proceeds by trying each equation (from top to bottom) by pattern matching on each argument (from left to right) until it successfully matches on all positions. Here too the execution can fail if the matching fails for all present equations.

To give a better idea, here is the new grammar for `Declaration`:

```
Declaration  =  identifier { Pattern } '=' Expression
             |  identifier '::' Type
```

There is an "external invariant" imposed on the declarations which can not be expressed in the EBNF form we are using. That is—if we chose to split up the definition of our function into two or more parts, we must keep them together in a single "group", in other words, we can not interleave two or more such "groups" with each other.

Before we present examples of both valid and invalid uses, we must cover the translation semantics necessary to support this extension. One possible translation may look like this:

```
identifier  pattern-A1  pattern-A2  '=' expression-A
identifier  pattern-B1  pattern-B2  '=' expression-B
```

Is equivalent to:

```
identifier  variable-1  variable-2  '='
    'case' variable-1 'of'
        pattern A1 '→'
            'case' variable-2 'of'
                pattern-A2 '→' expression-A

        pattern-B1 '→'
            'case' variable-2 'of'
                pattern-B2 '→' expression-B
```

Now we give an example of a valid definition split into multiple so-called *equations*:

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

which is equivalent to:

```
factorial :: Int → Int
factorial m = case m of
                0 → 1
                1 → 1
                n → n * factorial (n - 1)
```

For clarity, we have also included the type signature. It is, however, not necessary for the type signatures to be directly next to the group of equations.

As a side note, our valid example is also well-behaving in the sense that it covers all possible cases. No matter what the actual value of the argument is going to be, there will be a pattern which will match.

In contrast to that, here is an example of a definition which is not well-formed for multiple reasons:

```
vowel :: Char → Bool
vowel 'a' = True
vowel _   = False

foo x = x

vowel 'u' = True
```

The apparent mistake is the presence of `foo` in the middle of the "group" of equations for `vowel`. But even if we were to remove it, or to move it elsewhere, the declaration would still not be totally fine—the last equation matching the argument with a literal `'u'` comes only after the equation which will match against anything at all, this means that the whole equation is considered *redundant*, and we should probably reorder those equations or at least remove the last one to not cause any confusion. All of that is left to a programmer to decide, the compiler gives us a complete freedom to order our equations in any way we please.

1.6.5.3 Pattern Matching in Let Expressions

Our current notion of **`let`** is rather restricted when compared to the global *function declarations*. Now might be a good time to lift up those restrictions from before and allow **`let`** to introduce as many local bindings as necessary. Under the same treatment of local **`let`** declarations as those at the global level, we can now also define local functions, split their declarations into multiple equations, and give the type signatures for them using the already existing syntax. This extension is pretty straightforward and we will not give it more attention.

1.6.5.4 Pattern Matching on Lambda Arguments

So far, all the constructs which allowed pattern matching had a way to be "split" into many smaller, more specific, "branches" of which each would handle a specific part. The only exception to that rule is lambda. Since lambda functions can also feature patterns as a way to destructure their arguments, we must be careful to always cover all the cases within the single lambda, because there is simply no way to "split" a lambda into many "equation-like" definitions. Pattern matching in lambda functions should therefore be used only while paying utmost attention and

avoid causing a failure to match on the arguments and consequently raise an error which would result in termination of the whole program. The new syntax for a lambda expression now looks as follows:

```
Expression  =  ...
            |  '\' { Pattern } '→' Expression
```

Since we have already allowed to write lambdas with multiple arguments, that first extension applies to this new one too. However, there is a point in showing the "desugaring" in terms of simple, one-argument lambdas. It makes the notion much cleaner:

```
'\' Pattern '→' Expression
```

Is equivalent to:

```
'\' new-variable '→' 'case' new-variable 'of'
                         Pattern → Expression
```

> The `new-variable` needs to have a fresh name that will not collide with any already existing one.

By generalizing the notion above, we will easily be able to translate lambda functions with multiple arguments too.

Record Syntax in Patterns

As has been mentioned earlier, records are just a syntactic extension of the language and can be easily "desugared" away into ordinary constructors. That is, however, completely hidden to the programmer so they can use records in some interesting ways. The first one is in pattern matching. Suppose we have a data constructor with many fields and we want to get a value of just one of them. We can easily do so like this:

```
data LargeRecord a b = LR { w :: Int, x :: Char, y :: a, z :: b }

justZ (LR{ z = zVal }) = zVal
```

The syntax of the record pattern is very similar to the construction of the value. One thing worth noting is that it is possible to use the same name on both sides of the `=` in the record pattern. The second occurrence then overshadows the first one, so if we write for example this:

```
justZ (LR{ z = z }) = z
```

It will still work fine.

The second interesting consequence of the accessors being technically global identifiers is that we can have so-called *labeled update*. Consider the following example for illustration:

```
data Record = Record { a :: Int, b :: Char }

updateA int (Record { a = aVal, b = bVal })
  = Record { a = int, b = bVal }
```

There is an easier and nicer way to write the same using the labeled update syntax. Compare:

```
updateA int rec = rec{ a = int }
```

The version above is much shorter, but they are completely equivalent, one can actually expect the compiler to generate code similar to the original version if it is given the shorter version. The only limitation is that there always must be at least one field being updated. This feature is a direct consequence of the fact that each field can only be used in a single data declaration, because of that, the compiler can easily recognize what data structure is being updated and generate the necessary code for us.

1.7 Higher-rank Polymorphism

We have mentioned before, that the grammar allows us to write explicitly quantified types inside other types, and we have decided to restrict ourselves by not writing them—in other words, up until now we were only allowed to write type annotations with a single outer-most, left-most explicit forall or with no explicit universal quantifier at all. In this section, we are going to lift that restriction and explain what the newly gained flexibility brings to the table. Before we do that, let us first discuss some examples of simple programs we might want to write in our language.

Suppose that we want to define a function, which would take a lambda as its argument and apply it to two values of different types. The corresponding fragment of code may look like this:

```
data Pair a b = Pair a b

apply fn = Pair (fn 'a') (fn 42)
```

This of course requires the `fn` argument to have a polymorphic type. But aside from that, we might be inclined to think, that there is nothing wrong with the code and we could easily utilize that function by applying it to for instance `id` function. That is, however, not the case here—this code raises an error, both in Haskell and Glask. The reason for that is the way the type inference works. The problem might become a bit more apparent if we write the accompanying type annotation for `apply` ourselves:

```
apply :: forall a . (a → a) → Pair Char Int
```

Now it might become a bit more clear what is wrong with this piece of code. What we want is to define a function that accepts a (polymorphic) lambda and applies it to values of different types. That is, however, not what the type says. Instead it could be read as: "`apply` is a function that takes a lambda function that for a *specific type* takes a value of *that type* and returns a value of *that type*."

The source of the problem is exactly the *for some type* part—it means that the function `apply` is not as polymorphic as we would need. The *DHM* allows each type parameter to become at most *one specific type* with each use of the function—for that reason, the lambda given to `apply` can not have both **`Char → Char`** and **`Int → Int`** types.

In other words—only types of *some bindings* get generalized and function's arguments are not among those. The solution to it would be to somehow extend the type system so that it can understand that the original piece of code should be accepted; higher-rank polymorphism is exactly the way to go about this. The point of this section is to make the type checker accept the original example while requiring only a small change to it. We will need to write the type annotation for the `apply` function by hand to guide the type analysis process and make it understand that the lambda given to the `apply` will have a compatible type with what we are

trying to use it for. To make the connection between what was said at the beginning of this section, we give the full correct version of the code now:

```
data Pair a b = Pair a b

apply :: (forall a . a → a) → Pair Char Int
apply fn = Pair (fn 'a') (fn 42)
```

There is a point to be made about the impact of the change. The meaning of the type has changed significantly. To make the change obvious let us write the same type annotation in a more explicit way—even though it is not the way it would usually be written:

```
apply :: forall . (forall a . a → a) → Pair Char Int
```

We added the left-most, outer-most explicit forall. The reason why it would usually not be written like this, is the absence of any type parameter within that explicit forall. However, the presence of it now makes it clear that the inner universal type `(forall a . a → a)` is "nested" inside the complete type. That leads us to the second important point—to make this type annotation legal we need to lift our restriction stating that foralls must not appear *inside* types. So from now on, let us lift that restriction.

1.7.1 Higher-rank Types

Since we lifted the previous restriction and can now write types consisting of universal types it is worth going into a bit more detail regarding the classification of the newly allowed types. We introduce a term *Rank* of a type. The *rank* of a type describes the depth at which universal quantifiers appear contravariantly [12] :

Monotypes τ,σ^0 ::= a | $\tau 1 \to \tau 2$
Polytypes σ^{n+1} ::= σ^n | $\sigma^n \to \sigma^{n+1}$ | $\forall a.\sigma^{n+1}$

1.7.2 Limitations of Higher-rank Polymorphism

With the restriction on foralls lifted we must address what happens when we try to write programs like this one:

```
id i = i

foo :: (forall a . a → a) → Int
foo fn = 23

x = id foo
```

This program is not admitted by either Haskell's or Glask's type checker. The reason for the rejection is the fact that the type of `foo` has a non-zero rank and as such, it can not be passed to the `id` function. This might come as a surprise, considering that `id`'s type is (modulo alpha renaming) `forall a . a → a`—in other words, `id` takes *anything* and returns that thing. This simple example then might look like premature rejection. But there are fundamental concepts at play. The type system which would allow higher-rank types to be unified with type variables would be *impredicative* and implementation of such extension would require much more than just making it rank-polymorphic. *GHC*, the implementation of Haskell, used to support impredicative polymorphism with the use of the extension called *ImpredicativeTypes* but that extension is now deprecated (even though it is still available for backward compatibility). For

that reason, *impredicative polymorphism* is not a part of the current design. That also means that we can not construct a (completely) polymorphic data structure containing a value with a higher-rank type. Here is an example of that:

```
data Value a = Val a

foo :: (forall a . a → a) → Int
foo fn = 23

x = Val foo -- error
```

It is worth noting that since data constructors are basically ordinary functions we can make the example above work if we make the **`Value`** little bit less flexible:

```
data Value = Val ((forall a . a → a) → Int)

foo :: (forall a . a → a) → Int
foo fn = 23

x = Val foo
```

To conclude the explanation of higher-rank types—they are a source of great flexibility to the language and type system. But we should always pay attention to what parts of the program are using them and in what ways, a rule which happens to be true about almost any concept in programming.

1.8 Prefix, Infix, and Postfix Operators

In this section, we are going to address a specific limitation regarding operators in our language. Up until this point, operators were treated as built-in primitives which just "came to be" for the sake of the specific example. In this section, we are going to remedy just that. We are going to standardize operators and give them the same treatment as was given to functions—we will allow the definition of custom operators of various fixities.

We start by extending the definition of a variable and a constructor. From now on, we will consider operator identifiers escaped with a set of parentheses to be a variable. But only if that operator name does not start with `:`. The colon symbol is reserved for identifying so-called *operator constructors*.

To give a few examples for better intuition, we consider `+` to be an operator, but `(+)` is technically a variable. Simply because parentheses around an operator "lift" it into an expression. This makes it equivalent to a variable like `value`.

On the other hand, `:` and every other operator starting with that character is an infix constructor operator. When escaped, like `(:)` it also becomes an expression.

At the end of this section, we want to be able to define and use infix operators, prefix operators, and postfix operators, and we also want to be able to use ordinary functions in infix notation using the same lexical escaping as Haskell. That is—function name wrapped in backticks automatically becomes infix operator.

We also want to make sure, that programmers have to use as few parentheses as practically possible. That means we need to introduce a "protocol" for operator declaration to communicate how should each operator combine and interact with others in case there are no parentheses that would conclusively disambiguate the expression.

We approach this issue the same way Haskell designers did—we require each *operator declaration* to state what fixity (`in`/`pre`/`post`), associativity (`left`/`right`/`none`), and precedence (0–9) the operator does have. These properties are enough to decide how they should interact in actual expressions and whether their combination is allowed at all.

> When we talk about associativity, we do not mean the mathematical notion—the property of an operation to be parenthesized in either direction. For example, the following expression $a+b+c$ might be disambiguated both as $(a+b)+c$ or $a+(b+c)$ under the mathematical notion of associativity. Instead, we use the term associativity—with the specific `left/right/none` qualifier in mind—to indicate how the expression must be disambiguated during the parsing process. For the previous example, if we say that `(+)` is right-associative, it means that `a + b + c` is understood exactly as `a + (b + c)`. The converse applies for left-associative `(+)`. If it has been declared as non-associative, the original expression is illegal.

To make the notion of associativity even more useful, we extend this behavior to interactions between different operators with the same precedence too. Suppose we have `(+)` and `(-)` with the same level of precedence. Then this expression `a + b - c` can be disambiguated depending on the associativity of both operators. If both are left-associative, we get `(a + b) - c`. If both are right-associative, we get `a + (b - c)`. If at least one of them is nonassociative, the expression is illegal. And finally, if they do not have the same associativity, the expression is also illegal. With the introduction of prefix and postfix operators, it is possible to extend this notion to those too. We get a way to combine all three kinds of operators with the same precedence level in very specific ways. This will be described in more detail in later parts of this section.

Let us take a look at the example of how operator signatures look like:

```
prefix 9 ▷
infixr 5 +
postfix 9 !

... -- a and b are defined here as well as the operators

expr = ▷ a + b !
```

From the declaration at the top, we have all the necessary information to correctly parse the expression being bound to the variable `expr`. Since `( ▷ )` and `(!)` have higher precedence than `(+)` they will bind more tightly to the values they are being applied to. Therefore the expression in the example is equivalent to `( ▷ a) + (b !)`.

We have mentioned above, that there is a way to make ordinary functions behave like operators. The way to do that is quite straightforward—we wrap any ordinary function name within backticks. Like `` `foo` ``. This instructs the parser to treat that whole term as an operator. Since they become just ordinary operators, it is possible to give *fixity signature* for them too. The following fragment of code is therefore perfectly valid:

```
infixr 5 `plus`
postfix 9 `negate`

-- or even
prefix 0 `print`
```

The first declaration is how Haskell allows those "infix function operators" to be specified, the following two are unique to Glask. In any case—they bring more flexibility to the language. Take for example the `print` "operator" defined above. Because of that prefix signature, we can write expressions like `` `print` a + b ``. It might take some time to get used to it, but having a very weak prefix printing function can be especially useful.

The flexibility comes at a cost, however. The notion of first-class *prefix* and *postfix* operators introduces some new issues. Take for example the following fragment of code:

```
prefix 5 ▷
infixr 5 +

...

▷ a + b
```

This piece of code would be invalid because both operators have the same precedence, but the prefix one is not right-associative. Since our goal is to be really flexible, we decided to add the notion of associativity to unary prefix and postfix operators too. That solves the issue above but introduces a different one—how should the following fragment be understood and parsed:

```
prefixl 5 ▷

...

▷ ▷ ▷ a
```

If the operator `( ▷ )` (pronounced "triangle") was *right associative* as one would expect from a prefix operator, everything would be fine, however, it might happen, that a programmer will define *left associative prefix* operators and *right associative postfix* operators. To address that we introduce an **implicit rule** to our parser. This rule overrides the expected associativity for operators and implicitly disambiguates the previous example as `▷ (▷ (▷ a))`—the reasoning is simple—it is the only disambiguation that makes sense. We will give a precise description of the rule in chapter 2.3.1. This rule allows us to write even fewer parentheses but comes at a cost—it introduces some strange semantics when we attempt to combine specific operators together.

Consider this fragment of code:

```
prefix 9 ▷
prefixl 6 +
prefix 3 ⊢

...

▷ |▷ a + b
```

At first sight, it might seem, that because the `( ▷ )` has the highest precedence, it should bind the tightest. In this example, however, the complete opposite is true. The expression at the bottom of the fragment is understood as `▷ (⊢ (a + b))`. The reason for that is the implicit rule. It can be understood like this: the `( ▷ )` while being the strongest, is stuck behind a much weaker prefix operator, for that reason, it will have to wait until that operator `(⊢)` finally captures its operand and only then that sub-expression can be captured by the `( ▷ )`. But since the `(+)` is stronger than `(⊢)` it gets to capture `a` first. This is only one of the simpler examples of the impact the implicit rule has.

If Glask's goal was to become a real-world and practical programming language, its flexible syntax would probably have to be restricted in some significant way to avoid needlessly complicated expressions which's meaning could change significantly depending on many factors.

1.8.1 Constructor Operators

Glask has the same notion of *operators as constructors* as does Haskell. Any operator which starts with a colon is considered an *infix* constructor. Those might be very useful, especially for data types representing collections and various relations. We have made the decision to not attempt

to make constructor operators more flexible in terms of fixity. Probably the biggest reason for that was the fact, that we would either need to reuse the colon-at-the-beginning for prefix and postfix constructor operators too, or we would have to introduce a new special symbol (probably) per each variant. Each option had its downsides and since those prefix and postfix operator-constructors would get used even less than their function counterparts we decided against it.

1.8.2 Escaped Operators

The special-purpose backtick notation for functions has a counterpart within operators. Operators, independent of fixity, can be escaped by wrapping them in a pair of parentheses. Such escaping lifts them into ordinary values and they can be both applied to their arguments in a kind of "prefix" way or passed as arguments to other functions or operators. Here is a simple example

```
(+) 23 42
```

1.8.3 Operator Declarations

To complete the process of unifying the usage and declaration of standard functions and operators, we explain how operators can be declared. The specific syntax for it is resembling the syntax of the function definition. Functions are always defined in a pre-position kind of way. The declaration starts with the name of the function, which is then followed by the (possibly empty) sequence of patterns. Operators are defined just like that, first, they need to be escaped with a pair of parentheses just like when they might be used in a "prefix notation", the rest of the declaration is the same as for the functions, even their type annotations work like that. Here is a simple example of that:

```
infixr 0 $

($) :: (a → b) → a → b
($) fn a = fn a
```

In Haskell, one can write those in a bit more flexible way. For instance `(+)` might be defined as:

```
infixl 6 +

(+) :: Int → Int → Int
a + b = ...
```

And the same infix syntax can be utilized for "backticked" functions too. In that regard, our syntax might seem rather simple and inflexible. In practice it might not be such a big issue, functions and operators are always defined at one place and used many times over the whole program—therefore it seems to be more important to give the flexibility where it has a chance to be utilized more often.

1.9 Type Classes

In this section, we describe another fundamental feature of the language, which will extend the set of features related to polymorphism—*type classes*. Type classes are a way to express a standardized interface that any member of that type class must implement. For example

our primitive numeric operator `(+)` is actually implemented as a *method* in the **`Num`** type class. Because of how type classes work, they allow us to declare an "interface" containing methods and constants. If we later want to implement that interface for our specific type, we must declare that type to be an **`instance`** of that type class and give corresponding implementations of all methods and constants.

Here is the grammar necessary for both classes and instances:

```
Declaration
          =  ...
          |  ClassDeclaration
          |  InstanceDeclaration
```

```
ClassDeclaration
          =  'class' [ SuperClasses '⇒' ] constant variable
                [ 'where' { Declaration } ]
```

```
SuperClasses
          =  constant variable
          |  '(' { constant variable } ')'
```

```
InstanceDeclaration
          =  'instance' [ InstanceContext '⇒' ] constant InstanceType
                      [ 'where' { Declaration } ]
```

```
InstanceContext
          =  ? empty ?
          |  constant variable
          |  '(' { constant variable } ')'
```

```
InstanceType
          =  constant
          |  '(' constant ? zero or many distinct variables ? ')'
          |  ? constant applied to zero or many unique variables ?
          |  ? tuple type containing only distinct variables ?
          |  '[' variable ']'
          |  variable_1 '→' variable_2
```

Here is a small example of a simplified version of the **`Num`** class:

```
class Num a where
  (+) :: a → a → a
```

It tells the compiler that we wish to declare a new type class called **`Num`** which is parametrized by some type `a`—that is a reference to the future type which is going to be an instance of that class—like **`Int`**. It also says that instances of **`Num`** should implement a single method—operator `(+)`. The important part here is that type classes are only allowed to specify type signatures of their methods and constants. Instances, on the other hand, are only allowed to specify implementations of those, never type signatures.

Here is what the **`instance`** declaration for **`Num`** might look like:

```
instance Num Int where
  -- the int#+ is low-level primitive function for Int addition
  (+) a b = int#+ a b
```

1.9.1 Class Hierarchy

Any type class can specify zero or more superclasses. This is a quite useful way to model the idiom of building new functionality on top of an existing abstraction. Take for example Haskell's type classes related to numbers—its **`Num`** class defines operations like *addition* and *multiplication*, but does not define *division*. The division is only implemented later in the hierarchy by specific type classes like **`Fractional`**, where it is called `(/)`, or **`Integral`**, where it is called `div`. It might be worth noting that **`Integral`** itself is not an immediate sub-class of the **`Num`**, instead, it depends on two other classes—**`Real`** and **`Enum`**. That makes it an example of a class that depends on more than one *superclass*.

To denote that one class is a subclass of another class, we use the symbol **⇒**. The part on the left is the (possibly empty) set of superclasses and the part on the right is the class being declared. The symbol itself is pronounced *then*.

When it comes to **`instance`** implementation, the dependence of one class on another does not change the way instances are declared. Each instance declaration is only concerned with implementing methods and constants from one class. Here is an example of it:

```
class Num t where
  plus :: t → t → t

class Num t ⇒ Integral t where
  div :: t → t → t

instance B Int where
  div m n = int#div m n

instance Num Char where
  plus x y = int#+ x y
```

The example above illustrates nicely the intuition behind instance declarations even for type classes depending on other type classes. It is rather different from the notion that some object-oriented programming languages introduce. In our case, it is not so much about the dependent type class requiring the complete implementation inside its instance, as it is about the requirement for all instances of **`Integral`** to also be instances of **`Num`**.

There is an important invariant when it comes to type class hierarchy. The hierarchy in the form of a dependency graph must be acyclic. For that reason, the following example will not get accepted by the type checker:

```
class A t ⇒ B t

class B t ⇒ A t
```

The example also shows that if we want to define a type class that does not declare any methods or constants, we can just ignore the **`where`** block altogether.

1.9.2 Qualified Types

So far we have introduced the concept of type classes as a way of so-called ad-hoc polymorphism, they allow us to define methods and constants with the same name, but different behavior as

opposed to the standard polymorphic function—which is a single function guaranteed to work with values of any type.

The impact of type classes on the type system is rather significant. Because the fact that some specific type belongs to some specific type class needs to be accordingly reflected at the type level, we must modify the language of types and our notion of it.

This means that methods of type classes can only be used in such ways that the types of the values which they are being used on (or with) are compatible with the type class declaration defining those methods. This requirement is expressible at the level in the form of a collection of *predicates* which then form the *type context* of a specific type.

Here is a simple example to illustrate the point:

```
class Num a where
  (+) :: a → a → a

...

fun x y = x + y
```

In this example, the type of `fun` must reflect the fact, that `x` and `y` can be arguments to the `(+)`. In other words, they must have the same type and that type must be an instance of **`Num`** type class. The way it is reflected in the type of the `fun` is by *qualifying* the type variables representing a type of its arguments. In this case, it is going to be just one type variable since they both have to be of the same type. The corresponding explicit type annotation will look like this:

```
fun :: Num a ⇒ a → a → a
-- or with the explicit quantifier:
fun :: forall a . Num a ⇒ a → a → a
```

The part on the left-hand side of the **⇒** is called *type context* and it is a sequence of *predicates*. The predicate is composed of two things, the first one is the name of an existing type class and the second one is a type variable. The predicate represents the fact that said type variable stands for a type that is an instance of that type class. On the right-hand side of the **⇒** is just the type as we know it.

1.9.3 Type Contexts

It is useful to think of type contexts as a way of representing a *requirement* for the universally quantified type variables. Type contexts in our language restrict the type which is being qualified by requiring it to be an instance of some type class. Here is the full grammar for type contexts:

```
Context  =  Class
         |  '(' { Classes } ')'
```

```
Class    =  constant variable
         |  constant '(' variable Type { Type } ')'
```

To put it simply, we can only qualify type variables—**`Num a ⇒ ...`**, or a type, which consists of a type variable (on the role of type constructor) being applied to one or more types—**`Foo (m a Int) ⇒ ...`**. The potential use of the former is rather obvious, let us focus on the latter for a bit. The second form might be used when we want to qualify over a parametrized type

like **Maybe** or **Either**. The interesting part is that although the rules for instance declaration prohibit us from defining an instance like:

```
class Foo a

instance Foo (Maybe Int)
```

> All instance types must be of the form (T a1 ... an)

We can actually write a type context that looks very similar to the illegal form, the difference can be explained quite simply—while we want our instances on parametrized types to be totally flexible on the type parameter, we can have our type contexts to qualify over types parametrized by a specific type. Simply because if the corresponding instance is flexible on the parameter, there is no reason why the parameter could not be any valid type of our choice.

Here is an example illustrating the point above:

```
class Add m where
  add :: m → m → m

instance Num t ⇒ Add (Maybe t) where
  add (Just a) (Just b) = Just (a + b)
  add ...

fun :: Add (m Int) ⇒ m Int → m Int → m Int
fun ma mb = add ma mb
```

We have to point out that the example above is only to illustrate what is possible in the type system, it does not seem like the piece of code above would be a useful pattern in the real world. However, it shows that if we have the knowledge of some useful details, we may use it to our advantage and make the type signature of `fun` a little bit more specific than what the more conservative version might look like:

```
fun :: Add m ⇒ m → m → m
fun ma mb = add ma mb
```

To further iterate the example and make it more realistic while simultaneously showing more from the language, here is the same idea revised:

```
class Add m where
  add :: Num a ⇒ m a → m a → m a

instance Add Maybe where
  add (Just a) (Just b) = Just (a + b)

fun :: Add m ⇒ m Int → m Int → m Int
fun ma mb = add ma mb
```

> As a side note: Without the explicit type annotation, the type of `fun` is inferred to be `(Add m, Num a) ⇒ m a → m a → m a`.

