A compiler for Cada

Submitted May 2013 in partial fulfilment of the conditions of the award of the degree of MSci (Hons) Computer Science.

Michael Benjamin Gale
School of Computer Science
University of Nottingham
mbgale@acm.org

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature _______________________________

Date ________/_______/_______
Abstract

Monads have evolved into a powerful and versatile tool for functional programming, but remain a challenging concept for those versed in other paradigms. However, their usefulness encourages us to explore means by which they can be made more accessible. While efforts have been made to introduce syntactic constructs for this purpose, such as the do-notation and monad comprehensions in Haskell, little attention has been paid to developing syntax specifically for the unique helper functions which accompany individual monads.

High-level programming languages aim to hide machinery in an underlying system which is frequently used in programs. In cases where no such abstractions are available, programmers may have to write repetitive or difficult code. For this reason, it is desirable to enrich functional programming languages with abstractions for some frequently used monads.

Our goal in this dissertation will be to explore the role of the state monad, to design a functional programming language which makes effective use of our observations, and to implement a compiler for it in Haskell. In the process of developing this language, which we shall name Cada, we will explore the foundations of functional programming and learn that implementing a compiler for a modern functional language is nothing to be afraid of.
5 Implementation

5.1 Architecture ............................................. 42
  5.1.1 Directory layout ..................................... 43
5.2 Configuration ............................................. 44
5.3 Lexical analysis ......................................... 46
  5.3.1 Tokens ................................................ 47
  5.3.2 Multi-line comments ................................. 47
5.4 Parser ..................................................... 48
  5.4.1 Parser monad .......................................... 49
  5.4.2 Left-recursion ....................................... 52
  5.4.3 Ambiguities .......................................... 53
  5.4.4 Locations ............................................ 54
5.5 Abstract syntax tree .................................... 55
  5.5.1 Construction and validation ......................... 56
5.6 Pretty printing .......................................... 59
  5.6.1 Pretty printing combinators ........................ 59
  5.6.2 Pretty printing of abstract representations ...... 60
5.7 Type system .............................................. 61
  5.7.1 Type environments .................................. 61
5.8 Module system ........................................... 62
  5.8.1 Interface files ...................................... 64
5.9 Kind inference ........................................... 65
5.10 Conversion .............................................. 67
5.11 Type inference .......................................... 69
  5.11.1 Type errors ......................................... 69
  5.11.2 Type functions ..................................... 70
5.12 Backend .................................................. 72
  5.12.1 Code generation .................................... 73
  5.12.2 Runtime support .................................... 74
  5.12.3 Entry point ......................................... 75
5.13 Testing .................................................... 75

6 Conclusion .................................................. 78
6.1 Evaluation of the implementation ..................................... 80
6.2 Future work ...................................................................... 82
  6.2.1 Constructors and modifiers ......................................... 82
  6.2.2 Generalisation ............................................................. 82
  6.2.3 Different type inference algorithm ................................. 83
  6.2.4 Language extension for Haskell ................................... 83

A Appendix .............................................................................. 88
  A.1 System F ......................................................................... 88
  A.2 Hindley-Milner ................................................................. 89
  A.3 Lexical grammar .............................................................. 91
  A.4 Context-free grammar ..................................................... 93
  A.5 Abstract syntax tree ....................................................... 98
  A.6 Type system ...................................................................... 103
\textbf{\large Introduction}

Computer systems are responsible for the control of many aspects of our modern lives, ranging from wrist watches to rockets. These exponentially increasing responsibilities mean that we are obliged to ensure that our software is more reliable than ever. At the same time, the growing complexity of programs poses challenges as to how we can achieve this goal.

Traditionally, software engineers ran their programs through sets of rigorous test cases to ensure that there were no bugs. While this approach ruled out a reasonable number of faults, it has never been able to track down all of them. To make matters worse, the growing size of software means that more and more time is needed to design a comprehensive set of test cases.