The last version of the example shows two interesting concepts. The first one is the fact that type class instance ought not to be just types of kind *Type*, they can very well be higher kinded type constructors—**Maybe** in this example. In this case, the way the **Add** is declared forces all

instances to be higher kinded. Specifically, they must be of kind `* → *` because that is how it appears in the type annotation for `add` method. The second interesting thing is the fact that we can qualify methods (and constants for that matter) within the type class declaration. This is going to be the case at all places where we were able to write types up until this point.

1.9.4 Type Contexts and Explicit Type Annotations

One of the important consequences of having type contexts is that within explicit type annotations given by the programmer, the type context must always cover all the requirements of the corresponding expression. It is, for example, not allowed to only specify just a few of the predicates and leave the rest for the inference to find. That would result in an error. In a way, it is very similar to the situation with explicit type annotations and polymorphism. One can never write a more polymorphic type than what the actual inferred type is. In that sense type context is a form of making the type less polymorphic therefore omitting any necessary predicate would result in an error.

1.10 Type Synonyms

Sometimes it might be useful to give some existing type a specific name or abstract in an interesting way some of the type arguments an existing higher kinded type. The goal is not to define a new data type, but instead use the ones that are already present in the system under a better fitting name or with a simpler interface.

Glask offers so-called *type synonyms*. They offer exactly that functionality. They are simple lexical aliases for existing types—data types and other synonyms.

Here is an example showing the use of a simple type synonym:

```
data Maybe a = Nothing | Just a

type ReportedAge = Maybe Int
```

The keyword **`type`** is used to define a new type synonym.

There are some restrictions, however. A type synonym must always be defined in such a way that its definition does not create a cycle with itself or any other type synonym. An example of a cyclic type synonym definition looks like this:

```
type First = Second

type Second = First
```

This is not allowed as it would make it possible to write type definitions that can not be simply expanded before the type analysis takes place.

The second restriction forces us to always apply type synonyms to the right number of type arguments. That is—as many types as many type parameters its definition declares.

This limitation makes it impossible for type synonyms to function as some kind of type functions. A practice that would extend the type system with higher-order features, possibly rendering the type unification undecidable, non-unitary, or both [2].

1.11 Typed Holes

In this section, we describe a feature of the language which greatly extends its interactivity—*typed holes*. Typed holes are quite useful when we want to get some *type clues* from the compiler.

If we are not sure what exact expression should we write at some place in the program, we can fill it with a so-called *hole* and let the type checker infer the type of it, which might give us some clues. Consider the following example:

```
identity :: a → a
identity x = _
```

Upon running this code through the type checker it will tell as that the hole should have the type `a` where `a` is a bound variable from the type annotation. But typed holes are useful even without type annotations, they can take advantage of the type inference in finding out what is the type of the expression they represent.

We can use the "anonymous" version of a hole by just writing _ as an expression or we can give it a name by following the _ with a valid variable identifier. Like so:

```
_result
```

1.12 Laziness

Glask employs a non-strict evaluation model. It differs from most other programming languages in the sense that the evaluation happens only if the result is needed. The other part of the laziness property is the ability to do some level of memoization—an expression is not evaluated multiple times if it is not necessary.

When we want to be sure that we have evaluated the expression fully, we must do what is called *deep forcing*. Deeply forcing the value will attempt to fully, deeply evaluate the lazy value into its *normal form*. However, this sort of *deep* evaluation is usually neither necessary nor useful. Most of the time we will need just the value to reach a so-called *weak head normal form* (WHNF). That is—only forcing the value until we encounter a data constructor. At that point, the evaluation is suspended, which means that the values inside the constructor are not evaluated until they are needed. From now on, when we mention *forcing* we are referring to this weaker, less eager notion of the operation.

Writing programs in a lazy language has an interesting consequence—we can take advantage of it by constructing infinite data structures without causing the program to hang at run time, at least under the assumption that we will not attempt to traverse or consume in any other way the entire infinite data structure. A simple example of a well-behaving program utilizing the laziness might look something like this:

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

repeat :: a → List a
repeat x = Cons x (repeat x)

take :: Int → List a → List a
take 0 _ = Nil
take _ Nil = Nil
take n (Cons x xs) = Cons x (take (n - 1) xs)

expr = take 5 (repeat 4)
```

The short program above successfully finishes even if we force the evaluation of the `expr`. It is an illustration of the fact that unless we attempt a non-terminating recursion, we can have infinite data and consume them piece by piece. Not all infinite values can be safely consumed, however. Here the very important part is the fact that the constructor **`Cons`** can hold two unevaluated

values, in other words, arguments to constructors are not forced during the construction of the data structure. On the other hand, if we ever attempt something like the following, we might get into an infinite loop, but we will certainly not get the behavior we aimed for:

```
num = 5 + num

expr = num + 1
```

Forcing the evaluation of `expr` will cause the evaluation to hang or perhaps the integer to overflow, that very much depends on the specific implementation. In any case, it is obviously not a well-behaving program, unless we intended to write a malfunctioning program. It is therefore always important for the programmer in a lazy language to know and understand the evaluation model. Laziness empowers programmers to easily achieve tasks that might otherwise be hard to express in programming languages without it, but must not be seen as a simple and harmless concept. Instead, it should be viewed as a feature of the language as any other—one that can be useful but also easily misused.

1.13 Purity

Usually, when we say that a programming language is *pure*, we mean that it does not have unattended mutations. In such a language, every value is immutable and the output of any function depends only on its input.

The immutability does not tend to be an issue from the point of implementation of functional programming language. The situation tends to get a little bit more complicated when it comes to so-called *side effects*.

One of the most important side effects is the ability, of the program, to communicate with the outside world. That includes reading from any sort of input and producing any sort of output.

Practical pure languages use different approaches to handle effects. Haskell programs, for example, use a specific **`IO`** monad for doing side effects that interact with the outside world [13]. However, implementing it is a non-trivial task.

Glask has a way around it. Because it only lives in its REPL, it is possible to directly type the input into the live environment and use Glask code to operate on it. This gives us the required level of interactivity and also delays the design decisions about the implementation of effects to the future.

Chapter 2
Implementation Strategy

This chapter aims to describe implementation details related to the language described in the previous chapter. The structure of this chapter is going to closely relate to the implementation pipeline:

- We start with the implementation of the lexer and parser, or lexical and syntactical analysis.
- We will then move on to the largest part of this project—the type system.
- In the next section, we give some details about the translation of the larger surface language into a smaller version of the language—our *core*.
- Finally, we will describe the evaluator.

Some of the parts of the implementation, especially the type system, were implemented by following a couple of publications from Haskell authors. We will give some context about adopting implementations from those publications as well. This thesis, however, will not attempt to cover the entirety of the content of each publication—there are some significant details, that are best described in those papers and we will not attempt to re-explain or re-define them here. Instead, we will introduce them, show some basic examples and point at the specific resource in case their exact definition would be beneficial for the reader.

2.1 Lexer

For the lexing and parsing, we have picked two specific tools to generate the code for both the lexer and the parser. The same tools are used by the *GHC*'s developer team for lexing and parsing Haskell. The tool for lexing is a lexer generator *Alex* [14]. To define a lexer we must write a specification in the form of a translation grammar based on regular expressions, and let *Alex* process it and generate a Haskell module that implements the lexer for us. The definition of the lexer is rather simple and fits within two hundred lines of code. Specific details are not of major interest to us. The only really interesting thing about the lexer is that it uses a monadic interface to communicate with the parser. That way they both share the same state. The explicit state also allows the lexer to "stream" tokens "on-demand". In simple terms, the lexer is generated in such a way that it reads and produces only one token at a time and is invoked by the parser whenever it demands the next token—be it a look-ahead or otherwise. This turns out to be very useful, for example when implementing the off-side rule. In any other sense, the lexer is ordinary and brings no surprises.

2.2 Parser

Our parser is generated by the *Happy* [15] parser generator. It requires us to write a description of the grammar in such a way that for each production rule we also write a small snippet of Haskell code which will be used to create the correct data representation when invoked by *Happy*. Here is a simple example:

```
Literal   ::  { Literal }
          :   integer                 { Lit'Int $1    }
          |   double                  { Lit'Double $1 }
          |   char                    { Lit'Char $1   }
```

> Where we assume **`Lit'Int`**, **`Lit'Double`**, and **`Lit'Char`** constructors being defined and in scope.

Happy also allows us to write type annotations for non-terminals in most cases, which is quite useful for documenting production rules like the one above. What is also quite useful is the ability of *Happy* to generate a parser that uses the same monad as the lexer. This way they can interact rather closely if it is necessary. Aside from implementing the off-side rule, it might also be useful for better error reporting or more flexibility in general if it is ever needed.

The complete process of syntactical analysis is split into two parts. The first part produces an almost completely parsed tree representation of the program. What is then left to do is to parse function and operator applications. The main reason for delaying that part is because Glask allows users to define their own operators with complete control over their syntactical semantics. We therefore can not know the correct meaning of an expression until we have seen all the operator signatures. In theory, this might be the right place to utilize the monadic state of the parser. It could be used to store the information about user-defined operators as they are discovered. We have, however, decided to split the syntactical analysis into two parts. This way we can isolate the process of parsing function/operator applications from the rest of the parsing. This approach will make the second stage more open to extensions.

To summarize, the first part of the syntactical analysis is done by the generated parser and produces a tree-shaped structure representing the program. It represents applications as sequences of expressions instead of binary nodes. The second part of the syntactical analysis operates on this structure in addition to some meta information. An example of one such meta information is the complete set of operator signatures, set of record field-constructors, and other useful information. That information is necessary during the second stage, be it for correctly parsing applications or desugaring record syntax into ordinary constructor syntax—an operation that is trivial and does not deserve further description.

2.3 Parsing Prefix, Infix, and Postfix Expressions

As was established in section 1.8—Glask's syntax allows not only user-defined infix operators but prefix and postfix ones too. In Haskell, there is some limited version of prefix and postfix operators using so-called *operator sections*, but they require parentheses around each use [16]. Unlike Haskell, Glask has both prefix and postfix operators as a first-class concept. They can be defined and used without the need for sections or operator escaping.

However, Glask parses operators with a similar approach to the one Haskell uses. That is—use a known algorithm to process the precedences and associativities of various-fixity operators. The algorithm of our choice was invented by Edsger Dijkstra and is known as the *Shunting Yard Algorithm* (*SYA*) [1]. The original algorithm needs to be modified to support all three positions of operators, together with function applications in the lambda calculus style. Namely, all the

different ways our various-fixity operators interact (not only) with each other need to be taken into an account.

The introduction of prefix and postfix operators gives rise to a whole class of ambiguities. Those come from the interaction between infix and both postfix and prefix operators.

Before we go and investigate those ambiguities, we will preface with a related idea. There is a point in having a notion of associativity for prefix and postfix operators from the usability standpoint. It is the same reason why we might want to define infix operators with no associativity at all. To disallow interactions between operations that should not be combined. Take for example an *equals operator*, written `(==)`, in Haskell, it is defined as not associative, so that expressions like:

```
a == b == c
```

are syntactically not allowed. For that same reason, it might make sense to give prefix and postfix similar treatment.

Let us now move on to the ambiguity-related part. Consider the following example:

```
▷ a + b
```

Let us assume that both operators have the same level of precedence. That alone does not tell us how the expression should be understood. It might be parenthesized "from the left", or "from the right"—or using left associativity, or right associativity. In those cases it would look like this:

associativity	equivalent form
left-associative	`(▷ a) + b`
right-associative	`▷ (a + b)`

The meaning of each version is completely different and it becomes very important to know what associativity each operator has. If they are both left-associative, the `(▷)` is applied first, if they are both right-associative, it is the opposite. Finally, if they have different associativity or at least one of them is non-associative, it results in an error.

There is another simple expression that might advocate for the notion of associativity of unary (prefix and postfix) operators. See what happens if we combine prefix and postfix operators with the same precedence levels:

```
▷ a ! -- where (!) is postfix
```

Clearly, this expression must either be illegal or we need to introduce some mechanism to make it unambiguous. That is precisely the reason why we have defined associativity for unary operators too.

2.3.1 The Implicit Rule

There is another class of problematic expressions that needs to be taken care of. The issue is not in the expressions being ambiguous, instead, it comes from an emergent property of operators. Let us preface the motivation with an example very well known to most Haskell programmers:

```
fn a b $ gn x y
```

The operator `($)` is defined as follows in Haskell:

```
infixr 0 $

($) :: (a → b) → a → b
($) f x = f x
```

It follows that the expression can be parenthesized and understood like this:

```
(fn a b) $ (gn x y)
```

The point we are making is that `($)` splits the application into two parts. The only reason why it happens is because an ordinary function application has higher precedence than any operator in Haskell. However, when we introduced prefix and postfix operators with user-definable precedence, we made it possible to define prefix and postfix operators which act in the same way in regard to infix operators. In other words, if there is a prefix or postfix operator with low-enough precedence interacting with infix operators it might split the expression, sometimes even in surprising ways.

Here is an example of that:

```
prefixl 3 ▷
infixl 6 +
infixl 7 *

a * ▷ b + c
```

Because of the low precedence of `( ▷ )`, the way this expression is understood is as follows:

```
a * (▷ (b + c))
```

And that is even though the `(*)` has higher precedence than `(+)`.

It is worth noting that in our language we can define even more useful variant of `($)` than it is possible in Haskell. The reason for it is because `($)` does not work with other infix operators. Here is an example of a syntactically invalid expression in Haskell:

```
a * $ b + c
```

It will not be understood as `a * (b + c)`, instead it will raise an error. Within our language we can define our own version of it as a prefix operator and write the expression like this: `a * ▷ b + c`, which will be understood as `a * (b + c)` for `( ▷ ) = id`.

The example above is a rather mild manifestation of the new kind of interaction that can now happen between operators. To make the above work in our system, the *Shunting Yard Algorithm* must be modified to correctly handle those cases. Later in this chapter, we will give a detailed description of our version of the algorithm, for now, we can just say that ordinarily, the algorithm uses precedence to decide which operator should bind first. Because it does not take into account prefix or postfix operators it can not cover the situation demonstrated above. The algorithm needs to be extended in a few ways. The specific extension which makes it possible to handle the example above is called *the implicit rule* and it is defined as follows:

The implicit rule states that:

- infix operator followed by a prefix operator of arbitrary associativity has a lower binding power to the right relative to the prefix operator on that side, and
- infix operator preceded by a postfix operator of arbitrary associativity has a lower binding power to the left relative to the postfix operator on that side, and

- both prefix and postfix operator respectively, when followed, or preceded respectively, by an operator of the same fixity and arbitrary associativity has lower binding power than the operator it is followed, or preceded by, respectively.

Informally we can describe the rule as a specification for when one operator "waits for" another operator to "have a go". For example:

original expression	after disambiguation
`a + ▷ b`	`a + (▷ b)`
`a ! + b`	`(a !) + b`
`▷ ▷ a`	`▷ (▷ a)`
`a ! !`	`(a !) !`

with no regards for the specific associativity of each operator.

This allows us to write significantly more expressions without parentheses. However, it may also introduce confusion and errors. It is worth mentioning that such flexibility indeed comes at a cost, in this case, it may be a cost of clarity. Such a feature can have really surprising consequences as it turns out. We will cover those in the following short section.

2.3.2 Consequences of the Implicit Rule

In this short section, we will give one example of a direct consequence of the implicit rule. We will also give one indirect, implementation-dependent, but worth mentioning nonetheless.

We can get a rather unconventional behavior if we combine enough operators in a single expression. Here is what might happen:

```
prefixr 0 ▷
postfixr 0 ?

infixl 5 +
infixl 5 -

x + ▷ y ? - z
```

Since the addition and subtraction are the strongest operations and they are left-associative, one might expect the general shape of the expression to look something like:

```
(x + (▷ y ?)) - z
```

Despite it being a reasonable expectation, it is not the case. Instead, that expression is parsed as:

```
x + (▷ ((y ?) - z))
```

Which has the associativity completely flipped. The reason for it is quite subtle, it is caused by the interaction between the implicit rule and our earlier addition—associativity for unary operators. And because the prefix and postfix operators in the example are right-associative the result can be explained like this:

The `(▷)` must give way to `(?)` (because of their associativity). It also must give way to the `(-)`, because the subtraction is stronger than `(▷)`. Lastly, the fact that `(+)` is the last is trivial at this point—it is "stuck" behind the `(▷)` which has "lost" in all interactions.

> This behavior has symmetry to it. If the addition and subtraction are right-associative and both unary operators are left-associative, similar unusual interaction is going to play out.

We think that this interaction is rather a curious example of how flexibility can give rise to all sorts of unexpected behavior and may lead to a much less readable code. Unfortunately, that is not the final extent of the surprising interaction between operators. In our implementation, it also extends to ordinary function applications. To make the parsing algorithm simpler we have decided to pre-process the expression by making all function applications explicit by adding an explicit function application infix operator. Its name does not matter, but we will refer to it as `(@)`. This operator makes all function applications of arbitrary length significantly easier to process with our version of the Shunting Yard Algorithm. On the other hand, since at the time of the *SYA* running the function application is viewed as an ordinary infix operator it might be subject to the "unusual" semantics described above.

From a practical standpoint, it could be both an intentional behavior and an unwanted one. It all depends on whether we view an ordinary function application as just another (implicit) operator, which should be a subject to all the rules as any other operator would, or whether we see the function application as something special and above all operators. We do not attempt to answer that question. Our goal is not to assess those practical aspects of the design, instead, we tasked ourselves with designing a language with enough flexibility to observe those new challenges which might arise.

2.3.3 Shunting Yard Algorithm

In this section, we give two variants of our extension to the Shunting Yard Algorithm in pseudocode. The first version will be used to parse expression applications—expression sequences that correspond to one or multiple applications of various functions and operators. The second version will be used to parse applications within patterns. There is a slight difference in semantics when it comes to pattern applications and ordinary expression applications. We will explain it later in this section as well as the reason why a bit different approach to the *SYA* might be useful.

We also explain how the pre-processing to insert *explicit function application* operators is done. Finally, we describe how to disambiguate operators such as `(-)` which might be used both in prefix as unary and infix as binary. It will also be done as a pre-processing pass.

In terms of the interface, the *SYA* has a sequence of expressions on the input and a sequence of expressions on the output. While the former one is in the order as written in the program, where applications are just sequences of expressions and operators, the latter one orders the applications in the postfix notation. The next step is then to translate that postfix notation into a tree structure, where the AST for application is a binary node.

2.3.3.1 Parsing Expression Applications

Usually, the algorithm works with, besides the input, two data structures. It needs an *operator stack* and an *output queue*. Since their names are quite self-explanatory, we can advance to the actual pseudocode. It is presented in the listing 1

> Our version runs on the input which has already been processed by the parser, generated by *Happy*, therefore it is not concerned with actual *tokens*, instead it translates the input into two kinds of values—*non-operator* terms and *operators*. This transformation is fairly trivial and will not be discussed further.

The main difference to the original *SYA* as described in the *Algol 60 translation: An Algol 60 translator for the X1 and making a translator for Algol 60* [1] is that our version is not concerned

with parenthesized sub-expressions. At this point in the processing, they are already parsed away. If there is any parenthesized sub-expression within the one we will be processing, it will be represented as an application (in the form of the sequence of expressions) and it will need to be processed by the *SYA* recursively. Another difference is in the support for "unary" prefix and postfix operators. The original version does not support them, whereas our implementation does. As a consequence the part handling operators is more complicated than the original version, not by much, however, the algorithm is still quite straightforward to explain.

```
while there are tokens to be read on the input:
  read the token
  if the token is:
    - non-operator:
      put it on the output queue

    - prefix operator:
      push it on the operator stack

    - infix operator OP1:
      "operator loop":
      while there is operator OP2 on top of the operator stack:
        if the OP2:
          - is postfix:
              put OP2 on the output queue
          - has lower precedence than OP1
              BREAK "operator loop"
          - has higher precedence than OP1:
              put OP1 on the output queue
          - has the same precedence as OP1:
            if both are non-associative:
              report an error and terminate
            if their associativity is not equal:
              report an error and terminate
            if they both are LEFT associative:
              put the OP2 on the output queue
            if they both are RIGHT associative:
              BREAK "operator loop"
      push OP1 on the operator stack

    - postfix operator OP1:
      "operator loop":
      while there is operator OP2 on top of the operator stack:
        if the OP2:
          - is postfix:
              put OP2 on the output queue
          - has lower precedence than OP1:
              BREAK "operator loop"
          - has higher precedence than OP1:
              put OP2 on the output queue
          - has the same precedence as OP1:
            - if both are NON-associative:
                report an error and terminate
            - if their associativity is not equal:
                report an error and terminate
            - if both are LEFT associative:
                put OP2 on the output queue
            - if both are RIGHT associative:
                BREAK "operator loop"
      push OP1 on the operator stack

pop the entire operator stack onto the output queue
```

Code listing 1 Pseudocode for *SYA* on expressions. (Highlighted parts are the extension.)

2.3.3.2 Parsing Pattern Applications

The way pattern terms are disambiguated is quite similar to the algorithm above. There is a small difference, however. The language of patterns is significantly more sensitive to the presence of parentheses. Consider the following declaration:

```
data Triple a b c = T a b c
```

Any time the `T` will be used within the pattern, it must always be applied to exactly three pattern arguments with no additional parentheses. Here is an example of a legal use:

```
first :: Triple a b c → a
first (Triple x y z) = x
```

And here are a couple of illegal examples:

```
second :: Triple a b c → b
second ((Triple x y) z) = y

third :: Triple a b c → c
third ((Triple x) y z) = z
```

One way to make sure that this invariant is preserved by the algorithm is to introduce an additional structure—an *application queue*. The other part of the change will be the absence of the insertion of explicit application operators. Inserting those into patterns would make things more complicated during the process because they are not constructors, which makes them illegal operations within pattern terms. Instead of inserting those, we use the application queue accordingly. Any time we find a non-operator token we put it into the queue, and when we eventually find an operator token, we transform the content of the whole queue into a structure representing a pattern application and empty the queue. Together with the rest of the algorithm, it should have a consequence of not accepting pattern terms containing extra parentheses.

The listing 17 in the technical appendix gives the full algorithm, modified to parse patterns in the pseudocode.

The algorithm is much simpler than the one for parsing expressions, partly because there is only one class of operators in patterns—infix ones. However, this does not mean the language and the current implementation could not be extended in the future. The extension following the same principles which were introduced for expressions should be compatible with the current design.

2.3.3.3 Inserting Explicit Function Application Operator

Inserting the explicit function application infix operator `(@)` follows a very simple rule. We first assume all tokens are either non-operator values, or an operator. We then simply traverse the sequence of tokens from left to right, each time looking at two consecutive tokens at once. The simple rewrite rule would then look something like this:

```
first second  ⟶   first @ second
                      when first is either:
                        - a non-operator token
                        - a postfix operator token
                      and  second is NOT one of:
                        - infix operator
                        - postfix operator
              ⟶   first second
                      otherwise
```

Simply put, the `(@)` goes after things that belong to non-operators. That means non-operators themselves, but also postfix operators, because those bind to the left and therefore they are a *part of a non-operator value*. At the same time that non-operator term we have been talking about can not ever by applied to the operator—which would happen if the thing on the right of `second` was either infix or postfix operator. If it is a prefix operator, that we do not mind because prefixes bind to the right and as such they are *part of a non-operator value* on that side. Here are some examples to solidify the intuition:

```
-- precedence does not matter, so we omit that information

prefix _ ▷
infix _ +
postfix _ ?

...

f a + b c    -- with explicit application looks like:
f @ a + b @ c

f a ? ▷ b    -- with explicit application looks like:
f @ a ? @ ▷ b
```

2.3.3.4 Disambiguating the Subtraction Operator

A very similar idea is going to be used to identify operators which can be both prefix and infix and pick the right option. Once again, we traverse the sequence of tokens as in the previous step, only with a different rewrite rule:

```
first operator#AMBIGOUS   ⟶  first operator#PREFIX
                                  when the first is either:
                                    - infix operator
                                    - prefix operator
                          ⟶  first operator#INFIX
                                  when the first is either:
                                    - non-operator
                                    - postfix operator
```

In simple terms, the fixity of an operator which can be both prefix and infix, is decided based on the token *before* it. If the token is infix or prefix operator, as in `1 + - 2`, or `1 ▷ - 2`, it means that the ambiguous operator (here it is `(-)`) is understood as a prefix one.

Alternatively, if the token is either a non-operator or a postfix operator, as in `1 - 2`, or `5 ? 2`, it means that the ambiguous operator (it is still `(-)`) is understood as an infix one.

2.4 Type System

In this section, we will describe the implementation of the type system for the Glask language. We will start small, with a simple *DHM* implementation using a constraint gathering approach. We will then extend the implementation one, or a few, concepts at a time, eventually covering the whole specification of the type system.

Throughout the rest of this chapter, we will reference a few publications which we have used to guide our implementation of the type system. The first one of those is the *Typing Haskell in Haskell* [2]. It covers the implementation of the *DHM* together with qualified types and type classes. For the implementation of higher-rank polymorphism, we have followed the *Practical*

Type Inference for Arbitrary-rank Types [3]. To desugar type class-based overloading into a smaller core language we have followed *Implementing Type Classes* [5]. We also mention the rather instrumental *How to Make Ad Hoc Polymorphism Less Add-hoc* [4]. Despite it not being directly applicable to our language, it offers a good introduction to the issue.

2.4.1 Inference Informally

Let us start with an example of a Glask program:

```
twice f x = f (f x)

quad y = twice (\ a → a + a) y

(+) :: Int → Int → Int
```

> Suppose that the language currently does not support type classes and the `(+)` operator has a type shown in the example.

If we were to try to analyze this program and deduce types of functions `twice` and `quad` we could get quite far with just the knowledge we have so far. We could look at `twice` and see that it takes at least two arguments, so its type is going to look something like:

```
?f → ?x → ?(f (f x))
```

where the notation `?_` represents a type placeholder for the term _. From looking at the body of the `twice`—specifically the expression `f (f x)` we can further deduce that `f` is also going to be a function (from the top level application), its type is going to look something like `?a → ?b`.

> The rule for function application in the general shape `fn arg` says that the type of `fn` must be a function taking the type `?arg` as the input. The type of the result of that function is the type of the original application —`?(fn arg).` So `fn arg` produces a constraint `?fn = ?arg → ?(fn arg)`.

What we are effectively doing is collecting typing constraints from the shape of the program. At this point, it might make sense to continue in a bit different style. We will skip to the end of the process of collecting and show how each constraint came to be. We start with constraints for `twice`:

```
1)  ?twice  =  ?f → ?x → ?(f (f x))   from declaration of twice

2)  ?f      =  ?(f x) → ?(f (f x))    from the application f (f x)

3)  ?f      =  ?x → ?(f x)            from the application (f x)
```

> Note about notation: the symbol `=` denotes a kind of equality and is used to represent constraints.

Now we can keep going and add constraints generated for `quad`:

```
 4)  ?quad  =  ?y → ?(twice (\a → a + a) y)    from declaration of quad

 5)  ?(twice (\a → a + a))  =  ?y → ?(twice (\a → a + a) y)
                                                 from the application
                                                 (twice (\a → a + a)) y

 6)  ?twice  =  ?(\a → a + a) → ?(twice (\a → a + a))
                                                 from the application
                                                 twice (\a → a + a)

 7)  ?(\a → a + a)  =  ?a → ?(a + a)           lambda rule

 8)  ?((+) a)  =  ?a → ?(a + a)                application rule

 9)  ?(+)  =  ?a → ?((+) a)                    application rule

10)  ?(+)  =  Int → Int → Int                  from the type of (+)
```

At this point we can solve those constraints by going over them in any order we like and finding opportunities to simplify parts of the constraints by doing a substitution. Here is what we might end up doing:

```
11)  ?a → ?((+) a)  =  Int → Int → Int       from 9, 10

12)  ?a  =  Int                               from 11
```

Constraint number 12 is the first time we have narrowed down a type of some variable. From now on, when we are using a constraint containing `?a` we will first replace all occurrences of exactly `?a` with an **`Int`**. In this regard, we have obtained a sort of substitution, a mapping between `?a` and `Int`. In the rest of the process, we will consider the result of the unification to be a substitution. In some cases that substitution will just be empty.

Those partial substitutions also need to be combined as they are collected. Currently, we just remember them and use them when necessary, but there is a better more formal way, which will be discussed later in this chapter.

```
13)  ?((+) a)  =  Int → Int                          from 11

14)  ?((+) a)  =  Int → ?(a + a)                     from 8 using 12

15)  Int → Int  =  Int → ?(a + a)                    from 13 and 14

16)  ?(a + a)  =  Int                                from 15

17)  ?(\a → a + a)  =  Int → Int                     from 7 using 12, 16

18)  ?twice  =  (Int → Int) → ?(twice (\a → a + a))
                                                     from 6 using 17

19)  ?f  =  Int → Int                                from 1 and 18

20)  ?x → ?(f (f x))  =  ?(twice (\a → a + a))       from 1 and 18

21)  Int → Int  =  ?(f x) → ?(f (f x))               from 2 using 19
```

```
22)  ?(f x)  =  Int                                  from 21

23)  ?(f (f x))  =  Int                              from 22

24)  ?x → Int  =  ?(twice (\a → a + a))              from 20 using 23

25)  Int → Int  =  ?x → Int                          from 3 using 19, 22

26)  ?x  =  Int                                      from 25

27)  ?(twice (\a → a + a))  =  Int → Int             from 24 using 26

28)  ?twice  =  (Int → Int) → Int → Int              from 18 using 27

29)  Int → Int  =  ?y → ?(twice (\a → a + a) y)      from 5 using 27

28)  ?y  =  Int                                      from 29

29)  ?(twice (\a → a + a) y)  =  Int                 from 29

30)  ?quad  =  Int → Int                             from 4 using 28, 29
```

For the sake of clarity, we collect all the important constraints about variables in a single list:

```
?twice  =  (Int → Int) → Int → Int
?f  =  Int → Int
?x  =  Int

?quad  =  Int → Int
?y  =  Int
?a  =  Int
```

Now we can interpret the results. The type of `twice` is **`(Int → Int) → Int → Int`** and the type of `quad` is **`Int → Int`**. While the type for `quad` is correct and makes good sense, we can observe that the type for `twice` is needlessly specific. It is true, that in this specific program that type would work for `twice`, but it also makes sense to ask, why should `twice` be limited to work only on values of type **`Int`** when from the structure of the function nothing seems to imply that it will not work for values of other types, like **`Char`**. In other words, if we extend the original program like this:

```
twice f x = f (f x)

quad y = twice (\a → a + a) y

(+) :: Int → Int → Int

foo = twice (\b → b) 'z'
```

we will not find a solution for it. More specifically, when we collect all the constraints for such a program and attempt to solve them, we will discover a conflict. There will be a constraint stating that **`Int = Char`**, which is obviously not true. This error will occur despite that just from looking at the program we can observe that it is a perfectly valid one. The function `twice` does not make any concrete assumptions about the type of its second argument. The only requirement is that it is a type that is compatible with the first argument of `twice`—the function `f`.

The reason why it does not work in our current framework is because we have committed an accidental monomorphisation on the type of `twice`. In other words, we have forgotten to

generalize it as soon as it was possible. When we generalize a type we quantify all unsolved "type placeholders" (we can also call them type meta-variables) under a forall. This transforms the type of `twice` into a so-called *type scheme*. Generalizing a type of a function like `twice` makes it possible to call it with values of different types each time the call is done. That, of course, assumes that every invocation meets the requirement of the generalized type. This, however, introduces the issue of finding the right time to generalize. We resolve this issue by doing a *dependency analysis* on all declarations. That puts the declarations in such an order that they always depend only on things that are before them in that ordering. Looking at our example we would sort the declarations like this:

```
twice f x = f (f x)

(+) :: Int → Int → Int

quad y = twice (\a → a + a) y

foo = twice (\b → b) 'z'
```

The order shown above is not the only possible order, but it makes the most sense in regard to our explanation. With that being done, we can now again collect constraints for all of those, this time, however, we collect one declaration at a time and look for an opportunity to generalize the type of the current declaration. Here is how it might go:

```
1)  ?twice  =  ?f → ?x → ?(f (f x))          from declaration of twice

2)  ?f  =  ?(f x) → ?(f (f x))               from the outer
                                             application f (f x)

3)  ?f  =  ?x → ?(f x)                       from the inner
                                             application (f x)
```

We now attempt to solve those constraints and generalize the type of `twice` as much as possible.

```
4)  ?(f x) → ?(f (f x))  =  ?x → ?(f x)      from 2, 3

5)  ?(f x)  =  ?x                            from 4

6)  ?(f (f x))  =  ?x                        from 4 using 5

7)  ?f  =  ?x → ?x                           from 3 using 5

8)  ?twice  =  (?x → ?x) → ?x → ?x           from 1 using 7, 6
```

At this point, we can not find anything more concrete about those types. In other words, `?x` is unknown to us, because `twice` puts no requirements on it. Now is the time to generalize the type of `twice`. We assume `twice :: forall x . (x → x) → x → x`. This assumption looks similar to the type signature for `(+)`, indeed we will treat it the same way—later when we find an occurrence of `twice` in the program we will look up this assumption and use the type scheme it contains. There is, however, a difference between how we use assumptions that contain a type vs. the ones containing type schemes. For the latter one, we first must *instantiate* the type scheme before it can be used within a constraint. We do it by using the list of bound type variables (sequence of identifiers between the word `forall` and the period). Within the body of the type scheme (the part after the period), we replace all occurrences of each one of them with a fresh, unused type placeholder (denoted with the question mark).

Here is how it might look when we continue with the `quad`:

```
 1)  ?quad  =  ?y → ?(twice (\a → a + a) y)    from declaration of quad

 2)  ?(twice (\a → a + a))  =  ?y → ?(twice (\a → a + a) y)
                                                from the application
                                                (twice (\a → a + a)) y

 3)  ?twice  =  ?(\a → a + a) → ?(twice (\a → a + a))
                                                from the application
                                                twice (\a → a + a)

 4)  ?(\a → a + a)  =  ?a → ?(a + a)           the lambda rule

 5)  ?((+) a)  =  ?a → ?(a + a)                the application rule

 6)  ?(+)  =  ?a → ?((+) a)                    the application rule

 7)  ?(+)  =  Int → Int → Int                  from the type of (+)
```

> Notice the numbers starting from 1 again. That signals that to solve this batch of constraints we will not include any constraints from the previous steps.

Now we need to solve those constraints too:

```
 8)  ?a → ?((+) a)  =  Int → Int → Int        from 6, 7

 9)  ?a  =  Int                               from 8

10)  ?((+) a)  =  Int → Int                   from 8

11)  Int → Int  =  Int → ?(a + a)             from 5 using 9, 8

12)  ?(a + a)  =  Int                         from 11

13)  ?(\a → a + a)  =  Int → Int              from 4 using 9, 12

14)  (?e → ?e) → ?e → ?e  =  ?(twice (\a → a + a))
                                              from 3 by instantiating
                                              the type of twice

15)  (?e → ?e) → ?e → ?e  =  (Int → Int) → ?(twice (\a → a + a))
                                              from 14 using 13

16)  (?e → ?e)  =  (Int → Int)                from 15

17)  ?e  =  Int                               from 16

18)  ?(twice (\a → a + a))  =  Int → Int      from 15 using 17

19)  Int → Int  =  ?y → ?(twice (\a → a + a) y)
                                              from 2 using 18

20)  ?y  =  Int                               from 19
```

```
21)  ?(twice (\a → a + a) y)  =  Int            from 19

22)  ?quad  =  Int → Int                        from 1 using 20, 21
```

At this point, we have solved the type for `quad`. It does not offer any opportunity for generalization, so we just assume `quad :: Int → Int` and we move on to the next section. However, because that step would just repeat what has already been shown, we skip it and assume `foo :: Char`. All of that, however, can only work if we properly generalize and instantiate each time it is possible, effectively making the types of those functions as polymorphic as possible.

What we have just done is the core idea behind the DHM type inference. In the following sections we are going to give a little bit more formal description focusing on the most important parts of the implementation.

2.4.2 Damas-Hindley-Milner Type Inference

The type inference in our implementation of DHM works almost exactly like we have shown in the informal demonstration above. In the following sections, we will show how to represent *constraints* and *substitution* in our implementation as well as giving a description of the algorithms used for their manipulation.

2.4.2.1 Representation of Types

The major part of the implementation is the representation of types. It is designed to represent type expressions for the process of type inference. This means that it does not correspond one to one with the notion of Glask's types. To be more specific, in our **`Type`** data type, we have a variant for a type meta-variable, which is not a type. As has been said above—that is because our representation needs to be used during the process of inference, where the meta-variable is a central concept.

Another difference to the actual Glask types is the ability of the implementation representation to model types, that are not valid in Glask. For example tuples of higher-rank types. We will explain why those are not legal later—it has to do with the property of predicativity.

We will, from the very beginning, represent the types in such a way that the later addition of the higher-rank polymorphism will be as smooth as possible. For that specific reason, we will not introduce a notion of type schemes which is a very common way of representing universally quantified types in DHM implementations. Instead, we will start right away with the notion of a universally quantified type as an ordinary type.

The listing 2 shows our representation of types.

One thing worth explaining is the notion of a *rigid type variable* and a *flexible type meta-variable*—specifically the difference between those. While the former one represents an actual type, which happens to be a *variable* (bound by a `forall`), the latter one is more of a placeholder. Technically a meta-variable is not really a type in the theoretical point of view, but in the context of the implementation we can say that the **`T'Meta`** represents a thing that will eventually become a type, we just do not know what type yet. The inference engine can introduce a new type meta-variable whenever it needs to assign a type to some term, but the type is not known yet. On the other hand, type variables (sometimes called rigid variables) are used to represent polymorphism in the universal types. The distinction on the type level (between values of **`T'V`** and **`M'V`**) will be quite important, especially when we will need to represent a substitution between one of these to some type. In such cases, we definitely want to know that we only replace one or the other, but we can never replace both.

```
data Type = T'Var T'V
          | T'Meta M'V
          | T'Con T'C
          | T'Tuple [Type]
          | T'App Type Type
          | T'Forall [T'V] (Qualified Type)

data T'C  = T'C String Kind

data T'V  = T'V String Kind

data M'V  = M'V String Kind

data Kind = K'Star
          | K'Arr Kind Kind
          | K'Meta String
```

Code listing 2 Definition of Type and Kind representation with auxiliary data types

> The reason for the **`T'C`** (type constant) and **`T'V`** (type variable) containing a kind will be explained in the future sections. It was primarily influenced by the design choice in the *Typing Haskell in Haskell* [2].

2.4.2.2 Type Constraints

We have intentionally chosen our type constraints to be rather simple. They will be taking the shape of a simple, binary equivalence relation on types. We could understand it as stating that those two types it relates are "the same thing". We say it is a relation on types specifically, but the relation itself does not put any requirements on its arguments. It can be easily generalized. The reason to generalize it will be apparent later when we will be able to use it to construct all sorts of constraints, not only typing ones.

We model constraints like this:

```
data Constraint a = a `Unify` a
```

> We chose the name **`Unify`** for the constructor because the concept of *unification* is precisely the one we will utilize. It will be discussed in more detail in the following section.

2.4.2.3 Unification

The discipline of unification is a well known and studied topic [17], [18], [19]. It fits our use case rather nicely—it allows us to figure out whether two things, which might contain unspecified parts (not yet deduced types) can be the same thing. What is more—if they can be unified together we obtain the so-called *unifying substitution* which is a mapping between a notion of identifier and usually the thing we are trying to unify, in our case—a type. We will, once again, generalize our substitution data type so that we can use it to do many different things across the implementation. We will cover the substitution in more detail in the next section. But first, we will go over the process of unification. From a high-level point of view—any time we successfully unify two types we might obtain a non-empty substitution. If we apply that substitution to both of those types, we will make them exactly alike. From the implementation point of view, we might represent the unification on types as a function with the following type:

```
unify :: [Constraint Type] → Either Error (Subst M'V Type)
```

The actual type of the generalized version of unification differs quite a lot from the one above. The reason for it has to do with a lot of implementation details being involved in the exact representation. And therefore giving the actual type would be counter productive at this point. It is best to abstract away some of the details and present a type, which encapsulates the idea correctly while communicating the original intent.

When it comes to unification, type meta-variables can unify with any type, whereas a rigid type variable only unifies with another rigid type variable of the same name (besides a meta-variable). There is an important point to be made regarding the last constructor within the **`Type`** data definition. The **`T'Forall`** represents a universal type, at this point of our implementation, we will only ever see foralls to be used as a representation for fully generalized types inside a typing context. As a consequence, we will never see universal types taking place in the process of unification. That will be true for the entirety of the implementation and has to do with the type system having a property of predicativity. So aside from the last one, all of those type terms, or rather their representations, can participate in the unification. Those more complicated, containing types themselves—like **`T'Tuple`**, and **`T'App`**—will be unified recursively. The ones representing simple type terms can be split into two groups, depending on the shape of the result of their successful unification. The successful unification of two type constants or two type variables will only produce an empty substitution, the success itself is informative enough. In contrast to that, unifying the flexible type meta-variable, when it is successful, will produce a non-empty substitution, binding the name of the meta-variable to the other type. But even the unification of a type meta-variable can fail. For instance, if the so-called *occurs check* fails. A simple example of that might look like this:

```
fn = fn fn
```

The problem with this short program is that the type of `fn` is infinite. Here is the demonstration:

```
1)  ?fn  =  ?fn → ?(fn fn)                    from the application (fn fn)
```

When we attempt to solve this constraint, we end up in an infinite cycle as can be seen below:

```
2)  ?fn  =  (?fn → ?(fn fn)) → ?(fn fn)                  from 1 using 1
3)  ?fn  =  ((?fn → ?(fn fn)) → ?(fn fn)) → ?(fn fn)     from 2 using 1
...
```

Fortunately, the problem can be identified right at the beginning. The constraint number (1) has a general shape of `?a = ?a → ?b`. In other words, the constraint says that the type on the left is also part of the type on the right. But because the whole idea of the constraint is that the type on the left is supposed to be equal to the type on the right, this means that the type on the left is part of itself. That is exactly what makes the type infinitely recursive and what we will never allow. This is what the occurs check is for. It is supposed to identify situations where a type would be "composed of itself" and reject such types, failing the unification process and the whole type analysis.

As a side note—it is still completely fine to unify two meta-variables which are equal, in that case, the result is success and an empty substitution, no occurs checking ever takes place.

2.4.2.4 Generalization

In the informal introduction at the beginning of this section, we have introduced a process of generalization. In this short section, we will give a bit more detailed explanation of that operation and show how it might be represented in our implementation.

Here is a possible signature for the `generalize` function:

```
generalize :: [M'V] → Type → Type
```

The type signature demonstrates that `generalize` needs to be given a list of type meta-variables aside from the actual type to generalize. Those type meta-variables are exactly the ones that can be transformed into rigid type variables and closed under the `forall`. This needs to be done because in some cases we might need to generalize before we know all the concrete parts of the type. For example when we are generalizing the type of a local declaration defined inside a larger context. That could mean that eventually, some of those type meta-variables will become known and specific types, like **`Int`** or **`Char`**. In such cases we are not allowed to transform those into rigid type variables, instead, we need to ignore them for a time being and leave their possible generalization, or resolution, to the future part of the inference. Simply put, we can only generalize such type meta-variables which were introduced at the level of scope we are currently handling—if some type meta-variable was introduced for a declaration in the outer scope, we must not generalize it.

Type Schemes and Universal Types

One thing worth mentioning is that `generalize` returns a **`Type`**. Traditionally, when following the implementation of a simple *DHM* type inference we would be presented with a notion of a type scheme. Type schemes are very similar to our universal types in their structure and meaning.

The main difference is that usually in DHM the universal type is not a first-class, meaning it can not be a part of the user-definable expressible types. This leaves type schemes as something which is not really a type and only exists in the system in the layer above the notion of types. But because we plan to implement higher-rank types anyway, we decided to start with the notion of universal types being first-class from the very beginning. The same approach has been taken in the paper *Practical Type Inference for Arbitrary-rank Types* [3]. In this regard, we will not use the term type scheme, instead, we will call them universal types or foralls.

2.4.2.5 Substitution

At this point, we have covered most of the concepts which make up the type inference engine. In regards to the substitution, we have already explained how it is created and what are some of its uses. In this short section, we show how it will be represented in our implementation:

```
newtype Subst k v = Sub (Map k v)
```

> The **`Map`** is a standard associative collection from Haskell's `containers` library [20].

One thing worth noting is that substitutions can be composed together. Doing that is necessary when more than one constraint is to be solved. When composing two substitutions together we always need to make sure that there is no collision between those two substitutions. In other words, there must not be any such two mappings—one per each substitution—so that they are associating two different types with the same key.

2.4.2.6 Environment

The final piece of representation still missing is the infrastructure for keeping around the information about types of identifiers (variables and constants). The specific terminology may vary but we will use two terms in our explanation and implementation. We have a notion of a *type*

context, which is the specific collection associating the types to identifier names. We also have a notion of an *environment*, which is the whole data structure that is kept around during the type analysis. It contains the type context and a few more things. However, at this point of explanation, the only relevant part of the environment is the type context itself. Here is the representation of the type context in our implementation:

```
type Type'Env = Map String Type
```

It is a simple type synonym for a **`Map`** collection.

2.4.2.7 Inference for Programs

So far we have introduced the core design of our implementation. We now know the fundamental parts, and we should be able to write a simple `:type` command in some simple version of a REPL—that is, a command which takes an expression and infers its type. We have also explained how to run the inference on simple cases of top-level declaration. However, to analyze the whole program, with many declarations—some explicitly annotated, and others unannotated—we need to know how to combine the inference for both kinds of declarations, and also how to compose the inference for many sections of declarations (either global or local ones). This is the goal of this section. We will explain the main idea of the inference for the whole program (a sequence of declarations). We will cover the algorithm behind the type inference for both *explicitly annotated* declarations (also called *explicits*) and *unannotated declarations* (also called *implicits*). The last thing covered in this section will be a strategy of combining these two and running the type inference for the whole program.

Before we do that, however, we need to establish our main strategy: we will conduct a sort of "dependency" analysis on all bindings (at the current scope) to find out which binding is depending on which other binding. Consider the following example:

```
first :: Int → Int → Int
first x y = x

bar a = first a a
```

We can observe that for `bar` to work, it needs for `first` to be in scope. In that regard `bar` is dependent on `first`. On the other hand, `first` does not depend on anything. If we had to make a topological ordering of the declarations starting from those not depending on anything (or on language primitives and built-ins only), and continuing with those which's dependencies are already in the ordering, we could represent the current example as: `[ first, bar ]`. Before we generalize that into a rule, consider also this rather strange-looking example:

```
fn x y = gn y x

gn a b = fn b a
```

In the program above, we can not say that one of the functions comes before the other one. They both depend on each other. For cases like these, we extend the notion of the ordering—instead of having an ordered sequence of a single declaration, we order a section of those, in graph theory we call them *Strongly Connected Components*. To summarize, the topological ordering is a sequence of *SCC*s or just *components* from now on. Each component contains a set of bindings. The component can represent either a single binding or a set of mutually recursive bindings. The former has been demonstrated in the previous example, and the latter in the current one.

The only reason why we want to sort all the bindings topologically is to make sure we infer types as polymorphic as possible for each of our bindings. If we attempt to run the inference on the whole program as a single component, we might accidentally monomorphise some of the

bindings. The way that might happen was demonstrated during the informal introduction at the beginning of this section 2.4.

Inference for Implicitly Typed Declarations

We have already established that we want to split the whole program into a smaller components and find one of their correct orderings.

This of course means, that if there are more ways to topologically sort a given set of bindings, any of them will work. We will now explain the strategy behind the inference for implicits. It might be useful to assume, for a major part of this section, that the whole program consists of only unannotated bindings. We will later address it before we move on to the next section.

Simple Components

To run the inference on a component containing only a single, non-recursive binding we can assume that types of each of its dependencies are already present in the typing context. Let us reason about that assumption a little. If it is the first component in the sorted list of bindings it only depends on primitives, which are always present in the typing context. If it is from somewhere further down in the sequence, all its dependencies have already been processed, inferred and their most polymorphic types have been inserted into the typing context. And finally, since the component does not hold a "recursive" binding, the binding inside it can not depend on itself.

In reality, it would not be a problem for the binding to depend on itself. Our implementation design will be able to handle such cases just fine. It is, however, useful for the intuition we are building. For now, we assume that it does not depend on itself, therefore it is true that every binding (together with its type) it needs is already present in the typing context. Thanks to those assumptions, the type inference for such a form is trivial. Depending on the representation of declarations of variables and functions, we just run the inference on the body of the binding. (We will show our representation in the future section, together with an implementation of the type inference for implicits.) Once the inference returns a type and a set of type constraints, we can solve those constraints and apply the resulting substitution (if successful) to the type produced by the type inference. Then we can generalize all unsolved meta-variables in the type. Let us demonstrate on a small example:

```
fn a b = b + b
```

> We make an assumption that `(+)` is a language primitive, an operator with the type `Int → Int → Int`.

It is trivial to observe that `fn` will end up being in the first (and only) component, while the typing context contains at least the following assumption `(+) :: Int → Int → Int`. Therefore running the inference on an expression which is equivalent to `(\ a b → b + b)` will result in the type of `b` being inferred as `Int` and the type of `a` will be left unsolved. That directly translates to the fact that the type of `fn` should be `forall a . a → Int → Int`. Once we have generalized all unsolved type variables we make the type into a universally quantified type—in other words, we close it under a `forall`. That is exactly what we have shown in the previous paragraph. This generalized type is then put into the typing context and the following component gets analyzed.

Components with Mutually Recursive Bindings

The process of the type inference for a group of mutually recursive bindings has a very similar approach to what we have shown above. The only difference is a few more additional steps are

necessary for it to work.

Let us show another example:

```
fn a b = if False then fn a b else gn a b

gn x y = fn x y
```

We can simply observe, that for this program to work we need to know the type of `fn` when inferring the type of `gn`, but we also need to know the type of `gn` when inferring the type of `fn`. What is more, we also need to know the type of `fn` when inferring the type of `fn` itself. The last one sounds rather impossible to satisfy. Fortunately, it is not the case. We can use a "trick" to make the current example work rather well. The core idea of the "trick" is that before we run the inference for `fn` we give both `fn` itself, and `gn` too, *some* type so that the type inference can look those assumptions up in the typing context. To be a bit more specific—we assign each binding in the mutually-recursive component a fresh flexible type meta-variable and put those newly created assumptions into the typing context. Just because of that, when the type inference is invoked in the next step, it will be able to find types for all identifiers within each binding. In some cases, types of all the bindings within the component might not be as polymorphic as one would expect. Consider for example a variant of the program above:

```
fn a b = if False then fn a b else gn a b

gn x y = fn 'a' y -- we changed x for 'a'
```

It might not be obvious, but the type of `fn` will be `(forall a b . Char → b → a)`. That is just a consequence of inferring those two definitions together and not being able to generalize them independently. On the other hand, the following annotated version of the program works both in Haskell and in Glask:

```
fn :: a → b → c
fn a b = if False then fn a b else gn a b

gn :: x → y → z
gn x y = fn 'a' y
```

And the reason for it is those type annotations. Their presence divides the original component into two independent ones.

This just goes to illustrate that sometimes the type inference will infer less polymorphic types than one might expect.

To come back to the inference for a group of mutually-recursive bindings—once all the bindings are inferred and we obtain multiple sets of type constraints, we will solve them all together as usual. In case of success, the substitution can be applied to all type assumptions for the current bindings and we can finally generalize unsolved variables within those types. Same as in the simple version, we put the newly generalized assumptions into the typing context and move on to another component.

Inference for Explicitly Typed Declarations

The type analysis for explicitly typed bindings is much more simple. Despite the type being available for the analysis, it is also done by utilizing the type inference. The main difference is what happens after we infer the type of the binding. That is precisely the point where the inferred type must be compared to the one given in the annotation. The comparison needs to check that the type given, is consistent with the reality—with the type inferred by our algorithm.

However, before we explore that aspect of the type analysis for explicits, let us first take a look at the high-level strategy for analyzing explicits.

We can observe that there could be programs where implicit bindings depend upon explicit ones and vice versa. That might make us think that our components will need to contain both explicitly typed and implicitly typed bindings at the same time. That is not the case, however. If there are any implicitly typed bindings depending on some explicitly typed ones, all that is needed is to put all of those type signatures into a typing context before the inference starts. In other words, there is no need to include explicitly typed bindings in the process of the "dependency analysis" and consequent sorting of components. Instead, we put all the explicitly typed bindings into a single set of bindings and analyze each one of them separately. That takes place only *after* the inference for implicits is finished. At that point, specific dependencies of any of the explicits are not relevant. All of the type assumptions are already present in the typing context anyway.

The actual analysis done on each specific explicit is rather similar to the one done on a simple non-recursive component. The type inference is invoked on the body of the declaration. Once again, type constraints are solved and substitution is applied to the inferred type, which is then generalized. At this point, we have two universally quantified types or two type schemes. The final task is to check that the inferred one is at least as polymorphic as the "offered" one (the one from the annotation). To rephrase it—the one given in the annotation can be exactly as polymorphic as the one inferred, and it can also be less polymorphic. For a better intuition, see the following example:

```
id :: a → a                -- valid, it is AS POLYMORPHIC
id x = x                   -- as the one inferred

id'int :: Int → Int        -- valid, it is LESS POLYMORPHIC
id'int = id                -- then the one inferred

illegal :: a → b → b       -- invalid, it is MORE POLYMORPHIC
illegal x y = id'int y     -- then the one inferred
                           -- actual type is :: a → Int → Int
```

Normally, we would now explore this relation in more detail. It turns out it would immediately change in the next section on *qualified types* and then again in the section on *higher-rank polymorphism*. Instead, we wait until after the last extension to the type system and we will explain it then. So far we only give a name to that relation—a subsumption [21].

Inference for the Whole Program

Most of the explanation of the type inference for the whole program has already been covered in the previous section. Just to summarize it—we first conduct the dependency analysis on implicit bindings. Using that information we topologically sort them from the ones not depending on anything, besides maybe language primitives and explicit bindings, to those depending on some implicit bindings too. Then all type signatures from explicit bindings are collected and put into the typing context. This step assumes that internally type signatures are explicitly quantified by a forall even if the programmer did not write it that way. In the next step, one component at a time, all implicit bindings are solved, extending the typing context in the process. As the last step, explicit bindings are then checked. Each one can be checked in isolation, there is no interaction between them at all. In the end, we have a typing context with type schemes for all declarations (at the current level).

2.4.2.8 Constraint Gathering Approach vs. Online Inference

There tends to be some level of flexibility when it comes to when should the type constraints be solved. Some implementations might do it using an "on the fly" approach. That is—whenever

we obtain a new constraint, we immediately call `unify` and obtain a new substitution. But since this leads to the type inference and unification implementations being coupled together, it might not be the best design for our implementation. We prefer to decouple those two concepts and isolate each of them in their own part of the system. This should both raise the explanatory value of our implementation and allow us to refactor and change the code a lot—something which will be necessary since our implementation is based on work covered by various sources.

Our approach also has another interesting and positive aspect. The constraint gathering is fundamentally a syntax-directed algorithm. In other words, for each specific syntactic construct of the language, there is a straightforward way to turn it into a constraint during the analysis. This goes especially well with our design philosophy—it allows for an elegant algorithm with a high readability value.

The gathering of constraints is built on the idea that most of the time we can just keep collecting typing constraints up until some point. Then we attempt to solve them all at once and either obtain the resulting substitution or error out. There are some exceptions, for example, let expression—since it generalizes types of its bindings, we have to solve at least a subset of all constraints sooner, but that is not going to be an issue.

All in all, the constraint gathering approach will only require us to implement some way of accumulating typing constraints as they are discovered during the process of type inference. We will not offer any implementation, or type signature for that just yet. How we implement the accumulation is very much dependent on how the implementation manages the state. And since our implementation is written in Haskell, we will need to manage it explicitly. We will use some specific monads to do just that. However, talking about those kinds of details would be premature at this point, as it would introduce a whole lot of very specific properties which would require extensive explanation.

2.4.3 Custom Data Structures

In this section, we will describe steps we took to implement support for user-definable custom data structures. We will split the description into two parts, first part is going to be concerned with only monomorphic data types, whereas the second part is going to add support for parametrized data types, that will allow us to define custom "generic" data structures. We will also explain the design of the kind system which now becomes necessary with the presence of custom data types.

2.4.3.1 Monomorphic Custom Data

The representation we have chosen for our data structures looks as follows:

```
data Data         = Data { type'name    :: String
                         , constructors :: [Constr'Decl] }

data Constr'Decl  = Con'Decl String [Type]
                  | Con'Record'Decl String [(String, Type)]
```

> Regarding the "record syntax": We can see it purely as a syntactic extension. All uses are be desugared back to simple constructors, both at construction sites and at pattern matching sites. Therefore we will consider it just a syntactical concept and expect the implementation to desugar it before the type checking takes place.

The representation above only takes into account data declarations that are not parametrized by types. Later we will extend it a little, to support parametrized, higher kinded types.

Before we go any further, we present an example relating the data representation in our implementation to its source code counterpart. The following piece of code:

```
data Result = Invalid Message | Valid { response :: Data }
```

will be represented as:

```
Data { type'name = "Result"
     , constructors =
          [ Con'Decl "Invalid" [ T'C "Message" K'Star ]

          , Con'Record'Decl
              "Valid"
              [ ( "response", T'C "Data" K'Star) ]
          ]
     }
```

Before the type inference can begin, our compiler needs to collect all data declarations within the program and do some work. One part of that work is registering all the constructors into the typing context as just ordinary functions with known types. The advantage of the syntax we have adopted is that all types of constructor arguments are known beforehand. So to get the type of each constructor we simply compose the list of types inside the **`Constr'Decl`**. Assuming that at this point the desugaring has already been done, we only need to concern ourselves with the first constructor = **`Con'Decl`** which contains the collection of types directly. Having them, we can fold the sequence into a function type with the resulting type being a type constant with the name of the whole data type—`type'name` in our representation.

With all constructors collected in the typing context, we are free to run the type inference on the program. At this point, the need for the kind analysis already arises. Suppose that we get this source code to process:

```
data Bool = True | False

data Foo = First Bool Bar
```

The constructor **`First`** from the **`Foo`** type expects a value of the type **`Bar`** as its second argument. It is pretty clear, however, that there is no **`Bar`** type. This just makes the need for some sort of kind analysis obvious, even in the presence of just monomorphic data declarations. But we will not implement the kind analysis just yet. Instead, we will wait until we have extended the system by supporting higher kinded data types. That way we can avoid implementing a "sort of" kind analysis which would get thrown away anyway.

2.4.3.2 Polymorphic Custom Data

To extend the design introduced above, we start with a small change to the representation of our data declaration:

```
data Data = Data { type'name    :: T'C
                 , type'params  :: [T'V]
                 , constructors :: [Constr'Decl] }
```

The definition for **`Constr'Decl`** stays the same.

> The changes we have made are quite small. First, we change the type of `type'name`, the **`String`** will no longer do. The exact reason will be explained later in this section. Second, we add a field `type'params`, which represents all the type parameters of the data type declaration, in other words, those are all the type variables in between the name of the type and the `=` symbol.

Once again, we show an example of the source code and a corresponding representation. The following code:

```
data Either a b = Left a | Right b
```

will be represented as:

```
Data { type'name = T'C
                     "Either"
                     (K'Star `K'Arr` (K'Star `K'Arr` K'Star))

     , type'params = [ T'V "a" K'Star , T'V "b" K'Star ]

     , constructors = [ Con'Decl "Left" (T'V "a" K'Star)
                      , Con'Decl "Right" (T'V "b" K'Star)
                      ]
     }
```

In terms of the process which registers all the data constructors into the typing context, not much has changed. We need to consider the type parameters when constructing the type of each constructor from its arguments and from the defined type itself. The proper form of the result type of each constructor is a *type application* of the currently defined type constant (`type'name`) to all of its arguments (`type'params`).

With our change, however, we have introduced yet another possibility of writing not well-formed type terms. This time it is the possibility of the type application of a higher kinded type constructor being applied to an incorrect number of type arguments. The matter is not as simple as it might seem—at some places we allow those "partial applications" of type constructors and at some places we do not. That of course assumes that the type constructor being applied to some number of arguments is even parametrized by that amount of type parameters.

This and the issue described above are solved by a kind analysis. In our case, it is going to be implemented as a kind inference, because our language does not have a notion of kind annotations. In that regard, it is worth noting that extending the language with kind annotations and even some more kind system related extensions should be rather straightforward. Our design is not deliberately trying to prevent any of those future extensions.

2.4.3.3 Kind Inference

Our implementation of kind inference is going to be built around the same infrastructure as the type inference, only much simpler. Since the kind system only concerns itself with the shape of the things it analyzes and not the specific "values", constraints it operates on are purely shape-oriented. The high-level idea is that function types must obey the rule that each side of the **`(→)`** must have a kind **`Type`**, whereas the type application implies that the left part of the application must have a kind `?x → ?y` where the `?x` is the kind of the right part of the type application and `?y` is a valid kind.

In terms of kind constraints, the implementation is very simple. It is strictly subsumed by the high-level strategy for the type inference. We will therefore not spend much time describing the details of it. Instead, we will focus on aspects of the kind system, that might not be immediately

obvious. Those are details like kind polymorphism and how to run the kind analysis on all three types of declarations—data declarations, type class (and instance) declarations, and function (and constant) declarations (that might be accompanied by type annotations).

Kind Inference for Declarations

The kind analysis plays an important role in our implementation. Its purpose is to check that all of our types—be it those explicitly written, or those inferred—are well-formed. Despite the fact that the inference itself is rather simple, there is one aspect of the process, which might not be immediately obvious. It is the order in which the inference needs to analyze all the declarations in the program.

Without rank polymorphism, the kind inference would have a simple task. It is quite straightforward to show that in such language, data declarations can only ever depend on other data declarations. It is not possible to make a data type that would depend on a type class or any other construct. Therefore we would first do our dependency analysis on all data declarations and obtain a sequence of components of data declarations from that. We would then proceed with a kind analysis on type class declarations, which can depend on data declarations and other type class declarations. It might even be possible to process all of those within a single component. Finally, we would be able to check that all type annotations for functions and constants contain only well-formed types.

That is, however, not what we can do in our system. Our data declarations might feature constructors taking qualified types as arguments. This now makes it possible to have a data type, that depends on a type class declaration. Here is a simple example:

```
data Foo = F (forall x . Bar x ⇒ x → x)

class Bar a
```

This is exactly why we now need to perform the dependency analysis on a "mix" of data declarations and type class declarations, as they can depend on each other. However, it is still true, that type annotations for functions and constants can be isolated and analyzed only after the previous step has been done. It is even possible to defer the kind analysis for those until we encounter them during the process of type analysis. This is, in fact, the approach we have taken. That way, we do not need to walk the whole AST multiple times just to find all nested type annotations to check that they are well-formed.

It is also worth noting that when we talk about dependency analysis on type classes, it includes all of the type declarations within the corresponding type class. The reason for that is Haskell's specific approach to kind inference, which will be explained in the next section. Simply put, Haskell decides the kind of the type variable in the type class by taking into account all the type annotations too. And because of that, Glask does the same. Here is an example, for which the absence of the method annotation would cause a failure in the kind checker:

```
data Foo m a = A (forall x . Bar m ⇒ m a → m a)

class Bar m where
  bar :: m a → m a
```

Without that `bar :: m a → m a`, it is inferred that **`Bar`** expects a type of kind **`Type`**. That in turn makes this type `forall x . Bar m ⇒ m a → m a` not well-formed and causes a failure to kind check.

Kind Polymorphism

One interesting aspect of the kind system is "kind defaulting". Let us approach the explanation from a perspective of the dependency analysis on data declarations and type class declarations.

Our implementation of the type inference needed to split the entire sequence of bindings into smaller sections. Something similar happens in our kind system. The collection of all data declarations and class declarations also needs to be split into topologically sorted mutually-recursive sections. The reason is different this time around. The main motivation is not to make each data type, or a type class, as kind-polymorphic as possible. It is actually to make sure that the possible flexibility can be collapsed and defaulted into the kind **`Type`** in all cases where it might not matter. So the complete opposite.

This is a restriction put up by Haskell's earlier versions and lifted later with a pair of *GHC* extensions *PolyKinds* [22] and *KindSignatures* [23]. Nothing should prevent us from extending our implementation so that it supports kind polymorphism in the future. We have chosen not to do so for the sake of keeping the scope of the project small.

To come back to the kind defaulting, let us provide an example:

```
data Pol m a = P (m a)
```

By just looking at the example above, we can deduce that **`Pol`** seems to be kind-polymorphic on its arguments. But since we know that kind polymorphism is not covered by the current design of the language we must be able to decide which of the following uses are incorrect:

```
-- the simple case
data Simple = S (Pol Maybe Int)

-- a bit more complicated case
data Aux m = A (m Int Char)

data Complicated = S (Pol Aux Either)
```

Both of these uses would be valid under the *PolyKinds* extension, but since we do not have it implemented, we must decide which one is going to be correct. Haskell by the report decides to only allow the first one as a direct consequence of the kind defaulting that happens at the end of the inference for each section [24]. In simple words—after we infer the whole section of mutually-recursive data declarations, we find all unsolved kind meta-variables and extend the resulting substitution by defaulting all of them to kind **`Type`**. Only then can we move on to the next section of declarations.

2.4.3.4 Type Inference for Patterns

The type inference on pattern terms is very similar to its expression counterpart. The only difference is that patterns themselves are a binder construct. When a pattern variable is being inferred, instead of its type already being in the typing context, we need to assign it a fresh type variable and return it. Another consequence of patterns being binders is that their inference computes a set of typing assumptions—so that pattern variables within that pattern can be used in the corresponding context. Aside from that, it is very similar to the inference on the rest of the AST.

2.4.4 Type Classes and Qualified Types

The entire implementation of type classes in our type system comes from the paper *Typing Haskell in Haskell* by Mark P. Jones [2]. We will go into more detail about the specifics of the implementation in the next chapter, where we will discuss our way of adopting Jones's

implementation and possible deviations and our reasons for those. In this chapter, we will explain the high-level idea of the feature and give some context about how it will fit into our current design.

We start with the largest change to data structures used to represent the program. As we are approaching the territory of qualified types we must have a representation for a type context, qualified type, and a predicate.

2.4.4.1 Predicates

The **`Predicate`** data structure is used to represent a relation between a type and some type class. In some cases, like in type context, the type argument might be quite restricted, but that is not necessary to express in the shape of the data structure. Its definition is rather simple:

```
data Predicate = Is'In String Type
```

2.4.4.2 Qualified Types

It turns out it will be quite useful to define the **`Qualified`** type as a parametrized, higher kinded type, polymorphic on the thing it qualifies over. The benefit of that will become apparent later, for now, we give just the representation:

```
data Qualified t = [Predicate] :=> t
```

To relate the representation to the syntactical form, we show that the following term:

```
Show a ⇒ a → String
```

will be represented as:

```
let a       = T'V "a" K'Star
    context = [ Is'In "Show" (T'Var a) ]
    type'   = T'Var a `T'Arr` T'Con (T'C "String" K'Star)

in T'Forall [a] (context :=> type')
```

> Notice the explicit `forall` which has been omitted in the code example.

2.4.4.3 Inference For Qualified Types

So far we have given a possible representation of the data structures we will work with. We will now explain what changes to the inference the presence of qualified types causes.

We can conceptually split the impact of qualified types on the type inference into two parts; when and how are predicates introduced into the system, and how they are eliminated or otherwise processed.

The first part is going to happen when the inference deals with variables and literals. That makes it a problem well isolated to the “bottom” part of the inference process.

The second part, the processing and elimination happens at the boundaries of generalization. In other words, it happens when types of bindings need to be generalized. That is precisely when the predicates in the system need to come up and manifest themselves into the type of those bindings.

Creating Predicates During the Inference

As has been mentioned above, the only source of new predicates during the inference are so-called *overloaded values*. Those are values whose types have non-empty type contexts. For the most part, those are going to be methods, functions with types featuring non-empty contexts, and numeric literals.

When the type inference encounters an identifier, it needs to instantiate its type. When the identifier belongs to a method or a function with a qualified type, instantiating the type with a non-empty context means that the type context needs to get instantiated too. This creates a set of predicates quantifying over freshly introduced type meta-variables. This is very much consistent with what the instantiation has been doing to types so far.

When it comes to numeric literals, the process will be a little bit different. In Glask (as well as in Haskell), literals like `23` have a polymorphic type, like `23 :: forall a ⇒ Num a`, or `42.5 :: forall a ⇒ Fractional a`.

Processing Predicates at Generalization

This section will introduce a couple of new concepts necessary for making it possible to process predicates and put them into types of bindings that are being generalized.

Before we introduce those operations and explain what they do with the set of the predicates, we will first offer some intuition behind

When we run the type inference on a specific binding, we obtain a type and a collection of predicates both of which were computed from the right-hand side of that binding. Similar to the type inference without type classes and qualified types, when we handle bindings, we will at some point, need to generalize the type of that binding. Not all type meta-variables within the type of that binding are to be generalized, however. Consider the following example:

```
func num = let fn x = num in ...
```

The type of `fn` is `forall a . n` where `n` is a type of `num` and it happens to be a type meta-variable. It is obvious that when `fn` was being analyzed, the assumption `num :: n` has already been present in the type context. In other words, the type meta-variable `n` has not been introduced by the binding for `fn`. For that reason, when we generalize the type of `fn`, under no circumstances are we allowed to generalize over it. For the type of `fn` it means that it will contain a meta-variable and whenever it gets instantiated, this will introduce the meta-variable (`n`) to the context.

Something similar is true for predicates too. When we generalize the type of some binding, we need to figure out which predicates in the current context should be put into the type of that binding and which predicates should be "deferred" for the outer scope to handle.

This brings us directly to the first operation that we need to cover. It is called a `split`. Its task is to take some contextual information and a list of predicates and split those predicates into two lists. We will call the first one—*deferred* predicates and the second one—*retained* predicates.

Both of those names are quite self-explanatory. The predicates that were marked as deferred, will *not* participate during the generalization of the current binding. The ones marked as retained, on the other hand, will need to be processed a little bit more, but after that, they will become the type context for the to-be-generalized type.

The logic of splitting the set of predicates is quite simple. We inspect each predicate and find all meta-variables within it. If the predicate only contains meta-variable which are *not* to be generalized just yet, we mark that predicate as deferred. Otherwise, we consider that predicate to be retained.

There are some additional details regarding the processing of predicates that we have decided to omit for the sake of simplicity. Specifically, during the splitting, even before we partition the set of predicates into two parts, all of the predicates need to be so-called *reduced*. Reducing

predicates is a form of "simplification". In some ways, it makes the set of predicates smaller, for example by removing redundant and duplicate predicates from it, in some ways it might make it even larger, by decomposing predicates which allow it. (For example predicate `Eq [a]` can be reduced to `Eq a`, because the specific instance declaration `instance Eq a ⇒ Eq [a]` allows it.)

In any case, the predicate reduction is a very important step that helps to simplify some of the aspects of dealing with type classes and class-based overloading. However, we do not need to explore it in that much detail. It is very well covered in the paper and we would not get much from exploring it any further.

Before we go into handling implicits and explicits, however, there is one more thing we will cover. It is a matter of *ambiguous* types and type *defaulting*. The following short section will explain what it is and how it influences our system.

Ambiguity and Defaulting

The syntax of qualified types makes it possible to write what we call ambiguous types. Let us explain what it means for a type, to be ambiguous. According to the Haskell 98 Report, section 4.3.4 [25], we say that the type is ambiguous when it has a predicate in the context that contains a type variable that does not occur anywhere in the type besides the type context itself. Here is an example of one such type `Num a ⇒ Int → Int`.

One popular example of how this might happen is the function `stringInc`:

```
stringInc x = show (read x + 1)
```

> From now on `(+)` becomes a method of `Num` type class.

We can observe that the function is supposed to take a `String` and read a number from it. This number is then incremented by one and converted back to `String`. So the function takes a `String` and returns a `String`. The actual type of the function is, however, `(Read a, Num a) ⇒ String → String`, an ambiguous type.

In situations like this one, we can add inline type annotations to specify what the type of `read x` should be, like `stringInc x = show (read x + 1 :: Int)`. This gives enough information to the type analysis to know what `(+)` it should use and prevents those predicates from being propagated into the type of `stringInc`.

> The example above was taken directly from the *Typing Haskell in Haskell*.

However, the designers of Haskell decided to introduce a special infrastructure to automatically solve issues that involve numeric literals. It is called *defaulting* and its purpose is to automatically decide what numerical type should be used if it satisfies all the ambiguous predicates. We will not go into more depth here. Once again, it is a pretty interesting detail of Haskell's (and Glask's) type system, but it is a detail nonetheless.

Together with `reduce` the *defaulting* is used in the function `split`. It is there to make sure that all of the retained predicates that will become a part of the to-be-generalized type will never be ambiguous. A thing, that might easily happen without it, as the partitioning of predicates only cares about the predicate containing *at least one* meta-variable which is supposed to be generalized. It follows from that, that if the predicate contains another meta-variable that can not be generalized yet, inserting it into a type context would make that whole type ambiguous. For that exact reason, the defaulting takes place on retained predicates before they are returned from `split`.

Coming back to aspects of the inference that we will cover and explain, there is some difference in how predicates are dealt with, between implicits and explicits. The next few sections are going

to explore that difference as we conclude the description of this sub-topic and move on to another extension of our type system.

2.4.5 Implicitly Typed Declarations

We will start this section by acknowledging a specific limitation of Haskell's type system known as the *monomorphism restriction* [26]. It would be outside the scope of this thesis to attempt to explain the reason why it exists and what exactly it is. We have chosen to adopt Haskell's approach to type classes. That means we need to deal with this restriction too. With that being said, we now can explain the consequence of the restriction and how the implementation must handle it.

The restriction applies to declarations that have the shape `variable '=' Expression` and to the whole strongly connected component with at least one such declaration. We call those *restricted declarations*, and the important observation is that those are not functions from the syntactical point of view (even though the expression part might be a lambda abstraction). If the inference finds out that there is a restricted declaration, it restricts the whole component.

First, it means that types of all bindings in that section can not have type contexts. If the `split` returned a non-empty collection of retained predicates, those will be merged again with the deferred ones and returned from this level of inference.

That is not all, however. All the to-be-generalized meta-variables within the type of a restricted binding will be dropped from the to-be-generalized set. This makes good sense, because if we generalized them and there would still be some predicates featuring them, that would be an error, as we would not be able to correctly handle them later at a higher level.

Later, at the top level, we expect the implementation of inference for the whole program to do the defaulting. If that succeeds, we get rid of all remaining predicates and the type analysis completes, but if any predicate can not be eliminated, we get a type error reporting the reason for the failure.

A small illustration of what the monomorphism restriction does to our program might look like this:

```
number = 23
```

From now on all numeric literals are being overloaded, `23` has a type **`Num a ⇒ a`** instead of the previous **`Int`**. It is easy to see that the declaration of `number` is a restricted one. This means that the restriction must be applied to it. As a result, when we ask Glask's REPL for the type of `number` it will tell us it is **`Int`** (in Haskell that would be **`Integer`**, but Glask does not have an integral type for arbitrary large numbers [27].).

In case none of the declarations in the current component are restricted we can forget about the restriction and qualify types for each declaration with retained predicates while quantifying over flexible type meta-variables which are not free in the typing context (because they were introduced by the current declaration and not some outer-scope one). In this case, we only return the deferred portion of predicates for the outer scope to resolve.

2.4.6 Explicitly Typed Declarations

To support explicit type annotations containing qualified types we need to make sure that the type context given in the type annotation is no weaker than the one which would otherwise get inferred. That does not necessarily mean that they need to be equal modulo alpha renaming. Consider the following example:

```
class A a where
  fa :: a → a → a

class A b ⇒ B b where
  fb :: b → b → b

fun :: B t ⇒ t → t
fun bt = fa bt bt
```

If no type annotation is given, the one that is inferred is `A a ⇒ a → a`. That is certainly not the one being written in the example. Despite that, the one given in the example is also correct, because if some type is an instance of **B** it *must* be an instance of *A* too. In that sense using `B t` is a stronger requirement than using `A t`. This is only one way to show that two distinct type contexts may be correct for some declaration and we need to reflect that in our checking.

To generalize—when we said we need to check that the given type context is no weaker than the one inferred we meant we need to invoke some more complicated logic. We need to filter out all inferred predicates that are *entailed* by the context from the annotation. Predicates that are left are split into deferred and retained ones and we check that the retained ones are empty. If they were not empty it would mean that there are some predicates required at the current level, that are not covered by the type context in the annotation. That would make the context from the annotation weaker than it should be. In that situation we would raise an error, reporting those missing predicates.

See the following example for a better intuition:

```
class A a where
  fun :: a → a

foo :: Num a ⇒ a → b → b
foo a b = let bar :: Num ⇒ x → x → x
              bar x x' = fun x + fun x'
          in  b
```

Let us go over the example and explain what is going to happen at each declaration. We are going to approach it from a little bit different angle. We are going to skip right into the middle of the type analysis, this way it will be easier to focus on the important stuff. Let us start at the moment the local `bar` binding was inferred and now it is being checked whether the "offered" type annotation for `bar` gives a "strong enough" type context. The set of predicates inferred from the body of `bar` contains only two entries: `[Num x, A x]`. The former one is in the "offered" typing context, so it is released and we no longer need to care about it. What is now left is `[A x]` which we will attempt to split. We *retain* `[A x]` and an empty list to *defer*. Now we observe an erroneous state—the set of retained predicates is not empty, which means that the type given by the programmer to the `bar` is not complete in the sense of the typing context being weaker than the one inferred.

Let us now modify the example above a little bit and continue with the explanation focusing on a different scenario.