It becomes apparent that a different solution is needed to counteract the increasingly high chances for a fault to go unnoticed. Purely functional programming languages provide a promising answer to this problem. If we model computations using mathematical functions, as opposed to procedures which manipulate a computer’s memory, then we can formally prove properties about program using common techniques such as, for example, mathematical induction.

Purity is its own worst enemy, however, as it forces us to make everything about functions explicit. Tasks which are easily accomplished in imperative languages can quickly turn into major obstacles for such a language. An even bigger and seemingly insurmountable problem is the need of side-effects, such as terminal input and output, in real software systems.

Despite these obstacles, a series of research discoveries has turned the tide in the favour of pure languages. Moggi (1991) introduced \textit{monads}, a concept which originates in category theory, to functional programming as an approach to structure programs. Wadler (1992) was then quick to use them as abstractions for \textit{effects}, such as \textit{e.g.} failure and exceptions.

Finally, Peyton Jones (2001) showed that we can safely hide side-effects using a monad if we keep such impure computations separate from pure functions with the help of a type system such as Haskell’s (Peyton Jones, 2002).
These breakthroughs have laid the foundations which were required to make purely functional programming languages an attractive option for the software industry. As a result, monads have become the superstars of functional programming and have even made their way into languages of other paradigms.

Our focus will remain on Haskell, however, where syntactic sugar for monads has been added to the language in the shape of the do-notation, list comprehensions, and the more general monad comprehensions. However, we can observe that, with the exception of list comprehensions, all of these notations are compatible with any monads. The obvious implication of this is that none of the syntactic sugar makes use of helper functions which accompany individual monads. Many of the effects which were implemented using monads in Wadler’s paper have no corresponding syntactic abstractions, despite their significance in software development.

1.1 Project description

We argue that software programs typically have to maintain a lot of state such as database connections, file handles, and configuration values which may have to be read or changed. If it was not for the state monad all of this information would have to be explicitly passed from function to function. Even with the help of a monad this approach still makes it much more complicated to achieve what would be trivial tasks in imperative or object-oriented languages and requires programmers to understand many of the inner workings of monads.

It is in the nature of high-level programming languages to provide abstractions for such machinery. This project’s goal will therefore be to explore modern, purely functional languages in depth through the design of Cada, a language based on Haskell98, and an implementation of a compiler for it. We will also investigate the role of mutable state in the context of this language. It makes sense that tools which simplify the work of software engineers will make them more productive and contribute to the usefulness of a language. Thus we wish to accomplish the following goals:

- Gain an understanding of the relevant programming language theory, such as the λ-calculus and its typed variants, as well as of concepts in functional programming, including monads and equational reasoning. Introductions to all these topics are given in chapter 2.

- Identification of recurring patterns in which the state monad is used. This will serve as the foundation for the design of the abstractions we wish to provide in our language. We note that our aim is to reduce redundant code and to hide as much of the underlying machinery as possible. We present our observations and ideas in chapter 3.
• Design of syntactic constructs based on the findings of the previous goal in a way such that they are compatible with the usual definitions of the state monad and state monad transformer. This will enable us to work with functions in existing libraries. A formal specification of our language is described in chapter 4.

• Implementation of a compiler for the resulting language to apply the results of the preceding goals, understand the limitations we had to impose, incorporate the ideas we have developed, and to test and compare programs written using the new abstractions and without them in comparable languages. We discuss the details of our implementation in chapter 5.

Though, our intention is not to develop a fully featured general-purpose programming language as compilers can be very large and complex pieces of software. Therefore, it is vital that we limit our scope to the aspects which are significant to our investigation, by imposing the following limitations:

• Runtime performance is not a primary concern for us. Therefore, it would be undesirable for us to spend a significant amount of time on the implementation of a code generator which, for a functional programming language, would also rely on an extensive runtime system. Luckily, we have the option to either implement an interpreter in our implementation language or to compile to another high-level language.