```
class A a where
  fun :: a → a

foo :: Num a ⇒ a → b → b
foo a b = let bar = id (fun a)  -- at the level of bar - ok
                                -- but foo is going to fail
                                -- because of it later
          in  b
```

Once again, skipping right to the level of `bar`'s declaration, this time we observe that the body of `bar` does produce a different set of predicates: `[A a]`. No filtering is going to take place, since `bar` is not annotated with an explicit type annotation, but the splitting of the set of predicates is going to happen nonetheless. During the split we will find out that all predicates within the list are being deferred—they only qualify over types involving meta-variables introduced by the outer scope. All the deferred predicates are returned from the inference. Fast-forwarding to the moment where the "context checking" for `foo` takes place—the same kind of issue as in the previous example is going to be discovered. **`Foo`**'s context is weaker than the inferred one because it lacks a predicate qualifying the type variable `a` with **`Num`**.

> A note about naming fresh inference meta-variables: for the sake of a simpler explanation we have assumed that the names of the freshly introduced meta-variables will be consistent with the names we used in the type annotation. We also skipped some steps and assumed that the future unification will unify type variables for `x` and `x'` so we can treat them as one and assume only one constraint. It is not a terrible simplification to do so, and it served the explanation rather well.

2.4.7 Type Checking Class Members

In this section, we will describe the main idea behind the type analysis of type class methods (or constants for that matter). In a way, they get treated very similarly to explicit declarations, but with some small differences. Since *Typing Haskell in Haskell* does not contain any explanation about *class members*, we needed to come up with a strategy ourselves. Throughout this section, we will attempt to explain why our specific strategy was chosen and how it works.

The same as explicits, type assumptions for methods need to be put in the typing context right at the beginning of the type inference. Consider the following example:

```
class Foo a where
  foo :: a → a

gn x y = foo x
```

For the inference to work, it is pretty obvious that at the time the `gn` declaration gets analyzed, `foo`'s type needs to be present in the typing context. That should serve as a clue to us, about *when* do all the class members need to be registered in the typing context. Here is the first difference between explicits and methods in regard to type analysis. Before we simply put all the type signatures from all the classes into the typing context, we must change each one of them a little. For each method we need to add a single predicate to their existing type context. That predicate adds the information that the corresponding type variable, from the type of the method, belongs to that specific type class. To be more concrete, in the example above, we need to add **`Foo`** `a` to the type context of the type of `foo`. Simply because the correct type of `foo` is **`Foo`** `a` ⇒ `a` → `a`. If the predicate **`Foo`** `a` was missing from it, the meaning would be completely different.

At this stage we do not need the actual implementations of those members, the same way as we did not need the rest of the explicit declaration—we just need their type signatures. Once all the implicit bindings are done being inferred, we can move on to check each one of the actual implementations of methods. Once again, following the same general strategy as with explicits. The only major difference in this part is the fact that for each method implementation we have to once again modify the type it gets checked against. Consider the same example extended by some instance declarations:

```
class Foo a where
  foo :: a → a

instance Foo Int where
  foo x = x

instance Foo Bool where
  foo b = b

gn x y = foo x
```

In this example, we have two actual method implementations to check. The one from the **`instance Foo Int`** and the one from the **`instance Foo Bool`**. Just to reiterate—the type of `foo` is **`Foo a ⇒ a → a`**. However, to actually type check the specific implementation, we specify the type by substituting the known type parameter for each occurrence of the *class type variable* and we safely remove the predicate **`Foo a`**. That leads to the first implementation being checked against the type **`Int → Int`** and the second against **`Bool → Bool`**.

The reason why we want to specialize the type for each specific implementation is to make the type, being checked, as specific and concrete as possible without loosing any polymorphism. In this case, it is safe to do so, because if we are, for example, defining an implementation of a `foo` in the **`instance Foo Int`**, it will be assumed that the argument is going to be an **`Int`**. So there is no reason to not specialize.

Alternative Approach to Method Checking

However, we could probably approach it from a bit different point—we might observe, that since our type checking of annotated declarations is based on inference anyway, we do not actually need to substitute the *class type variable* with the instance type. Instead, we let the inference infer the type and a set of predicates for the body of the specific method implementation. Then we check the type and the list of predicates and test if it is valid and satisfiable. However, there are at least two reasons why it might not be the best idea.

The first one is the fact, that it would make type checking of methods a little bit more involved and it is not apparent if it would work in all situations.

The second one is a little bit less apparent—for now, we use type inference even for the annotated declarations, but in the future, we are going to change our implementation significantly, utilizing a concept of bidirectional type analysis. We can then take advantage of the type annotation being as specific as possible. For example, it can lead to better error messages when type analysis of methods fails for a reason related to the instance type.

2.4.8 Higher-rank Polymorphism

In this section, we will cover another major part of the type system, higher-rank polymorphism. We will explain how its addition is going to change our current design and address some of the decisions we have made at the beginning with this extension in mind. We also give a short explanation of what higher-rank types (HRT) might be used in practice. This chapter and the whole extension of the type system with HRT are done according to the paper *Practical Type Inference for Arbitrary-rank Types* [3].

2.4.8.1 Higher-rank Types

In this short section, our goal is to address a notion of rank and to give some intuition about types with higher ranks. As has been defined in chapter 1.7.1, the notion of a rank of a type

is tied to the number of `forall`s in the type. Here is a couple of examples to help with the intuition:

```
min :: Int → Int → Int
```

The type of `min` has a rank equal to 0. Simply, there are no `forall`s at all.

```
apply :: forall a b . (a → b) → a → b
```

In this case, the rank of `apply` is 1. There is a single left-most, outer-most `forall` quantifier, and as such, the type is not considered higher-rank. Until this extension, these two shapes of types were the only ones we were allowed to use in our language. That is going to change. From now on, we will allow `forall`s to occur anywhere within a type. However, that is just a syntactical relaxation. Still, there will be a few places where using higher-rank types will not be allowed. We will discuss those later in this section.

Now we continue with the examples and show a type with a rank higher than 1:

```
takes'poly :: forall a . (forall b . b → b) → Int
takes'poly _ fn = let a = fn 'a'
                      b = fn 23
                      c = fn True
                  in a
```

We have also included a possible implementation for a better intuition. The main point of the example is that the type of the argument of `takes'poly` is a polymorphic function. This is exactly the reason why the rank of this type is 2.

Although our definition of a rank of a type is correct, we only want to measure a rank of such types that there are no `forall`s needlessly moved to the right-hand side of **→**. We call this form a *weak-prenex form* and in the following section we will explain what it is and how can it be reached.

Weak-Prenex Form and Weak Prenex Conversion

In the paper, the conversion to the weak-prenex form is done by "floating out" all the `forall`s on the right-hand side of **→**. According to its definition (page 23 of [3]), we can convert types like:

```
forall a . a → (forall b . b → a)
```

into a weak prenex form, which looks like this:

```
forall a b . a → b → a
```

While doing the conversion we have to be careful to not introduce naming inconsistencies. Consider for example the following type:

```
forall a . a → (forall a b . a → b → a) → a
```

It is still possible to "float out" the `forall` on the right, we just need to add some renaming to the process:

```
1)    forall a . a → (forall x y . x → y → x) → a

2)    forall a x y . a → (x → y → x) → a
```

We can, however, only "float out" `forall`s on the right-hand side of the →, never on the left-hand side. Because of that, the following conversion is illegal:

```
1)    forall a . (forall b . b → b) → a → a

2)    forall a b . (b → b) → a → a -- illegal
```

This covers the essence of the weak prenex conversion. When a type has been processed by a weak prenex conversion we say it is in a weak prenex-form. This form turns out to be quite important in the type system supporting HRT. It can serve as a sort of canonical form of the type. Later when we will need to figure out if some higher-rank type represents a correct type for some expression, we will usually first convert it into a weak prenex form and only then do the type analysis.

2.4.8.2 Practical Use of Higher-rank Types

The *Practical Type Inference for Arbitrary-rank Types* [3] cites a couple of use cases for higher-rank types from a few other sources. The overall idea is that higher-rank types are one of those concepts that might not come up often, but when it does, there is just no way around it. We do not want to repeat the examples given in the paper or point at use cases in the real world. For that reason, it seems that the best way to motivate the usefulness of higher-rank types in our system is to show how they can be directly useful to the language itself.

A prime example is the use of higher-rank types in data constructors. Those, in turn, can be used to implement type class-based overloading. The important part is that this is exactly one of the cases where to solve the issue, we either really need higher-rank types, or we must give up some guarantees, a type safety in this specific case.

Desugaring Class-based Overloading using Higher-rank Types

In this section, we will offer a specific motivation for higher-rank types in our language. We will show how having them in the type system will make supporting class-based overloading simple and type-safe.

Despite this topic being a subject of one of the future sections, we need to give at least a little introduction to how type classes are "compiled away" in our language.

Glask takes the same path as Haskell and desugars type classes into other constructs of the language. This allows class-based overloading to be resolved during compile time and not during run time.

This means that when a function has a qualified type, it needs to be passed a so-called dictionary, so that it gets access to the correct overloading of a specific method.

Let us give an example to make it a bit more clear:

```
double :: Num a ⇒ a → a
double x = x + x
```

We can see that function `double` uses `(+)` on its argument. In a language where there are no type classes—that includes our core language—we need to somehow supply the correct `(+)` to `double`.

At this point, we can take a look at the shape of the type of `double` and see that if we replace the ⇒ with →, it will make the type of `double` just a regular function type.

If the first argument to the function is the instance, or rather its run-time representation, there will be no class-based overloading and `double` can obtain the correct `(+)` from that instance data structure. That is exactly how the desugaring of class-based overloading works in Haskell and Glask.

We now need to figure out how to represent type classes and their instances in a smaller language, a subset of Glask without type classes and qualified types. That is the topic of the following section.

Representing Type Classes and Instances in a Smaller Language

We start with a motivational example showing not only how we can represent type classes and instances as ordinary values, but also demonstrating how higher-rank types will make it possible to still type check their desugared versions.

Suppose that we want to define a type class with a method, which has an additional type context within its type. A possible example of that might look something like this:

```
class Example a where
  method :: (Foo b, Bar c) ⇒ a → b → c → a
```

The next thing we need to do is to find a representation for the instance value at the runtime so that it can be passed around to functions. Even though the type class only has a single method, we will try to come up with a universal solution. We create a new data type, which corresponds to a container for a single value:

```
data Example a = Example'Dict a
```

The definition above represents a container that will accept a value of any type. Let us see what will happen when we define an actual implementation of the method for the arbitrarily selected type and attempt to construct an instance of **`Example`**. We choose **`Int`** to be the instance type in our experiment:

```
instance Example Int where
  method i b c = i
-- for the sake of simplicity
-- we ignore the Foo and Bar predicates and never utilize them
```

After the desugaring, the run-time value corresponding to the instance **`Example Int`**, it might look something like this:

```
data Example a = Example'Dict a

method'for'int :: (Foo b, Bar c) ⇒ Int → b → c → Int
method'for'int i b c = i

instance'for'int = Example'Dict method'int
```

> To type check the snippet above in *GHC* we need to enable *NoMonomorphismRestriction* extension (only because there is no instance of **`Foo`** and **`Bar`** defined, which prevents defaulting). That is, however, just a minor detail and we only mention it so that the example above can be reproduced. Otherwise, it does not have any interesting impact on what we are doing.

Now the issue manifests itself clearly. When checking what is the type of `instance'for'int` in *GHCi* we get this result:

```
instance'for'int :: (Foo b, Bar c) ⇒ Example (a → b → c → a)
```

As we can see, the whole type context sort of "floated out" from the type of the method and ended up qualifying over the whole container type. However, this is not acceptable. This would mean that when *constructing* any instance of **`Example`** we would need to provide those two instances too (because this basically makes `instance'int` a function taking two arguments according to our desugaring). But that is not, how it should be. Instead, we need to be able to construct an instance representation, containing a function, that *when called*, will require those two additional instances to be passed in. That is not possible in the version of the type system before higher-rank types.

For the sake of completeness, here is how this issue will get solved if the type system supports higher-rank types.

We start by defining a data type for **`Example`** like this:

```
data Example a
  = Example'Dict (forall b c . (Foo b, Bar c) ⇒ a → b → c → a)
```

If we now ask *GHCi* what is the type of `instance'for'int` we get **`Example Int`**. This means that we can construct that instance-value without needing to offer those two extra instances and they are only required when the specific `method` is called.

Now it should be pretty obvious that types like the one above can only be expressed in the type system with the support for higher-rank types. Here, the important takeaway is that without HRT we can not represent type classes and instances with methods of even simple types (no HRT). All it takes is a single method with a non-empty context.

2.4.8.3 Impact of Higher-rank Polymorphism on our System

The biggest change necessary for supporting higher-rank polymorphism in our type system is changing the implementation from being inference based to being bidirectional. That means, that we get to utilize all the type annotations written by the programmer to help the type analysis do its job. In essence it works like this—whenever we find a type annotation, be it the one accompanying a declaration or the inlined one, we make sure that as much of that type gets propagated into the type analysis of the corresponding term, as possible. Consider the following, simple example:

```
haskell 23 :: Int
```

It is no longer the case that the type of the expression is inferred—**`Num a ⇒ a`**—and then some checking happens to make sure that the given type (**`Int`** in this case) is a valid instantiation of that inferred type. Instead, we analyze `23` while expecting it to be **`Int`** from the very beginning.

This is in fact the only way that we are able to correctly type programs like the following:

```
fn :: forall b . (forall x y . x → y → x) → b → b
fn g b =  let x = g b 23
              y = g b 'a'
          in  if True then x else y
              -- demonstrating that they have the same type
```

This works on the principle that within the body of the `fn`, `g` is known to have type `forall x y . x → y → x` because of the annotation. Once that is taken care of, there is not a big difference between using `g` and say, using a global function `g'` with the same type.

Bidirectional Type Analysis

From the high-level overview, the central design of the bidirectional type system, at least when it comes to the infrastructure, is the ability to push the expected types into the type analysis

for expressions. In this section, we will discuss how we have chosen to implement it and how our approach differs from the one described in the *Practical Type Inference for Arbitrary-rank Types* [3].

The largest change to the infrastructure is that from now on, the type analysis will work in two modes. The first one—the *infer* mode is what we have seen so far. The second one is called *checking* mode—in this mode, some type is carried through the analysis and is expected to be used to guide the type analysis.

In the paper, they take advantage of mutable references and, in a sense, unify these two modes in many situations. When in checking mode, the actual type is carried around as is expected, but when in infer mode, there is a freshly introduced flexible type meta-variable serving as an expected type to the type analysis. This has a nice consequence that when the type analysis needs to return a universal type, it does not need to do so using unification (which would not work, because of the restriction of predicativity) and instead just writes into the mutable reference. In our implementation, we have chosen to not utilize mutable references in an attempt to make the whole implementation pure and as explicit as possible. For that reason, we had to come up with a different approach.

Our approach makes the difference between the two modes explicit and apparent. A type of our version of the function `tc'rho` (one of the fundamental functions from the paper) looks close to this:

```
tc'rho :: Expression → Expected Type → Actual Type
```

> The exact type of the `tc'rho` looks a bit different, we still have not covered some implementation details, so we present a simplified version of the type annotation. Though simplified, it contains all the important parts and represents the actual type fairly well.

The type annotation reveals the existence of two type constructors—**`Expected`** and **`Actual`**. These two containers are exactly what makes the notion of two modes explicit in our implementation. They are defined as follows:

```
data Expected t = Check t
                | Infer

data Actual t   = Checked
                | Inferred t
```

Their definitions communicate clearly that in checking mode we expect to have a type available to check things against, whereas in the infer mode there is *no type* at all. Dually, the result of the type analysis is presented with the **`Actual`** variant. When in checking mode, the result of the type analysis will be represented with just a **`Checked`** variant, no further information is necessary, on the other hand, when in infer mode, we need the type analysis to produce the inferred type. And since we can not use type constraints for that, because unification on polymorphic types is not allowed, we pass it inside the **`Inferred`** `t` type.

This is the largest change done to the implementation from the paper. Our version should, however, still be equivalent to the original design and should not change how the type system behaves.

In the following part of this section, we will cover a few fundamental operations described in the paper and relate them to the implementation.

2.4.8.4 Subsumption

The *subsumption* is a relation between two types (it technically relates two polymorphic types, but conversion from a type like *Int* to a polytype is trivial—`forall . Int`) which can be used to check the relation of two types in regards to their polymorphism. Here is an example:

```
forall a . a → a  ⩽  Int → Int
```

The meaning of the relation is understood as: the type on the left is *at least as polymorphic* as the type on the right. This of course means that the type on the left can be *exactly as polymorphic* as the one on the right. It is also worth mentioning that we will only ever use this relation on pair of types where one of those types is an instantiation of the other. Otherwise, it does not make sense to ask which type is more polymorphic. Here is an example of the case that does not make sense:

```
forall a . a → a  ⩽̸  Int → Char
```

Even though both types are function types, the one on the right is not an instantiation of the one on the left.

The subsumption relation is very useful when it comes to deciding whether two types are "compatible" in some sense. For example in our system without higher-rank types, we could use the subsumption to check whether a type annotation given by the programmer is in the subsumption relation with the generalized type that has been inferred for the corresponding expression (or declaration). In that case, the direction would be the following:

```
inferred  ⩽  offered
```

To give more concrete example, we reuse the one given earlier:

```
int'id :: Int → Int
int'id x = x
```

If the type annotation is not present and we run the inference on the `int'id` declaration, we get a generalized type `forall a . a → a` simply because the function does not make any assumptions or put any requirements on the argument, it just returns it. Still, the type given in the type annotation is correct. We are allowed to use the type annotations to restrict the interface of declarations and make their types less polymorphic. So the relation would look like this:

```
forall a . a → a  ⩽  Int → Int
```

And it would successfully relate those two types, serving as a check that the type annotation is correct with respect to the actual implementation.

In the paper, there are at least two notions of subsumption. That is because with the extension of unrestricted `forall`, there are more types that we would like to relate and some of them will not be related using the simple notion of subsumption. Our goal is not to re-explain every little concept from the resources. For that reason, we will not explain how those relations work and how the simpler one must be extended to cover a wider variety of inputs. Instead, we recommend looking them up in the paper (chapter 3.3 in [3]).

2.4.8.5 Skolemisation and Deep Skolemisation

The last detail we will explain in this section is sort of a dual of instantiation. It is a concept called *skolemisation*. In our system, it is necessary for working with type annotations.

Consider the following example:

```
'z' :: a
```

We want to use the type annotation during the type analysis of the expression on the left-hand side of `::`. However, we can not do the same thing that we do when inferring a type of a variable. That is, to instantiate it.

The reason for it simple. The above is actually `forall a . a` and instantiating such type, would yield a fresh type meta-variable. Those are known to unify with any type at all. However, that is not what we want to happen.

The example above contains a type error. It is not true that a character literal has a type `forall a . a`. So instead of using instantiation, we must use a different operation.

That operation is called skolemisation. It is fundamentally very similar to instantiation, but instead of producing fresh meta-variables, it produces new rigid variables. Those are known to only unify with themselves or with a flexible meta-variable. That is exactly the behavior that we want.

It is also worth noting that the skolemisation does "float out" foralls from within the type.

Limitations of Higher-rank Types and Predicativity

In this short section, we will address a previously mentioned idea, that higher-rank types are not allowed to occur at just any arbitrary place within other types. Instead, there are limitations dictated by the property of the system that is known as *predicativity*.

Here is an example of a program that will be rejected even by our extended type system:

```
data Wrapper a = Wrap a

value :: Wrapper ((forall a . a → a) → Int)
```

No matter what the body of `value` will look like, its type annotation contains an illegal type. This can be simply explained by the fact that our flexible type meta-variables only unify with monotypes. This in turn means that types with ranks higher than 1 can not be effectively used at places that would lead to unification happening on them. It should be rather obvious that when we take the higher-rank type from the example above and skolemise it, we will still be present with a type containing a `forall` within it. The reason is because it is on the left-hand side of the → and as such it can not be "floated out" of the type.

The reason why it is prohibited is exactly predicativity. In the paper [3] they go into a bit more detail regarding why having this happening in a type system might be a source of problems, but we do not plan to elaborate on it. Instead we only want to acknowledge it as an existing limitation and perhaps an opportunity for an additional extension.

2.4.9 Type Synonyms

In this short section, we will briefly explain how the concept of type synonyms works in Glask.

We can observe that in the representation of types, there is no trace of synonyms. That is because type synonyms are just a syntactical concept and they technically do not interact much with the type system.

The first step is to make sure that all type synonyms are so-called *fully applied* and there are no cycles in their definitions. That means that whenever there is a type synonym as a part of some type term, we need to check that it is being applied to the exact number of type arguments as it defines. The following is therefore illegal:

```
type Result t = Maybe t

data Wrapped'Int m = Wrap (m Int)

foo :: Wrapped'Int Result
```

Even though the kinds of the types in the example might be correct, this is prohibited. The reason for that is the way type synonyms are handled. They are all expanded even before any type analysis begins. For that reason, they must be fully applied and their definitions can not contain any cycles. Only then can we be sure that the expansion will be possible and it will always terminate.

It should be obvious that expanding all synonyms before the type and kind analysis might lead to some less informative error messages in cases where there are any type or kind errors involving those.

This can be solved on two levels. The first one would be to first collect kind constraints (as part of the kind analysis) before the synonyms are expanded. This would offer some more informative errors when the kinds of type synonyms are not consistent. The errors would contain the name of the type synonym, instead of the actual type term on the right-hand side of its definition.

The second level is to make the expansion of type synonyms completely "lazy". This is something that is done in *GHC* and can also contribute to the quality and informative value of the error messages. It is, however, arguably more elaborate than the former option.

2.4.10 Typed Holes

Our final extension of the type system will be the addition of typed holes. The main idea of typed holes is that if a programmer, for whatever reason, does not know what expression should be written at a specific place, they might use a *hole* instead. This way, the type analysis will attempt to figure out the type of the expression that the hole is representing. To implement them we will take as much advantage of the bidirectional type analysis as we can get. The fact, that we get to utilize the "offered" type, significantly raises the relevance of the message our type analyzer will generate upon finding a hole.

The key idea is that we do not raise an error upon a hole discovery. Instead, we record some necessary information to our state and proceed with the type analysis.

In inference mode, this means that we create a fresh meta-variable and return it as the type of the hole expression. In checking mode, we already have a type that needs to be returned.

In any case, we record the name of the hole (possibly with some additional location info about the hole) together with that type to the state and continue with the analysis.

Later, to be exact—at the generalization, we also apply the corresponding substitution to the types recorded for found holes. If there are any holes with fully specified types (types that do not contain meta-variables) we can report all of those as an error to the user, together with information like what type annotation binds that specific type of a corresponding typed hole.

Some other implementations also make it possible to report possibly relevant variables in the scope of the hole in an efficient way. The general idea of the approach is to interpret the type of the hole as a function type. This function takes as arguments all variables in the local scope [28]. This is an effective way to compose all the important information (types of local variables) in a single thing.

Later, during the step where the typed hole needs to be interpreted to the user, this function type is decomposed and names of the local variables are retrieved.

Our implementation will do with the simple idea to record important information to the state in the form as they are.

2.5 Desugaring of Type Class-based Overloading

One of the significant benefits of type classes is the fact that they can be statically analyzed and compiled away. In this section, we will focus exactly on that subject. We will describe the core idea behind the desugaring process and give some context related to our implementation. This section builds significantly on ideas from two sources, Phil Wadler's *How to Make Ad Hoc Polymorphism Less Add-hoc* [4] that introduces the concept of type classes, and Jones's *Implementing Type Classes* [5] that describes a way to implement them in a language like Haskell. In terms of implementation, we will rely heavily on the second one as our language is a little bit more complicated than the one described in the first one.

In this section, we are going to go over the fundamental idea of what is the dictionary-passing style, how to represent instances as language values, and how to represent type class declarations as other first-class concepts. All of this will be necessary to desugar programs featuring those concepts into a well-formed and typeable version without them. We will also explain the idea behind our specific implementation together with its benefits and drawbacks.

In terms of type class overloading, a *dictionary* refers to the value that represents a specific instance during the run time. Consider the following example:

```
class Num a where
  (+) :: a → a → a

double :: Num a ⇒ a → a
double n = n + n
```

In a language without type classes we could rewrite this piece of code like this:

```
data Num a = Num (a → a → a)

double :: Num a → a → a
double (Num plus) n = n `plus` n
```

Such transformations can always be done. This means that we can eliminate all uses of type class overloading with this explicit dictionary-passing style. In the rest of this section, we will talk about how this can be done and what mechanics can be used.

The example above gives a motivation for the process of desugaring type class-based overloading into first-class language constructs. We can observe that the idea is simply about passing around the dictionaries as extra function parameters. However, it is not always necessary to pass the dictionary if some function needs to use some method from the corresponding class. In some cases, it is perfectly possible to synthesize the dictionary at the place. Consider the following example demonstrating just that:

```
class Num a where
  (+) :: a → a → a

instance Num Int where
  -- there int#+ is a primitive, built-in addition on Ints
  + x y = x int#+ y

double :: Int → Int → Int
double a b = a + b
```

The code above might get translated to something like this:

```
data Num a = Num'Dict (a → a → a)

(+) :: Num a → (a → a → a)
(+) (Num'Dict plus) = plus

num'int = Num'Dict (\ x y → x int#+ y)

double a b = (+) num'int a b
```

Here the `(+)` has been transformed into a so-called *selection* function. It serves as a simple getter for the dictionary and returns the method, that was originally named `(+)`. Our original `double` has been changed too. Its body now contains a selection from the global dictionary. Because the type of `a` and `b` is **`Int`** we can select the correct instance/dictionary without needing to accept it as an extra parameter. Whenever we can do that, we will always synthesize the dictionaries instead of relying on the caller to pass them. This behavior is consistent with the way predicates are inferred—the type analysis will not construct a type like **`Num Int ⇒ Int → Int → Int`** for `double` even if we do not give the type annotation, the type context in such type would both break the syntactical rules and it would also be pointless.

In that regard, the types of functions will always reveal whether the dictionary should be accepted, or synthesized. When the type of `double` was

```
double :: Num a → a → a
```

it was clear that the dictionary needs to be passed, whereas when it was

```
double :: Int → Int → Int
```

since there is no predicate related to **`Num`**, the use of **`Num`**'s methods within its body can be done by synthesizing the right instance.

This of course requires the compiler to generate those global bindings—one per each instance, as well as to transform each type class into a data declaration.

2.5.1 Dictionaries of Qualified Instances

It is worth noting that generating a global instance binding for instances that feature additional contexts is a little bit more involved. Since the idea is, that the whole context needs to be offered to all instance methods, or rather their implementations, we need to consider this when such an instance is constructed. This means that dictionaries for qualified instances can not be fully constructed before some of the types involved are known and additional dictionaries are available. Here is an example taken from Haskell's library:

```
class Eq a where
  (==) :: a → a → Bool

instance Eq a ⇒ Eq [a] where
  (==) [] [] = True
  (==) (a : as) (b : bs) = a == b && as == bs
  (==) _ _ = False
```

If we focus on the second equation for `(==)` inside the instance declaration, we can see that each occurrence of `(==)`, on the right-hand side of the definition, refers to a different function. The first one compares elements of the list, whereas the second one compares whole lists. When this will be desugared and type class-based overloading is being compiled away, getting the corresponding dictionary for the second one is going to be trivial—we will look for an instance of

Eq for a list—coincidentally, that is the one we are currently defining, which makes it easy to see that it will be available. On the other hand, the first occurrence of `(==)` is going to be a little bit more complicated. Since the specific type is not known, we will need to obtain the dictionary as an extra parameter to the `(==)`. This makes it obvious that technically, all methods within a qualified instance share the context that the instance introduces. Some might not ever need it, but that is not important. The instance makes it clear, that it requires it for its methods. All of this means, that in case of qualified instances, we need to define a global *constructing function*, instead of just a variable containing the dictionary. This *constructing function* takes all the dictionaries the instance's context requires and provides them to each one of the methods. Only then such constructing function produces a fully specified dictionary for this specific instance.

We can take advantage of functions being curried in our language and just partially apply all of the methods to the instance context. This way all of the methods carry their instance contexts with no extra overhead.

Class Hierarchy

In the specification of the language, we have introduced a concept of superclasses and subclasses. We have shown that it is possible to define a new class as a sub-class of one or more super-classes.

However, we decided to not include the implementation of desugaring for type classes that are subclasses of other classes. The reason for it is simple. Even according to the paper [5] implementing those is rather involved. As it is not a fundamental feature of our language, we have decided to prioritize other, more interesting parts of the system instead.

We could have removed the sub-classing relation from the language altogether, to not make our implementation look incomplete. But that would significantly change the type analysis part of the implementation making it less interesting. Instead, we postpone the implementation of desugaring for subclasses to the future. That way we can focus on the important stuff and still present a complete adaptation of type systems from all of our sources.

2.5.2 Utilizing Higher-rank Polymorphism

Most of the ideas here have already been described in the section 2.4.8.2 about higher-rank types. For the sake of completeness, we go over those ideas again in a little bit more detail.

In the *Implementing Type Classes*, the way the dictionaries are constructed is using the ordinary *tuple type*. This has a direct consequence that even in the presence of higher-rank polymorphism, since tuples are not impredicative, we can not desugar into well-typed programs in some cases. Consider for example the following class declaration:

```
class Example a where
  function :: Num b ⇒ a → b → b
  constant :: Int
```

If we chose to represent dictionaries as tuples, it follows that the type of those dictionaries would look like this:

```
example'dict :: Num b ⇒ (a → b → b, Int)
```

We can see that the type context for `func` "floated out" of the inside of the tuple. We have already established that this is unacceptable and incorrect, but we are going to reiterate it here. Simply put, when we are putting together a dictionary for some type, we are required to provide a dictionary for **`Num b`** first. That is, however, not what the intention was. The intention was to have a dictionary containing a method, that is the one expecting that dictionary for **`Num b`** when it is called.

There is a nice way around this issue using custom data types. Since we can easily define data like:

```
data Example a = Example'Dict (forall b . Num b ⇒ a → b → b) Int
```

We can use this approach instead of utilizing plain tuples and get exactly what we need. That is, a dictionary, that does not require to be given a **`Num b`** at the construction. Instead it contains a method, that requires a **`Num b`** dictionary when called.

2.5.3 Placeholder Approach

The biggest change the desugaring of type class-based overloading leads to, is making the type analysis work in a bit different way. Instead of it taking the AST on the input and producing a collection of type assumptions, it now needs to also produce an AST on the output. Since the desugaring into the dictionary-passing approach is strongly type-directed, it makes sense to let the type analysis do the translation. The whole idea is quite in detail described in Jones's *Implementing Type Classes*. In this section, we will only explain some changes to the approach described there, together with how the approach will fit into our current implementation.

2.5.3.1 Placeholders

In the *Implementing Type Classes*, the idea of a *placeholder* is introduced to solve the problem of only having the important type information after it is needed. In that sense, placeholders are a temporary expression variant containing just enough information so that when all the involved types are solved, we can go back and eliminate all of them.

2.5.3.2 Using Placeholders

The motivation above shows that the way the type class-based overloading is desugared is composed of two parts—insertion of extra function parameters (to make passing the dictionaries into a function possible) and use of dictionaries (either those being passed as extra parameters or those synthesized at place). It should be obvious that the whole infrastructure is very much dependent on the specific types of values that interact with the overloading.

2.5.3.3 Placeholder Insertion

In this section, we will cover all three kinds of placeholders. We will explain when is each one of them used, and what they represent.

Method Invocation

```
class Example a where
  method :: a → a

instance Example Char where
  method c = c

expr = method 'a'
```

This piece of code should eventually desugar to something looking like this:

```
data Example a = Example'Dict (a → a)

example'char = Example'Dict (\c → c)

expr = method example'char 'a'
```

It illustrates that when we invoke a method, we need to make a selection from some dictionary. This in turn means that the placeholder needs to carry the information about the method (name will suffice), and the type associated with the type class. In the context of our example, it means that since the type of method is `Example a ⇒ a → a`, after the type of method gets instantiated—suppose it gets instantiated to `Example t1 ⇒ t1 → t1`—we need to store `t1` into the placeholder. So the first kind of placeholder might look something like `<method, t1>`. Here is the whole example as it should look after the placeholder is inserted but before it gets eliminated:

```
data Example a = Example'Dict (a → a)

example'char = Example'Dict (\c → c)

expr = <"method", t1, "Example"> 'a'
```

> For convenience, we have also added the name of the type class from which the method comes. Technically it is not necessary, but it might be a good idea to take this opportunity to make things a bit more explicit and obvious.

Qualified Function Invocation

```
class Example a where
  method :: a → a

instance Example Char where
  method c = c

ex'identity :: Example a ⇒ a → a
ex'identity x = x

expr = ex'identity 'a'
```

This case is different in that we are applying a function that expects to be given a dictionary as one of its arguments. The whole example should eventually desugar to something like this:

```
data Example a = Example'Dict (a → a)

example'char = Example'Dict (\c → c)

ex'identity :: Example a → a → a
ex'identity example'dict x = x

expr = ex'identity example'char 'a'
```

In that regard, the desugared expression looks very similar to the case with a method. The main difference lies in the placeholder. Here is what the intermediate step might look like:

```
data Example a = Example'Dict (a → a)

example'char = Example'Dict (\c → c)

ex'identity :: Example a → a → a
ex'identity example'dict x = x

expr = ex'identity <"Example", Char> 'a'
```

In the case of an overloaded function invocation, the placeholder looks different. It only contains the name of the type class and a type for which the instance needs to be found. Again, the strategy for getting that type is similar to the previous case—after the function type gets instantiated, its type context needs to be mapped into a sequence of placeholders. Since the type context is just a sequence of predicates in the shape of **`Is'In Class'Name Type`** the mapping is a trivial matter. When it comes to the order in which placeholders or rather dictionaries should be passed into an overloaded function, the only requirement is to be consistent across all call sites. We have chosen to go with the order given by the type context.

(Mutually) Recursive Function Invocation

```
recursive x = recursive (x + 1)
```

The last case is a little bit different at its core. Because of the way we analyze types of (mutually) recursive functions, it is not known, whether a specific (mutually) recursive function will be overloaded or not, before the whole section has been processed and generalized fully. For that reason, we need to consider an option that calling a function, that is just being analyzed, might require additional dictionaries. This is the responsibility of the third kind of a placeholder. The only piece of information we have about the (mutually) recursive function at the time of type analysis, is its name and the type (possibly a type meta-variable) associated with it. That is exactly the information that goes inside it. After the types get stabilized and the elimination step is invoked, this placeholder might be eliminated trivially—in cases when the (mutually) recursive function is not at all overloaded. Otherwise, it basically becomes the second kind of placeholder and will be eliminated as such. For the sake of completeness, here is what the intermediate step with the third kind of placeholder will look like:

```
recursive x = recursive <"recursive", ?recursive> (x + 1)
```

> We are using the notation introduced earlier where `?recursive` represents a flexible type meta-variable standing for a type of recursive.

2.5.3.4 Identifying Overloaded Identifiers

In the text above we have been focusing on how to introduce the correct types of placeholders and when. For the approach above to work, we also need to be able to tell whether some identifier represents an overloaded function, method or (mutually) recursive function.

When it comes to identifying methods, the matter should be trivial, at this point we have already collected all the important stuff from the program. Method names and their types are amongst them. To identify overloaded values we add a piece of code that, for each finished section, finds all bindings with qualified types and collects them. Lastly, we handle (mutually) recursive functions inside each section. More specifically, when each binding in the section is assigned a fresh flexible type meta-variable, we also register each one of them as being recursive. The

upside of our implementation approach is that since this registration goes into an immutable environment, when the section is done analyzing, those recursive associations get out of the environment automatically. If any of them is overloaded, the step above each section will once again register it, this time marking it as overloaded and not as recursive.

2.5.3.5 Placeholder Elimination

In the previous section, we have discussed how placeholders are introduced to represent temporary forms that can be later eliminated. In this short section, we will cover the idea behind their elimination.

Eliminating Placeholders for Overloaded Functions

We start by covering the elimination process for the second kind of placeholder—*overloaded function invocation* one. This elimination lies in finding the right dictionary for the overloaded function being invoked. That in turn, involves examining the type inside the placeholder (which is at this point as specified as currently possible) and deciding whether the type gives enough information to figure out which dictionary should be used. Considering these definitions:

```
int'id :: Int → Int
int'id i = i

class Num a

instance Num Int

overloaded :: Num a ⇒ a → a
```

Here is a small example:

```
foo n = let r = overloaded n
            m = int'id n
        in  twice
```

The function `int'id` is used here only so that the type of the argument `n` can be decided by unification to be **`Int`**. That is exactly what we need. Let us first take a look at the declaration above with placeholders inserted:

```
foo n = let r = overloaded <"Num", ?n> n
            m = int'id n
        in  r
```

From the type of `overloaded`, it is obvious that the function expects to be given a dictionary representing an implementation of **`Num`** for its polymorphic argument. That is exactly the meaning of the placeholder at the call site—it temporarily stands for that dictionary until it can be eliminated and replaced with the actual dictionary.

The moment when that happens is the generalization at the level of the let expression. Part of the generalization is solving all the constraints for the whole **`let`**. This in turn leads to the conclusion that `?n` is an **`Int`**. When the corresponding body of the `r` declaration is eliminated, it finds the placeholder and figures that what it represents is a dictionary for a type class named **`Num`** for a type **`Int`**. Since that specific instance is available in the global scope, it can get eliminated away.

There are cases, however, when the elimination does not resolve into using a globally available instance. Sometimes the type ends up being a polymorphic variable and as such, a dictionary

for it needs to be obtained from the caller of the function. A simple example would be a variant of the previous one:

```
foo n = let r = overloaded <"Num", ?n> n
        in  r
```

Without the call to `int'id`, there is no way of knowing what exact type should `n` have. Because of that, the type of `foo` needs to reflect that uncertainty. It does so by requiring, that whatever type it is, it needs to be an instance of a **`Num`** type class. Hence, the type of `foo` becomes `foo :: Num a ⇒ a → a`.

There are multiple other possibilities of what the corresponding type carried by the placeholder can become. Let us take a bit more systematic approach and explain the idea of how it is determined whether a placeholder can be eliminated and if so, what should it resolve.

If the type carried by the placeholder is:

- a rigid type variable, then we expect that the corresponding dictionary will be passed as a function parameter, or
- a flexible type meta-variable, then we can not eliminate it just yet, or
- a concrete type, then we expect the existence of a global instance declaration for the corresponding pair—type class and type.

It gets a little bit more interesting when the type happens to be a type application. From our knowledge of how instance declarations look and what is syntactically allowed, we can infer, that to identify which instance declaration corresponds to types like **`Maybe Int`** or **`Either`** `a b` (type applications in general), depends only on the type constructor.

This comes as a direct consequence of the limitation put on instance heads. Those say, that the type being declared to be an instance must be either a concrete type (like **`Int`** or **`Bool`**) or it can be a type application, in that case, there will be a *concrete type constructor* applied to a non-empty *sequence of distinct type variables*. Consider this example:

```
instance Example Int

instance Example Maybe

instance Maybe a
```

but never:

```
instance Example (Maybe Int)

instance Example (Either Char b)
```

This means that the type constructor alone is enough to decide what instance (or rather a dictionary) should be used.

To come back to the systematic approach, when we identify the type inside the placeholder to be a type application, we need to find the leftmost, outermost part of the type. This will be the type constructor. Then we need to do a case analysis on it, and find out if the type constructor is:

- a rigid type variable, then the corresponding dictionary will be passed as a function parameter, or
- a flexible type meta-variable, then we can not eliminate it just yet, or

- a concrete type constructor, then we need to find the corresponding, or instance declaration (not the dictionary) and observe the context of that instance.
  - If it is empty, we proceed by looking up the dictionary for that instance.
  - If it is not empty, we will need to find a dictionary for each predicate in the context before the actual dictionary can be fully constructed.

Most of these options can be handled in a pretty straightforward way. The only one, that is a bit more complicated is the last. There is a neat trick we can use to make it significantly more simple, however. If we assume that qualified instances are constructed by some function, which needs to be given all the additional dictionaries and then it returns said dictionary, we can observe that elimination of that original placeholder can be done by delegation. The only thing that is done, is the transformation of the original placeholder into an application. We apply the constructing function to the sequence of new placeholders, where each one of those placeholders corresponds to one predicate in the instance's context. Then we run the elimination on that application recursively. Because all those new placeholders represent a dictionary to be either found or synthesized, we rely on the part of the implementation which we have already covered.

Eliminating Placeholders for Methods

The trick we have used in the previous section can be utilized for method placeholders too. We observe that the method placeholder represents a selection of a specific member from a specific dictionary. We can eliminate it by creating an application of the method selector to a new placeholder, which is the kind of placeholder we have already handled in the previous section, and again running the elimination on that application.

Eliminating Placeholders for (Mutually) Recursive Functions

Again, we can use the trick from above here too. Before we do that, however, we need to look up the type of the (mutually) recursive variable, obtain the type context from its type, and map all predicates from it to dictionary placeholders. If the context is empty, we just have a normal function. We then construct an application of the original variable to the sequence of new placeholders which we have just created. Once again, we run elimination on that application to conclude.

2.5.3.6 Infrastructure for Placeholder Elimination

It is clear, that to eliminate placeholders, we need a sort of environment. We will use it to store names of variables that bind dictionaries or constructing function. In our implementation, we use the same environment for both globally available dictionaries and the ones that are passed to functions as extra parameters.

This simplifies the act of looking up the correct identifier. We do not need to inspect the instance type and decide whether its shape points us to a globally available instance or a locally passed one. Of course, this means that in the process of elimination, when we enter a function, that will obtain an extra dictionary parameter, we need to register that association. But because our implementation makes those registrations "scoped", we never need to explicitly delete them. This is a nice consequence of using immutable data structures.

When it comes to adding extra function dictionary parameters, we only need to consult the types of those functions. If the type contains a non-empty context, we need to add one extra parameter pear each predicate inside the context. Here is a demonstration of doing just that. We start with a program in the form as written by the programmer:

```
class Num a where
  (+) :: a → a → a

double :: Num a ⇒ a → a
double n = n + n
```

Because the type of `double` has a non-empty context, and because there is a class method used within its body, the whole definition will get transformed into this:

```
data Num a = (a → a → a)

(+) (Num plus) = plus

double :: Num a → a → a
double dict'num'a n = (+) dict'num'a n n
```

There is really no additional complexity in adding extra dictionary-parameters to functions.

Desugaring into Typed AST

One of the things our approach of using custom data structures for dictionary values allows us to do is to desugar into a well-typed program. The motivation for this could be additional processing of the program representation, static analysis, or optimizations. In any case, if we decide to do any additional transformations on the produced abstract representation of the program, we might benefit significantly from it being explicitly typed. For example, *GHC* does elaborate into an explicitly typed core and re-runs the type analysis (in checking mode) whenever it needs to make sure that any transformation done on it preserves the type correctness. We do not attempt to do this for a simple reason. This is the last transformation the representation will be put through. In the next step, it will get interpreted. And since any type information during the interpretation is ignored, we do not need to implement this extra step.

2.6 Interpretation

In this section, we will describe the design of our AST interpreter. We will start by lowering the desugared surface-level AST into a smaller, simpler representation which we will call *core*. Next, we will cover the basic design of our interpreter, details related to (mutual) recursive definitions and their evaluation as well as the infrastructure necessary for lazy/non-strict evaluation and *computation sharing*. We are also going to cover some interesting aspects of the implementation like pattern matching and representation of data structures at runtime.

2.6.1 Core

As implied at the beginning of this section, the main purpose of the core is to take a role of a significantly smaller and simpler target. It needs to cover all features of the surface language with a much smaller footprint. The smaller the core the more straightforward the evaluation will be. Here is the representation of the core as a Haskell data type:

```
data Core = Var String
          | Prim'Op String
          | Lit Literal
          | Abs String Core
          | App Core Core
          | Tuple [Core]
          | Let [Binding] Core
          | Case Core [Match]
          | Intro String [Core]

data Binding = Binding { name :: String, body :: Core }

data Match = Match { patterns :: [Pattern], rhs :: Core }
```

It is worth pointing out that the definition for both **`Binding`** and **`Match`** has been simplified too. At the same time, definitions for **`Literal`** and **`Pattern`** stayed the same, since they represent primitive building blocks in both representations.

2.6.2 Transforming Surface Language into Core

The translation of the surface representation into the core is fairly straightforward. In this short section, we will present a few examples together with an enumeration of things that need to also be translated into the core in a bit more indirect way.

Lambda Abstraction

When it comes to translating (lambda) functions with multiple pattern arguments, there might be a couple of possible ways to go about it. Because our evaluation model does not allow for strict evaluation inside the language (as opposed to Haskell and its `seq` function [29]), we do not need to concern ourselves with the question of whether the translated version behaves as lazily as possible alternatives. Despite that and it not being an explicit goal of this work, we will use a transformation similar to the one that is done by *GHC*.

Let us consider the following fragment of a code:

```
if'then True x = x
```

During the translation to the core, we must eliminate all function declarations into a simple case of a variable that is binding a lambda abstraction expression. The next step of the transformation of the code above might look like this:

```
if'then = \ True → \ x → x
```

We have also denoted lambda abstractions in their primitive shape. The next step is to get rid of the pattern matching on lambda arguments. This might lead to the following form:

```
if'then' = \ a → \ x → case a of
                            True → x
```

> We have taken the liberty of optimizing the second lambda, more specifically, we have not introduced a redundant case expression for the second lambda to make the example more readable. In this case, it is legal to do so, because the value bound to `x` is never inspected until it is returned. However, our actual implementation will not do that optimization during translation to the core.

Case Expression

Another interesting part of the transformation is dealing with case expressions. The ones in the core have a specific property—they must always cover all the possible cases. This is in contrast to the surface language, where a user can write code that does not handle all possible cases. The easiest way to ensure that all possible values are handled, is to add an extra branch with a wildcard pattern. The only responsibility of that extra branch is to raise an error reporting a non-exhaustive pattern matching.

Here is an example of just that:

```
case 1 of
    2 → True
    3 → True
    _ → False
```

It translates into:

```
case 1 of
    2 → True
    3 → True
    _ → False
    _ → error "non-exhaustive pattern matching"
```

The example above illustrates that insertion of that extra branch has no effect in cases when the original case expression covered all the cases already. In that situation, the only downside is redundancy. But this could easily be solved in the optimization step. Since our implementation is not concerned with the real-world impacts of such redundancies, we will not take the opportunity to optimize those cases and leave them as they are.

Data Construction

The final part of the core data type is meant to represent the construction of data structures. Since data constructors in our language behave just as normal functions, in the sense that they are curried by default, we need to make sure that the evaluation will reflect that correctly. For that reason, constructors are translated into normal functions with special constructs in their

bodies. Those special constructs represent the construction of a value of a specific data type and are only executed when all arguments have been supplied. Here is an example of that:

```
data Maybe a = Nothing | Just a
```

This code gets translated into the equivalent core representation:

```
Nothing = Intro "Nothing" a

Just = \ a → Intro "Just" [a]
```

Or equivalently as a Haskell data structure:

```
[ Binding { name = "Nothing"
          , body = Intro "Nothing" [] }

, Binding { name = "Just"
          , body = Abs "a" (Intro "Just") [Var "a"] } ]
```

> It is worth noting that at the core level, there is no distinction between *constructors* and *variables*. To be more precise, the core does not have a notion of constructors at all.

The string value inside the **`Intro`** constructor serves as a *tag* for pattern matching.

2.7 Interpretation

In this section, we will focus our attention on the design of our simple interpreter together with the representation of fundamental data structures used for evaluation. We will also cover important aspects of the evaluation in regards to general recursion and (mutually) recursive bindings.

2.7.1 Lazy Evaluation

The core idea of our notion of a *lazy evaluation* is that expressions are only evaluated when they are needed and once they are evaluated, the result of that evaluation gets shared. This means that when the same expression would need to be evaluated again, the interpreter will find the value it produced and use it instead. Because any expression in our language can always be replaced with the term representing the value it evaluates to, we can share with no issues.

Here is an example illustrating the described concept:

```
duplicate x = (x, x)

expr = duplicate (2 + 3)
```

In the example above, once the `expr` is forced to evaluate, for example, if it gets serialized, the original argument `(2 + 3)` must be evaluated only once. This behavior is achieved by not evaluating the argument portion of a function application until it is necessary. We will cover it in more depth, but before that we first present all important data structures used to implement our interpreter.

First, we present **`Value`** data type. It represents a notion of a run-time value. In other words, when we successfully evaluate an expression within some environment, we obtain a value of the following shape:

```
data Value = Literal Literal
           | Operator String
           | Closure String Core Environment
           | Data String [Promise]

type Environment = Map String Promise
```

It is worth noting that our **`Operator`** is used to represent primitive operations. Those are necessary if we want to do things like arithmetics on primitive (literal) values.

The **`Closure`** then represents an evaluated lambda abstraction—it carries around the *environment* that contains all bindings that were in the lexical scope for the orginal lambda abstraction.

The **`Data`** represents a value constructed by evaluating an **`Intro`** expression. Because data structures in our language are also lazy, it carries a list of promises, instead of values. This allows us, amongst other things, to create infinite data structures without any issues.

Finally, the environment can be represented simply as a dictionary associating names of variables with *promises*.

The promise represents a possibly unevaluated expression or a value. For it to get evaluated later it needs to keep around the environment containing all things in the scope of the original expression. Alternatively, when it is finally evaluated (we say the promise gets *forced*) it represents simply a value that can be readily used next time that promise gets forced.

Our **`Promise`** data type is then defined as follows:

```
newtype Promise = Promise Address

type Store = Map Address Promise'Content

type Promise'Content = Either (Core, Environment) Value

type Address = Int
```

From looking at the **`Promise`** definition, we can observe that it is just a thin wrapper around a `type synonym` for **`Int`**. The reason for that is because the concept of a promise in our design does not necessarily require a specific implementation—it boils down to a couple of implementation details (tied mainly to a **`Store`** data type) and might end up disappearing entirely if we chose to do so. We have, however, decided to include it in our representation for the sake of making the intention of the design clearer.

The place where most of the interesting aspects happen is the **`Store`** and a logic handling changes to it. In our implementation, the **`Store`** must be passed around in such a way that it allows modifications to be propagated, much in the same way as if we have chosen mutable imperative implementation of it. That is because our design behind *sharing* pretty much relies on the machine being able to do this kind of changes to the global state. In the following chapter we will explain how this can be done in the pure language like Haskell with the help of the right data structure.

This is the full extent of the data types defined for interpreting programs in our language. In the rest of this section, we will go over some high-level ideas of evaluation. In the final part of this section, we will also explain how our two-layer design makes it simple to support (mutually) recursive bindings.

For now, we will only present a type annotation associated with our `eval` function together with a textual description of its responsibilities. The specific implementation details together with parts of its source code will be presented in the next chapter.

```
eval :: Core → Environment → Machine'State Value
```

For now, we will not define a **`Machine'State`** type. We can think of it as a box representing the evaluation state. In our case, it means that it carries around the **`Store`**. Its definition might be a simple `type synonym` like **`type Machine'State = Store → (Store, Value)`** or something more complex with a nicer interface. In any case, it takes care of the **`Store`** being present during the evaluation and then returned when the evaluation is done, so that changes done to it get propagated throughout the whole evaluation process.

The most important aspect of the lazy evaluation in the context of our representation has to do with promises. When are they introduced into the system and when are they forced. There are just a couple of places where a core expression is transformed into a promise and the same goes for their elimination. Simply put—we introduce a new promise when some expression gets bound to a variable. That includes applying a lambda abstraction to its argument, **`let`** bindings, specific cases of pattern matching, and our special **`Intro`** construct being "applied" to a list of expressions. General idea is that we evaluate only the necessary parts and for the rest, we use promises. Function application is done like this: we evaluate the function expression and obtain closure. Each closure carries its environment. We take the environment from the resulting closure and extend it by binding the unevaluated expression from the right-hand side of the original function application to the name of the function parameter. Then we can evaluate the body of the closure within that extended environment and obtain a final value. Here is an example of that:

```
(\ i → i) (2 + 3)
```

Instead of evaluating the argument `(2 + 3)` right away, we first evaluate the lambda expression, obtaining **`Closure`** `"i"` **`(Var`** `"i")` **`Map.`**`empty`. We then evaluate the body of that closure within an environment containing the binding for `i` and a promise of `(2 + 3)`. Since promises are technically just pointers into the store in our implementation, this means we first need to construct a value of **`Promise'Content`** and put it into the stores. Such a value will then look like this:

```
Left ( App
           (App (Var "+") (Literal (Lit'Int 2)))
           (Literal (Lit'Int 3))
     , Map.empty)
```

Dealing with the special **`Intro`** construct is even more straightforward. Consider for example the following situation:

```
... -- definition of a Maybe data type

Just 42
```

That expression gets translated into a core equivalent to the following:

```
App
    (Abs "a" (Intro "Just" [Var "a"]))
    (Literal (Lit'Int 42))
```

This means that when the core expression **`Intro`** `"Just"` **`[Var`** `"a"]` gets evaluated, it will produce a value that contains a single promise, representing a value obtained by evaluating a variable named `a` from within the body of the lambda abstraction. Forcing that promise results in forcing yet another promise—the actual argument to the lambda abstraction. This sort of indirection might not be necessary in cases like these, but as was mentioned earlier—our implementation is not concerned with optimizing the code.

When it comes to promises and pattern matching, or rather the case expression, the evaluation also must introduce a new promise and use it accordingly. Here is an example that demonstrates the issue well:

```
case 2 + 3 of
    v → Nothing
```

The sub-expression `2 + 3` mentioned in the example should never be evaluated. The reason for it is that the variable pattern does not require the value it matches on to be in any particular shape. The wildcard pattern has the same property—it also does not require the evaluation of the matched-on expression. In the context of the implementation, this can be dealt with by first introducing a new promise for the sub-expression `2 + 3` and storing that expression together with the current environment in the store. The promise pointing at the unevaluated expression is then passed into the part of the implementation which does the pattern matching. That part needs to be implemented in such a way that it forces the value being matched against only if it is necessary—that is if the corresponding pattern is a literal pattern or a data constructor pattern.

The situation gets a little bit more involved when it comes to bindings inside a let expression. We will cover that case in a little bit more depth in the following section on *(Mutually) Recursive Bindings* because the logic is shared between global declarations and local ones inside a let expression. In both cases, we need to make sure that all the right-hand sides of those declarations will be within the scope of all declarations at the current level (all the others and itself too).

(Mutually) Recursive Bindings

One of the important aspects of our language is the possibility to define (mutually) recursive bindings. We have already managed to handle the type-analysis part of this aspect and now we must solve the issue of how to make all definitions inside a single section "see" all definitions inside that same section (including itself) during the evaluation too. To be more precise, we need to make sure that when a body of any binding from a section gets evaluated, it is evaluated in the context of all bindings from that section.

In this regard, our current two-layer design of an environment and a store makes this quite straightforward. We follow a simple recipe: first, we go over all bindings inside a section and build a new environment by adding an association of the name of each variable and a promise pointing to the slot inside the store that is not yet utilized. In a way, we reserve a slot in the store for each binding's right-hand side as we go over them. In the next step, we need to build a new store. We do it by extending the current one while going over the whole section again, this time focusing on the core expressions for the right-hand-sides, of those bindings. To put them in the store we need an environment, which we have built in the previous step. This way, each unevaluated core expression will have access to all bindings in the current section. We only need to make sure that we are putting the correct core expressions in the correct slots in the store. As a side note, we do not need to iterate over the collection of bindings twice, we can easily fuse those two operations, which is what we do in our implementation. However, it makes the explanation a little bit clearer.

Chapter 3
Implementation

We have already covered all fundamental aspects of the implementation in chapter 2. Throughout this chapter, we present some of the parts in more details. We will also be focusing on our contribution and places where we differed from the implementations given in the resources we followed. We will also point out parts that required changes between extensions to the language.

In those cases, we might not only show a final implementation but also an intermediate version of it, either taken from a specific resource or as it was designed originally by us. We will also attempt to point out some places where those multiple implementations have met in interesting ways and would introduce (nonobvious) mistakes if not handled correctly.

All of those aspects come as a result of implementing our language in phases, starting with a small language with simple type inference and extending it "one paper at a time". This practice might have introduced a lot of opportunities for mistakes, but is otherwise a sensible way of adopting ideas from multiple sources. That is especially true when those sources are compatible to such an extent and a single source, covering the entire issue, is not available.

3.1 Implementation of Lexical and Syntactic Analysis

Our implementation splits this responsibility into three parts. A lexer generated by *Alex*, a parser generated by *Happy*, and a hand-written translation phase that goes from the primitive version of the AST to the complete version that can be used for the type analysis.

In this short section, we will pay attention to the last part, the hand-written one. We will offer some insight into how the implementation is structured and showcase some of its parts.

The main difference between the simple and the final representation is that the former simplifies the *application* expression and represents it as just a sequence, not a tree structure. The reason for doing that is to support user-defined operators. The second difference is specific to *types*. Taking inspiration from Jones's *Typing Haskell in Haskell* [2], we have also chosen to include *kinds* inside the *types*. Unlike the paper, we do not assume all kinds to be solved before the type analysis, so making this work together with a kind inference posed a challenge to solve.

3.1.1 Simple Version of AST

In figure 3.1 we present only relevant parts of both structures (representation of expressions). The reason for it is the size of both definitions and the amount of repetition.

> We can observe that both definitions are very similar. The biggest difference is in the representation of the application.

```
data Term'Expr
  = Term'E'Id Term'Id
  | Term'E'Op Term'Id
  | Term'E'Lit Literal
  | Term'E'Abst Term'Pat Term'Expr
  | Term'E'App [Term'Expr]
  | Term'E'Tuple [Term'Expr]
  | Term'E'List [Term'Expr]
  | Term'E'If Term'Expr Term'Expr Term'Expr
  | Term'E'Let [Term'Decl] Term'Expr
  | Term'E'Ann Term'Expr ([Term'Pred], Term'Type)
  | Term'E'Case Term'Expr [(Term'Pat, Term'Expr)]
  | Term'E'Labeled'Constr String [(String, Term'Expr)]
  | Term'E'Labeled'Update Term'Expr [(String, Term'Expr)]
  | Term'E'Hole String

data Term'Id
  = Term'Id'Var String
  | Term'Id'Const String
```

(a) Simple version of the AST.

```
data Expression
  = Var String
  | Const String
  | Op String
  | Lit Literal
  | Abs Pattern Expression
  | App Expression Expression
  | Infix'App Expression Expression Expression
  | Tuple [Expression]
  | If Expression Expression Expression
  | Let [Declaration] Expression
  | Ann Expression Sigma'Type
  | Case Expression [Match]
  | Hole String
  | Placeholder Placeholder

data Placeholder
  = Dictionary String Type
  | Method String Type String
  | Recursive String Type
```

(b) Final version of the AST.

Figure 3.1 Definitions of simple and complete version of the AST.

As was mentioned in chapter 2.3, applications are handled by a modified version of *Shunting Yard Algorithm* designed specifically to fit our syntax. That and all other aspects of translation is implemented in a module called **`ToAST`**.

The basis of the implementation is a monad and a type class **`To'AST`**. They are defined in listing 3.

The **`Translate`** monad is composed from **`ReaderT`**, **`StateT`**, and **`Except`** from the `mtl` package [30]. Its task is to facilitate the infrastructure necessary for doing the translation. Each part of the stack handles a part of the responsibility.

The **`ReaderT`** provides an immutable *translation environment* containing information like *fixities* of operators, information about data type *constructors*, or their record *fields*, and so on. The specific implementation is not fundamentally important. We will make do with the **`Translate'Env`** being a collection of all the important information used during a translation process.

The **`StateT`** provides a mutable state that is needed for the increment counter. This counter is going to be used for generating unique identifier names during the translation process. Finally, the **`Except`** handles errors or rather their reporting.

One of the biggest responsibilities of the **`To'AST`** translation is generating *kind variables* and carefully inserting them into the final representation of **`Type`** data structures. This means that, at least at the beginning, our types will contain mostly kind meta-variables. However, we must make sure that all mentions of the same type variable or a type constant will always contain the same kind meta-variable.

```
type Translate a
  = ReaderT
      Translate'Env
      (StateT
        Translate'State
        (Except
          Semantic'Error))
      a

class To'AST a b where
  to'ast :: a → Translate b

translate :: To'AST a b ⇒
             a →
             Translate'State →
             Translate'Env →
             Either Semantic'Error (b, Translate'State)
```

Code listing 3 Definition of the **`Translate`** type alias.

Next, we present the details of the part of the implementation that takes care of translating application terms into the final representation. Before we can do that, however, we must show the implementation of our modified *SYA*. It is structured as a single type class accompanied by a couple of useful data structures. In listing 4 we define that type class and its auxiliary data structures.

As has been explained in the previous chapter 2.3.3, the algorithm technically operates on a sequence of tokens. It only differentiates between those tokens by considering each one of them to be either an *operator* or a non-operator or rather a *term*. The most interesting part of this type class is the method called `process`, which takes a list of tokens and disambiguates the sequence assuming it is an application term. The result represents the original input but processed and sorted in the postfix notation.

Once the result is provided, the rest of the **To'AST** algorithm can transform the postfix notation into the tree-shaped structure of the final AST.

We omit the actual implementation of the **SYA** type class here. It is full of specific implementation details. Besides, the complete description of the algorithm for both **Expression** and **Pattern** was already given in the form of a pseudocode earlier. Instead, in the listing 5 we present an implementation of the **To'AST**'s instance for **Expression**, specifically the part concerned with the application.

```
class SYA a where
  process             :: [Token a] → Translate [Token a]
  to'token            :: a → Translate (Token a)
  explicit'app        :: Token a
  make'app'explicit   :: [Token a] → [Token a]
  disambiguate'minus  :: [Token a] → [Token a]

data Token a = Operator { fixity        :: Fixity
                        , associativity :: Associativity
                        , precedence    :: Int
                        , term          :: a }
             | Term a

data Op = Op { fix    :: Fixity
             , assoc  :: Associativity
             , prec   :: Int }
```

Code listing 4 Definition of the **SYA** type class.

Aside from the expected processing, one thing worth explaining is the approach taken in building the tree structure. Because the reading operates on the immutable sequence, it works by passing around a function that "knows" how to read the rest of the input. That way, when for example an operator and two of its operands are read from the input, that function is then applied to the rest of the input and takes care of it. Still, it may be considered just a minor implementation detail. That just shows that despite the implementation being a little bit more involved and verbose, it is done in a very straightforward way.

When it comes to processing pattern applications, it is even simpler than handling expressions. The whole algorithm builds on the same foundation and uses the same design. The only difference might be that patterns are more sensitive to parenthesizing. Therefore the implementation of the *SYA* needs to reflect that as we have explained earlier.

The rest of the implementation of the **ToAST** module is fairly mechanical and presenting it would not necessarily give us more valuable insight.

```
to'ast (Term'E'App t'exprs) = do
    tokens ← mapM to'token t'exprs

    let explicit'fn'app = make'app'explicit tokens

        disambiguated   = disambiguate'minus explicit'fn'app

    in'postfix ← process disambiguated

    to'tree in'postfix

    where
      to'tree :: [Token Term'Expr] → Translate Expression
      to'tree tokens
        = read' tokens (\expr empty → return expr)


      read' (Term t : tokens) fn
        = do
          expr ← to'ast t
          fn expr tokens

      read' (Operator{ fixity = Infix, term = t } : tokens) fn
        = do
          operator ← to'ast t
          let make'app left right operator
            = case operator of
                Op "@" → App left right
                Op _ → Infix'App left operator right

          let compose right tokens
            = read' tokens
                    (\ left tokens
                        → fn (make'app left right operator)
                             tokens )

          read' tokens compose

      read' (Operator{ fixity = Prefix, term = t } : tokens) fn
        = do
          operator ← to'ast t
          read' tokens (fn . App)

      read' (Operator{ fixity = Postfix, term = t } : tokens) fn
        = do
          operator ← to'ast t
          read' tokens (fn . App)

      read' [] _
        = do
          throwError $
            SYA $ Missing'Operand "There is nothing to read."
```

Code listing 5 Part of the `to'ast` function that handles applications.

3.2 Implementation of Type Analysis

In this section, we present a basis for the implementation of our type system. We will cover the central data type, once again it will be a stack of monad transformers. We will also present a few implementation details related to combining kind and type analysis, refactorings needed between the adoption of multiple paper's designs, and implementation of a function handling type analysis for annotated bindings.

The listing 6 presents the type alias definition for the structure that will allow us to implement the type analysis.

```
type Analyze a b
  = ReaderT
      Analyze'Env
      (StateT
        (Analyze'State b)
        (Except
          Error))
      a
```

Code listing 6 Definition of the **`Analyze`** type alias.

Our type alias has two type parameters. The first one represents a type of value that we will want to return from functions operating inside the context of our **`Analyze`** monad. That could be a **`Type`** or a **`[Predicate]`** or so on. Simply—it will be exactly what that specific part of the type analysis should produce.

A type annotation for a simplified version of the function analyzing types of literals might look something like this:

```
infer'lit :: Literal → Type → Analyze ([Predicate], Type) Type
```

It illustrates nicely what is going on with the first type parameter to the **`Analyze`** and hints at what is going to be the point of the second one. Because the type analysis of simple literals might produce a list of predicates (numeric literals are technically polymorphic values) it not only produces a corresponding type but that list too.

The second type parameter to **`Analyze`** is for specifying what sort of analysis is the structure enabling. We have mentioned in the previous chapter that there are a few similarities between the type analysis and kind analysis when it comes to the infrastructure. So to avoid having to define most of it twice, we need to parametrize the **`Analyze`** type correctly to obtain the right one. Here are two auxiliary type synonyms used to make things a bit more concise:

```
type Type'Analyze a = Analyze a Type

type Kind'Analyze a = Analyze a Kind
```

That second type parameter is passed only to the type constructor named **`Analyze'State`**. Its definition might look something like this:

```
data Analyze'State a
  = Analyze'State  { counter      :: Counter
                   , constraints  :: [Constraint a] }
```

The other part of the **`Analyze'State`** data structure is **`Counter`**. It is a simple increment-only counter used for generating unique names during the analysis process.

This is certainly not all there is to the type analysis (or kind analysis for that matter). However, it is not our goal to present the source code of the implementation in its entirety. Instead, the purpose of these examples is in giving a context to the specific explanation or other more detailed examples.

3.2.1 Combining Kind Inference with Type Analysis

In chapter 2, we have introduced a strategy for implementing both the kind inference and the type analysis as two separate concepts. To properly implement our type checker, however, we need to combine them. Specifically, after the kind analysis is done running and it produces a collection of *kind assumptions*, we carry this collection around during the process of type analysis. It is going to be necessary whenever we need to run the type analysis for an explicitly annotated declaration (or for a method). It will be used during a kind analysis of each type annotation to ensure that the type within the annotation is well-formed.

The algorithm for the kind analysis itself is fairly uninteresting at this point. It is a direct re-application of the one used for the type analysis on the AST. The only difference is that it operates on types and aims to build a correct kind for the type in the annotation. What is worth noting, however, is that during the process of the kind analysis, we can make it so, that the type annotation that might still contain types featuring kind meta-variables, is fully specified. In other words, all kind meta-variables within the type are replaced with correct kinds. That is certainly a very simple task. Since all the kind information is available in the environment we can just use it to "patch" all the incomplete parts of the annotation.

In the future section, we will show full source code for the function handling the type analysis of explicitly typed declaration. It will be demonstrated that the part handling the kind inference for annotated bindings can be incorporated easily at the beginning of that function.

All of the above applies to method declarations too. The only difference is that in the case of methods we can do the kind analysis even before the type analysis ever starts. That is because there are no nested classes or instances in our language and we have all the method declarations collected. We took that approach in our implementation. There is no trace of kind inference when doing the type analysis for methods.

3.2.2 Implementing Bidirectional Type Analysis

This section covers mostly modifications that were necessary when adopting the design from the *Practical Type Inference for Arbitrary-rank Types* [3]. We show two functions designed to handle type class predicates, the function handling the type analysis for explicitly annotated bindings, and a small part of the function handling type analysis for expressions.

3.2.2.1 Splitting the List of Predicates

One of the interesting consequences of adopting the implementation from *Practical Type Inference for Arbitrary-rank Types* was the change in how we treat type variables and where can they occur.

The implementation in *Typing Haskell in Haskell* [2] only expects certain operations to be done on types featuring flexible type meta-variables and not rigid type variables. One of the examples is the function `split` from the *Typing Haskell in Haskell* presented in the listing 7.

> Note that in the original paper **`Tyvar`** was used to denote flexible meta-variables as well as the rigid ones. In this specific use case, however, the function `split` operates on two lists of flexible type meta-variables.

The implementation from the paper works well until we implement the higher-rank polymorphism together with bidirectional type analysis. After the extension, types from annotations are

```
split :: Monad m  ⇒
         ClassEnv →
         [Tyvar]  →
         [Tyvar]  →
         [Pred]   → m ([Pred], [Pred])
```

Code listing 7 Original type annotation for function `split`.

```
split :: Class'Env   →
         [T'V]       →
         [T'V]       →
         [Predicate] → Type'Check ([Predicate], [Predicate])
split cl'env fixed'vars gs preds = do
  preds' ← reduce cl'env preds

  let (deferred'ps, retained'ps) = partition part' preds'
      part' = all (`elem` fixed'vars) . free'vars

  retained'ps' ← defaulted'preds cl'env
                                 (fixed'vars ++ gs)
                                 retained'ps

  return (deferred'ps, retained'ps \\ retained'ps')
```

Code listing 8 Adaptation of the original version of `split` from the paper.

no longer treated as something exceptional. Instead, types from annotations will get skolemised and they might become part of the inferred types in many situations. This, in regards to predicates produced by the type analysis, means that some of those predicates might contain types featuring skolems/rigid type variables. For that specific reason, the function `split` as taken from the *Typing Haskell in Haskell* will not suffice.

Whenever it would get called as a part of the type analysis for explicits and methods we need to call a different function instead. In the following few paragraphs we will explain the change, and show both the original and new version of that function.

Our original adaptation of the function `split` is presented in the listing 8.

> We can observe that the function's type is equivalent to the one given in the paper. The only difference is that the monad is not abstracted over in our adaptation. In terms of the implementation, it is very much the same function as the one Jones gives in the paper.

The original `split` has a simple purpose. It takes a list of predicates and it is supposed to split it according to a simple rule. It *defers* those predicates, that contain only free meta-variables that are free in the current type context (those can not be generalized just yet). On the other hand, all the other predicates (those containing either only meta-variables ready to be generalized or a mix of the two groups) will be considered *retained*.

As has been mentioned, some of the retained ones might contain meta-variables not yet ready to be generalized. In those cases, they will be defaulted by the standard mechanism before they are returned from the `split` function.

The issue with the `split` function has two sources—getting all free meta-variables from the predicate only gives a collection of meta-variables. Any potential skolem variable within that predicate will be ignored. That, for example, means that if `split` gets to work with a predicate that only contains skolem variables, that collection is going to be empty.

```
split' :: Class'Env   →
          [M'V]       →
          [T'V']      →
          [M'V]       →
          [Predicate] → Type'Check ([Predicate], [Predicate])
split' cl'env fixed'vars skolems gs preds = do
  preds' ← reduce cl'env preds
  let (deferred'ps, retained'ps) = partition part' preds'

      part' pred
        = (&&) (all (`elem` fixed'vars) . free'vars $ pred)
               (not (any
                      (`elem` skolems)
                      (free'vars pred :: Set.Set T'V')))

  retained'ps' ← defaulted'preds cl'env
                                 (fixed'vars ++ gs)
                                 retained'ps

  return (deferred'ps, retained'ps \\ retained'ps')
```

Code listing 9 New `split'` function working for both type variables and meta-variables.

Now the second reason comes to play—Haskell's `all` function works in such a way that it returns **`True`** for any predicate and an empty collection. The result of this situation is the issue—`split` is not working correctly for predicates that contain skolem variables.

To fix that, we can define a modified version of that function called `split'` that can operate on predicates containing both—skolems and also meta-variables, presented in listing 9. The difference between the original `split` and our modified `split'` is fairly minor—it just needs to do the same check for skolem variables too. That means that the new function needs to accept one more argument, a list of skolems. Then all free skolems from predicates will be checked against that list. Just one list will be enough because the idea is that the list of skolems given to the new function will represent all skolems introduced at the current level. This means, that any skolem that is not in that list, comes from a different scope. And if there is a predicate that contains only those skolems, it can be safely deferred.

> As a side note, we can easily use this new version of the function at all original call sites. In case of implicits, we just use an empty list for skolems.

3.2.2.2 Looking for Instance Definition

Another interesting place where the difference between type variables and meta-variables comes to play is a function called `byInst`. It is a function used when we have a class predicate like **`Num Int`** and need to figure out if there is a corresponding instance (declared by the programmer) that fits that predicate. Since not all predicates will look as simple as the example in the previous sentence, we need to use pattern matching to verify that the instance head indeed matches the predicate.

First we present the function, as given by Jones in the paper:

```
byInst :: ClassEnv → Pred → Maybe [Pred]
byInst ce p@(IsIn i t) = msum [tryInst it | it ← insts ce i]
  where tryInst (ps :=> h) = do
          u ← matchPred h p
          Just (map (apply u) ps)
```

> As Jones writes in the paper, the `msum` function searches for the first defined value in the list.

Now for the issue with this specific implementation, the predicate given to the function might sometimes contain a type featuring type meta-variables, skolem variables, or a mix of both. However, the instances defined in the class environment will always feature only rigid type variables. And because this function uses *matching* and not *unification* some valid programs will be rejected. One way to fix that is to "instantiate" the rigid type variables in the instance head for each tried instance.

```
by'inst :: Class'Env → Predicate → Either Error [Predicate]
by'inst cl'env pred@(Is'In name type') = first'defined insts
  where
    insts = instances cl'env name

    try'inst :: Instance → Either Error [Predicate]
    try'inst inst@(preds :=> head) = do
      let free'in'inst = Set.toList (free'vars inst) :: [T'V']
      let mapping = map (\ tv'@(T'V' n k) → (tv', T'Meta $ Tau n k))
                        free'in'inst
      let subst = Sub $ Map.fromList mapping :: Subst T'V' Type
      let (preds' :=> head') = apply subst inst

      case match head' pred :: Either Error (Subst M'V Type) of
        Left err → Left err
        Right u → Right (map (apply u) preds')

    first'defined :: [Instance] → Either Error [Predicate]
    first'defined []
      = Left NoInstance

    first'defined (inst : insts) = do
      case try'inst inst of
        Left err → first'defined insts
        Right preds → Right preds
```

Code listing 10 Definition of our version of the `by'inst` function.

The listing 10 shows our implementation (including our version of the `msum` function called `first'defined`).

At this point, we have shown all interesting places where the type-level distinction between representations of the type variable and meta-variable necessitates some changes in the code.

All in all, making those two notions be a different thing on the type level has turned out to be beneficial for the most part. Especially when it comes to substitutions and obtaining a set of free variables (or meta-variables) from within "a thing related to types" it is quite useful to know that the substitution will not change anything it not ought to.

3.2.2.3 Type Analysis for Explicitly Annotated Bindings

At this point, we have covered all of the important details involved in the type analysis for explicitly typed declarations.

In this section, we will show a complete source code for one of the main (and most complex) functions handling the type analysis. More specifically, we are going to show how we have implemented the function named `ti'expl`.

```
ti'expl :: Explicit → Infer [Predicate]
ti'expl (Explicit scheme Bind'Group{ name = name
                                   , alternatives = alts })
  = do
    (qs :=> t) ← instantiate scheme
    (ps, cs't, cs'k) ← infer'matches alts t

    subst ← get'subst

    let qs' = apply subst qs
        t'  = apply subst t

    Infer'Env{ type'env = t'env, class'env = c'env } ← ask

    let fs  = Set.toList $ free'vars $ apply subst t'env
        gs  = Set.toList (free'vars t') \\ fs
        sc' = close'over (qs' :=> t')
        ps' = filter (not . entail c'env qs') (apply subst ps)

    (deferred'ps, retained'ps) ← split c'env fs skolems gs ps'

    if scheme /= sc'
    then throwError Signature'Too'General
    else  if not (null retained'ps)
          then throwError Context'Too'Weak
          else return (deferred'ps, cs't, cs'k)
```

Code listing 11 Definition of the original `ti'expl` function.

We did not have to write the function from scratch, it was presented in the *Typing Haskell in Haskell* [2] for a simpler version of the type system. However, our extensions to the type system necessitated a few changes to the original function. It now handles substantially more than the original version. It needs to take care of kind inference for the type annotation, it also needs to accommodate changes implied by the new bidirectional type system, and deal with type class-based overloading. As a result, the type analysis also returns a new AST. Our goal, however, was to keep the function as simple and as similar to the original version, given by Jones,

as possible. Simply because it makes assessing the changes much easier than dealing with two completely different functions. We first present our adaptation of the original function in listing 11.

> The original adaptation of the function works with the notion of a *type scheme* as opposed to just using types. It also does not produce a new **`Explicit`** with placeholders. But more importantly, it looks almost exactly like the one in the *Typing Haskell in Haskell*.

Our modified version after the adaptation can be seen in listing 12.

```
ti'expl :: Explicit → Type'Check (Explicit, [Predicate])
ti'expl (Explicit sigma bg@Bind'Group{ name = name
                                     , alternatives = alts })
  = do
    sigma' ← kind'specify sigma

    (skolems, qs, t) ← skolemise sigma'

    (alts', ps) ← check'matches alts t

    subst ← get'subst

    let qs' = apply subst qs
        t'  = apply subst t

    Infer'Env{ type'env = t'env, class'env = c'env } ← ask

    let fs  = Set.toList $ free'vars $ apply subst t'env
        gs  = Set.toList (free'vars t') \\ fs
        ps' = filter (not . entail c'env qs') (apply subst ps)

    (deferred'ps, retained'ps) ← split' c'env fs skolems gs ps'

    if not (null retained'ps)
    then do
      throwError Context'Too'Weak
    else do
      return (Explicit (T'Forall skolems (qs :=> t))
                       bg{ name = name, alternatives = alts' }
              , deferred'ps)
```

Code listing 12 Definition of the `ti'expl` function.

There are two kinds of differences—new functionality was added, but some of the old responsibility was also "refactored away".

More specifically, the new version no longer handles situations where the signature is *too general*. That is, the type given in the annotation is more polymorphic than the actual type of the term. This responsibility is now handled by the insides of the bidirectional type system. To point in the right direction—a skolem variable only unifies with meta-variables or the same skolem variable.

The change relaxes some of the requirements on the **`Type`** in our implementation. We do not need to implement the **`Eq`** type class for our notion of types—something that the implementation from the paper required. We also obtain a little bit better error messages in cases where the type in the annotation is more polymorphic than it should be.

One interesting detail, that should not go unnoticed is related to the last line in the new version of our function. It should be obvious why we are building the **`Explicit`** again and returning it from the function. However, what might be not so obvious is why we are using the "skolemised version" of the type from the annotation.

The reason is quite simple. During the type analysis, it is the skolemised version of the type that is being propagated down the process. This leads to potential placeholders containing those specific skolem variables. And because `ti'expl` does not do the elimination itself, but delegates that responsibility further up, the type in the annotation being present during the process must be composed of those skolem variables, otherwise the inconsistency would prevent those placeholders from being eliminated.

3.2.2.4 Type Analysis for Expressions

To also give at least a limited insight into the implementation of the type analysis for expressions, we show a piece of code that handles type analysis for variables in listing 13.

Because of the placeholder approach, this part of the whole function is by far the most involved.

To give a bit more context to the implementation, we present a few other implementation details related to placeholders.

During the type analysis, we need to carry around information about all (potentially) overloaded variables. For that, just a list of pairs will do. Our *inference environment* carries a data structure with the type `[(String, Overloaded)]`. **`Overloaded`** is then defined as follows:

```
data Overloaded = Overloaded
                | Method String
                | Recursive
```

Each variant of the **`Overloaded`** data structure corresponds to one kind of placeholder. If the analyzed variable can be found in the collection, it will directly lead to that specific kind of placeholder being inserted.

There is nothing surprising going on in the source code. Even if the variable is overloaded, the logic behind the construction of the corresponding placeholder is fairly straightforward.

The only part where the implementation is a little bit more involved is in the *method* case. Here we need to traverse the list of predicates (produced by the instantiation) and search for a predicate that represents an information that this method belongs to the type class whose name is stored in the `cl'name` variable. This kind of predicate will always be present in the type context of the *method* (it is inserted by the implementation itself), but since the `find` returns a value of the type **`Maybe`** `t`, we have to handle the impossible case too.

From the implementation standpoint, it would be ideal to be able to omit most of this. To do that, however, we would probably need `inst'sigma` to give us some additional information. That would likely lead to substantial complexity or code duplication, so for the lack of a better, obvious solution, we just handle the impossible case explicitly for now.

```
ti'expr :: Expression    →
           Expected Type →
           Type'Check (Expression, [Predicate], Actual Type)
ti'expr (Var var'name) expected = do
  sigma ← lookup't'env var'name expected
  (ps, actual) ← inst'sigma sigma expected

  -- implementation detail
  let ty = case expected of
             Check t → t
             Infer →
               case actual of
                 Checked →
                   throwError $ Internal "inconsistent modes"

                 Inferred t → t

  m ← lookup'in'overloaded var'name
  expr' ← case m of
            Nothing →
              -- not overloaded
              return $ Var var'name

            Just Overloaded → do
              let placeholders  = map pred'to'placeholder ps
              return $ foldl App (Var var'name) placeholders

            Just (Method cl'name) → do
              case find (\ (Is'In c'n _) → c'n == cl'name) ps of
                Nothing → do
                  throwError $ Internal "malformed type context"

                Just (Is'In c'name ty) →
                  return $ Placeholder
                         $ Placeholder.Method var'name ty cl'name

            Just Recursive →
              return $ Placeholder
                     $ Placeholder.Recursive var'name ty

  return (expr', ps, actual)
    where pred'to'placeholder (Is'In cl'name ty)
            = Placeholder (Placeholder.Dictionary cl'name ty)
```

Code listing 13 Definition of the **`Analyze`** type alias.

> We could probably just rely on the other parts of the implementation being correct and make the pattern matching non-exhaustive, but we do not consider that to be a good practice. It might cause some issues in the future and it would definitely lead to some confusion if for whatever reason that one predicate was not present in the context of that type. If that were to happen in our "defensive" implementation, we would report at least a somehow informative error message.

```
type Machine'State a = State Store (Either Evaluation'Error a)

type Store = Map Address Promise'Content

type Promise'Content = Either (Core, Environment) Value

type Environment = Map String Promise

type Address = Int

newtype Promise = Promise Address
```

Code listing 14 Definitions of **`Machine'State`** and related data types.

3.3 Implementation of Interpreter

In the final section of this chapter, we present a few implementation details of the interpreter. We cover two aspects of it.

The first one will be the implementation of the **`Machine'State`** type, a central piece of the interpreter. Once again, we utilize a specific monad to provide us with the facility necessary for evaluation. For good measure, we also show the implementation of function `force` as it is a prime example of a function operating with the mutable state provided by the **`Machine'State`**.

The second one will be a part of the `eval` function. We present a section that handles the evaluation of applications.

3.3.1 Working with Mutable State

The data type used for evaluation, **`Machine'State`**, only needs to provide a mutable state for our store to function correctly. Technically, we could also let it handle errors during the evaluation, but as it turns out, we can do that easily with a simple **`Either`** data type, so we do just that.

The listing 14 presents definitions of the **`Machine'State`** data type and the rest of the related, fundamental data types.

We can see that all the types presented, except one, are just simple type aliases. The only exception is just a simple **`newtype`** wrapper around one of those aliases. When it comes to those types, there is no hidden complexity. The evaluation is as simple as it can be.

We present the function `force` in listing 15. The source code of `force` is also very simple. If we chose to change the representation of the promise (the concept) in our implementation, the change should be localized to this function.

3.3.2 Evaluating Expressions

The last interesting piece of the interpreter is the `eval` function. It is rather a large function, that tends to get quite involved in some parts. It would not give us any special insight to present it as a whole. Instead we picked the part that demonstrates the laziness, or rather its implementation essence. That is—the evaluation of the (function) application. We present the corresponding code in listing 16.

As we can observe, when we evaluate an application, it might not necessarily be a function application. In some cases, the application might be representing a primitive operation.

In those cases, we delegate the evaluation to a small function called `do'prim'op`. In a way, that function provides an “escape” from the defined language. There is no other way. If we want

```
force :: Promise → Machine'State Value
force (Promise addr) = do
  store ← get

  let slot = Map.lookup addr store
  case slot of
    Nothing →
      return $ Left IncorrectAddress

    Just v →
      case v of
          Left (core, env) → do
            val ← eval core env
            case val of
              Left err → return $ Left err

              Right val → do
                store ← get
                let new'store = Map.insert addr (Right val) store
                put new'store
                return $ Right val

          Right val → return $ Right val
```

Code listing 15 Definition of the `force` function.

```
eval :: Core → Environment → Machine'State Value
eval (App fn arg) env = do
  r ← eval fn env
  case r of
    Left err → return $ Left err

    Right (Operator name) → do
      r ← eval arg env
      case r of
        Left err → return (Left err)
        Right val → do
          do'prim'op name val

    Right (Closure param body env') → do
      store ← get
      let size = Map.size store
      let next'addr = size
      let new'store = Map.insert next'addr (Left (arg, env)) store
      put new'store
      let new'env = Map.insert param (Promise next'addr) env'
      eval body new'env

    Right v →
        return $ Left
               $ Unexpected
               $ "function expression didn't evaluate to closure"
```

Code listing 16 Part of the `eval` function.

to, for example, add two integers, we need a "low-level machine" to do it for us. The specific implementation of that function is not at all interesting and we will not present it here.

In cases where the application represents an ordinary function application with a function expression on the left-hand side, we can observe part of the laziness. The key is in `arg` not being evaluated. Instead it is transformed into a promise and inserted into the store. The address of its slot is then put into the environment within which the body of the function will be evaluated. That way, if the function does not use its argument, the argument will never get forced. That is, the original expression for it will never get evaluated.

3.3.3 Evaluating Overloaded Expressions

The last aspect of the interpreter that we will cover is related to the specific sequence of steps necessary for the correct evaluation of expressions with a qualified type.

Suppose that we want to evaluate expression like `23 + 42` using the REPL. If we first ask for its type, we will be given this result: **`Num a ⇒ a`**. For obvious reasons, when we evaluate that expression in the REPL, we expect the result to be a numeric value. However, it should be clear, that we can not simply evaluate this expression as it is. Its type demonstrates a requirement of an instance/dictionary of **`Num`** for *some type*. That should directly tell us, that the expression has been elaborated into a version with placeholders, but those were not eliminated yet. That should be the next step. The only issue is, that we do not know what that specific type is. Nothing within that expression implies that it should be one numerical type or other.

This is once again a situation in which we will utilize type defaulting. According to the standard rules, the type of this expression will be defaulted to **`Int`** and the elimination of all placeholders within it can take place. This in turn produces an AST with no placeholders and as such, it can be directly translated into the core and evaluated.

If, for whatever reason, the defaulting could not be successful, the REPL will report an error. No evaluation or even lowering to the core will take place.

Chapter 4
Evaluation

An important aspect of the presentation of any programming language is a set of simple and short examples that can be presented. It can also serve as a lightweight test suit for the whole infrastructure.

Our implementation contains a set of positive and negative examples in the form of simple programs written in Glask. Those examples can be modified and inspected easily, as they are ordinary files in the project directory. At the same time, they are loaded when a command to run all tests is invoked. To test the implementation of the language, we run `cabal test`.

All the examples can be found in the directory `examples` in the root of the project. All of the examples in this thesis can also be tested in the REPL.

4.1 Operators

The implemented parser for the language allows defining custom prefix, infix, and postfix operators. They can be used wherever ordinary operators, like the ones Haskell has, would be used, or where function applications could be used.

Our test suite covers their use, including demonstrations of the consequences of the *Implicit Rule*.

4.2 DHM Type Inference

Besides the extensions, the type system of the language is capable of a simple type inference equivalent to the one of the DHM type system. Most of the examples in the test suite feature variety of bindings both implicit and explicit. In the cases of the implicit ones, it can be observed that the inference works as described in the thesis.

4.3 Data Types

The implementation fully covers the functionality of custom data types. We can define data types with zero constructors, one or more constructors, and we can also use a so-called record syntax when defining and pattern matching on values of those data types.

We can also create higher kinded types as well as so-called phantom types, using the same infrastructure. Finally, we fully support higher-rank types inside custom data types. This means that if a constructor takes a higher-rank type when we pattern match on the value and bind it to a variable, that variable has a correct higher-rank type within the corresponding scope.

Our testing examples contain a set of tests of all of those features mentioned above.

4.4 Type Classes and Instances

The support for type classes, instances, and qualified types is implemented for the most part. The language supports type checking for single-parameter type classes. The only missing part is compile-time desugaring of type class-based overloading for instances of subclasses. The desugaring of type classes, that are not subclasses of other classes, is implemented.

Besides that, our test suit contains examples of simple type classes and their instances, type classes and instances for higher kinded types, and instance declarations with additional type contexts.

4.4.1 Type Class Hierarchy

As we mentioned in chapter 2.5.1, we did not fully implement the support for class hierarchy. The part taking care of the type analysis for subclasses and their instances has been implemented. However, the placeholder approach was not extended enough to accommodate it.

From the strategy point of view, the implementation should be fairly straightforward. It will definitely require some careful additions, but all of it should be just a matter of simple, localized implementation. The infrastructure needed for such an addition is already present in the implementation.

We have decided to not follow through on it to save some time during the implementation and allocate it somewhere else—on more interesting and fundamental concepts.

As a consequence, this part of the implementation creates an opportunity for a future work.

4.5 Typed Holes

Another aspect of the implementation that would benefit from additional attention is the implementation of typed holes. We have implemented a basic version of the feature. Typed holes can be used in Glask programs and they report errors together with some very basic information about the type of the hole like where that type comes from, when available.

However, there is more that can be done. For example, we could report relevant bindings in the scope of the hole and in some simple cases even offer a possible expression to fill the hole. Once again, this extension would only be a matter of development time. And since there is nothing instrumental to cover in terms of the implementation, we have decided to omit the extended implementation from the language and the thesis. Once again it is left as an opportunity for future extension.

The testing examples show uses of holes within both implicit and explicit bindings.

4.6 Type Synonyms

The final aspect of the language that is left for future implementation is the support for type synonyms. We have covered the fundamental aspect of their implementation in chapter 2.4.9.

Because they do not add any complication or a significant additional value to the complete type system—they only represent a convenience layer on top of the existing infrastructure—we have chosen to omit their implementation from our work. This omission has no negative impact on the work and does not take away from its contribution. Type synonyms are a simple feature that does not interact in any special way with the rest of the system.

4.7 Higher-rank Types

The final aspect of the type system that has been implemented is support for higher-rank types. The feature is implemented in such a way that the type system can take advantage of a minimal amount of user-supplied type annotations.

Aside from the limitation related to the property of predicativity, those higher-rank types can be utilized at almost all places where ordinary types are expected.

Our test suit covers those as well.

4.8 Translation to Core

The implementation translates the rich surface representation into the smaller core language exactly as described in the thesis. It eliminates the type classes and other high-level syntactic constructs.

The evaluation of this part of the pipeline is merged with the following aspect of the implementation, the interpretation. The REPL is also able to produce a core representation for any valid expression and print it out.

4.9 Lazy Evaluation

The interpreter implements a non-strict evaluation strategy known generally as laziness. It is shown to work in a set of examples featuring various patterns that would make strict languages diverge.

Conclusion

This thesis set out to implement a statically typed, lazy, pure functional programming language. It aimed to focus on the various features of the type system, providing an insight into the implementation composed of various resources. Its secondary aim was to implement a parser for a flexible syntax. The flexibility comes in the form of first-class support for prefix and postfix user-defined operators.

It delivered what has been stated in the thesis proposal and on top of that, it also offered an introductory-level explanation of the concepts covered in the listed resources and their implementations. It can serve as additional reading material to tie together the contents of those resources for beginner language implementors.

In the rest of this chapter, we will offer a few observations about the implementation aspect of the work.

Monadic Infrastructure

In the early stages of this work, the implementation was utilizing a number of different monadic data structures. All of them were defined with no generalization—each part of the system had its own monadic structure. There was a specific monad for type inference and a different one for solving type constraints. The kind analysis also worked the same way. A few other parts of the system had their own monads as well.

The main motivation for this level of separation was a concern for type safety. Our aim was to model, at the type level, the extent to which specific operations could change the state. What is more, this level of separation allowed us to communicate, through the type annotations, what those specific functions *do not do*.

Eventually, we discovered that to get these assurances, we do not need to introduce so many distinct data types. In most cases, we can just abstract over the specific type and only require it to have some *abilities* utilizing the class overloading. For example, operations that only need to use one effect—error reporting—can specifically require the abstracted type to be an instance of `MonadError`. Then all functions that only require that ability can safely use it and we do not need to handle multiple monads in the same system.

To some extent, we have changed the original implementation to embrace this style. However, there are still few data structures that could be safely merged together.

Kinds within Types

Glask's implementation was not a project being developed "from scratch" even if the thesis makes it sound that way. In reality, it was an iteration of a simpler language, called *Frea* [31].

In that original implementation, we have made the representation of types and kinds entirely separated. This was a direct consequence of an iterative design and implementation.

One of the major challenges was making sure that all type substitutions were so-called *kind preserving*. That is the main reason why we have chosen a different approach with Glask. Following the footsteps of Jones in *Typing Haskell in Haskell* [2], we have decided to put kind information inside the type variables and constants.

This itself led to some challenges, mainly to a need of applying the kind substitution to all things involving types. In the end, however, it showed to be a very reasonable design decision.

Technical Apendix

```
while there are tokens to be read on the input:
  read the token
  if the token is:
    - non-operator:
      put it on the application queue

    - infix operator OP1:
      if the application queue begins with:
        - constructor C:
          transform the content of the application queue
            into a token representing C being applied
            to the rest of the queue
          put OP1 on the output queue
          empty the application queue

        - anything else:
          report an error and terminate
      "operator loop":
      while there is operator OP2 on top of the operator stack:
        if the OP2:
          - has lower precedence than OP1
              BREAK "operator loop"

          - has higher precedence than OP1:
              put OP1 on the output queue

          - has the same precedence as OP1:
            - if both are non-associative:
                report an error and terminate
            - if their associativity is not equal:
                report an error and terminate
            - if both are LEFT associative:
                put OP2 on the output queue
            - if both are RIGHT assoc:
                BREAK "operator loop"
      push OP1 on the operator stack

pop the entire operator stack onto the output queue
```

Code listing 17 Pseudocode for SYA on patterns.

Bibliography

1. DIJKSTRA, Edsger. *Algol 60 translation : An Algol 60 translator for the X1 and making a translator for Algol 60*. 1961. No. MR 34/61. Available also from: https://www.cs.utexas.edu/users/EWD/MCReps/MR35.PDF.
2. JONES, MARK P. Typing Haskell in Haskell. 2001. Available also from: https://web.cecs.pdx.edu/~mpj/thih/thih.pdf.
3. PEYTON JONES, Simon; VYTINIOTIS, Dimitrios; WEIRICH, Stephanie; SHIELDS, Mark. Practical Type Inference for Arbitrary-Rank Types. *J. Funct. Program.* 2007, vol. 17, no. 1, pp. 1–82. ISSN 0956-7968. Available from DOI: 10.1017/S0956796806006034.
4. WADLER, Philip; BLOTT, Stephen. How to Make Ad-Hoc Polymorphism Less Ad Hoc. *[No source information available]*. 1997. Available from DOI: 10.1145/75277.75283.
5. PETERSON, John; JONES, Mark. Implementing Type Classes. *ACM SIGPLAN Notices*. 1995, vol. 28. Available from DOI: 10.1145/173262.155112.
6. CHURCH, Alonzo. A Set of Postulates for the Foundation of Logic. *Annals of Mathematics* [online]. 1932, vol. 33, no. 2, pp. 346–366 [visited on 2022-06-13]. ISSN 0003486X. Available from: http://www.jstor.org/stable/1968337.
7. HINDLEY, R. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society* [online]. 1969, vol. 146, pp. 29–60 [visited on 2022-06-14]. ISSN 00029947. Available from: http://www.jstor.org/stable/1995158.
8. MILNER, Robin. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*. 1978, vol. 17, no. 3, pp. 348–375. ISSN 0022-0000. Available from DOI: https://doi.org/10.1016/0022-0000(78)90014-4.
9. DAMAS, Luís. *Type assignment in programming languages*. 1984. Available also from: https://hdl.handle.net/1842/13555. PhD thesis. University of Edinburgh, UK.
10. *The Glasgow Haskell Compiler*. Available also from: https://www.haskell.org/ghc.
11. *The Glorious Glasgow Haskell Compilation System User's Guide: The forall-or-nothing rule*. Available also from: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/explicit_forall.html#the-forall-or-nothing-rule.
12. KFOURY, A. J.; TIURYN, J. Type Reconstruction in Finite Rank Fragments of the Second-Order λ-Calculus. *Inf. Comput.* 1992, vol. 98, no. 2, pp. 228–257. ISSN 0890-5401. Available from DOI: 10.1016/0890-5401(92)90020-G.
13. *Haskell Wiki: IO inside*. Available also from: https://wiki.haskell.org/IO_inside.
14. *Alex: A lexical analyser generator for Haskell*. Available also from: https://www.haskell.org/alex.

15. *Happy—The Parser Generator for Haskell*. Available also from: https://www.haskell.org/happy.

16. *The Glorious Glasgow Haskell Compilation System User's Guide: Syntactic extensions*. Available also from: https://downloads.haskell.org/~ghc/6.10.2/docs/html/users_guide/syntax-extns.html.

17. HERBRAND, Jacques. *Recherches sur la théorie de la démonstration*. 1930. Available also from: http://eudml.org/doc/192791.

18. ROBINSON, J. A. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*. 1965, vol. 12, no. 1, pp. 23–41. ISSN 0004-5411. Available from DOI: 10.1145/321250.321253.

19. MARTELLI, Alberto; MONTANARI, Ugo. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 1982, vol. 4, no. 2, pp. 258–282. ISSN 0164-0925. Available from DOI: 10.1145/357162.357169.

20. *Assorted concrete container types*. Available also from: https://hackage.haskell.org/package/containers.

21. PIERCE, Benjamin C. *Types and Programming Languages*. 1st. The MIT Press, 2002. ISBN 0262162091.

22. *The Glorious Glasgow Haskell Compilation System User's Guide: Kind Polymorphism—Polykinds*. Available also from: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/poly_kinds.html#extension-PolyKinds.

23. *The Glorious Glasgow Haskell Compilation System User's Guide: Explicitly-kinded quantification—KindSignatures*. Available also from: https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/kind_signatures.html#extension-KindSignatures.

24. *The Haskell 2010 Language: Kind Inference*. Available also from: https://www.haskell.org/onlinereport/haskell2010/haskellch4.html#x10-970004.6.

25. *Haskell 98 Language and Libraries, The Revised Report: Declarations and Bindings*. Available also from: https://www.haskell.org/onlinereport/decls.html.

26. *Haskell Wiki: Monomorphism restriction*. Available also from: https://wiki.haskell.org/Monomorphism_restriction.

27. *Haskell Wiki: FAQ*. Available also from: https://wiki.haskell.org/FAQ#What.27s_the_difference_between_Integer_and_Int.3F.

28. *Online conversation with creator of Idris programming language*. 2022. Available also from: https://www.reddit.com/r/ProgrammingLanguages/comments/thhyet/implementing_typed_holes/.

29. *ghc-prim-0.8.0: GHC primitives*. Available also from: https://hackage.haskell.org/package/ghc-prim-0.8.0/docs/GHC-Prim.html#v:seq.

30. *mtl: Monad classes, using functional dependencies*. Available also from: https://hackage.haskell.org/package/mtl.

31. *A simple and lazy programming language with Damas-Hindley-Milner type inference and higher kinded types*. Available also from: https://github.com/lambduli/frea.

Contents of the Attached CD

- `readme.txt` ..... brief description of the contents of the CD
- `src`
  - `impl` ..... implementation source code
  - `thesis` ..... source code of the thesis in LaTeX
- `thesis.pdf` ..... text of the thesis in PDF