• As a result of the above, we are also not too worried about optimising the generated code. If we compile to an existing language, then the respective compiler will most likely take care of optimisations for us.

• Some features of general-purpose languages, such as e.g. floating point numbers are vital for graphics processing and scientific calculations, but their correct implementation is often very time consuming. More importantly, they are not required for us to accomplish our goals and we may therefore give such functionality a low priority.

1.2 Recommended reading

This dissertation is written for undergraduate students in computer science with no expert knowledge of the field. We assume that readers are familiar with a functional programming language, preferably Haskell, and formal specifications of programming languages. Some good reference texts in the respective areas include Programming in Haskell (Hutton, 2007) and Types and Programming Languages (Pierce, 2002).
1.3 Acknowledgements

This dissertation has benefited greatly from the input of the members of the Functional Programming Laboratory at the University of Nottingham. In particular, I am thankful for the useful advice and support from Henrik Nilsson, Laurence E. Day and Florent Balestrieri. I am very grateful for the discussions I have had with Alan Mycroft at the Computer Laboratory of the University of Cambridge and Simon Peyton Jones at Microsoft Research. Lastly, many thanks goes to my supervisor, Graham Hutton, who has been very supportive from the conception to the end of this project, and especially during the final few weeks.
### Background

Imperative programming languages have traditionally formed the dominant tool for the development of computer software, due to their abstractions for the underlying hardware which permits developers to quickly write powerful and efficient programs for virtually any task. Few restrictions on what programmers may do in them quickly leads to a situation where it becomes impossible to verify that programs do what they are supposed to. High-profile failures in safety critical systems, such as the arithmetic overflow error aboard the Ariane 5 rocket bear witness to that (Dowson, 1997).

Purely functional programming languages aim to solve this dilemma through the elimination of side-effects, thus enabling us to use equational reasoning to prove properties of programs. However, pure programs on their own are useless as there is, for example, no way to interact with a user through a terminal, read data from a sensor, or write to a file. Peyton Jones (2001) proposed a workaround for this problem, which allows impure computations to be embedded in a pure language by getting the type system to keep both worlds separate.

Since then, languages like Haskell have gained much popularity outside of the academic community, but many tasks remain easier to accomplish in imperative or object-oriented languages. In the remainder of this chapter we will look at the key ideas in functional programming needed to understand why this is the case and to establish the theoretical background of this project.

### 2.1 A tale of two calculi

According to the Church-Turing Thesis, only values of functions which can be represented using a Turing machine are computable and no other process can carry out more powerful computations. Other models, such as the $\lambda$-calculus (Church, 1941) have been shown to be equivalent to Turing machines and thus reinforce this hypothesis. This may be surprising, given the simplicity of terms in this model:

$$ e = x \mid \lambda x.e \mid (e e) $$

These three productions correspond to variables, abstractions, and function application. We distinguish between free and bound variables, where abstractions are used
to bind names in subterms. For example, \( x \) is bound in \( \lambda x.x \), but free in \( \lambda y.x \). There is only one reduction rule known as \( \beta \)-reduction which describes function application:

\[
(\lambda x.e_1) \ e_2 \ \Rightarrow_\beta \ [x \mapsto e_2]e_1
\]

A subterm where \( \beta \)-reduction may be applied is known as a \( \beta \)-redex. If a term contains none, then we say that it is in \( \beta \)-normal form. Substitution in the \( \lambda \)-calculus is capture-avoiding. If, for example, \( e_2 \) contains names which are bound in \( e_1 \), then they will be given new names. Two terms in the \( \lambda \)-calculus are equivalent up to renaming, which we describe as \( \alpha \)-conversion:

\[
\lambda x.e \ \Leftrightarrow_\alpha \ \lambda y.\ [x \mapsto y]e \ \text{ where } y \text{ is not free in } e
\]

For the purpose of reasoning about our programs, let us introduce one final transformation, known as \( \eta \)-conversion:

\[
f \ \Leftrightarrow_\eta \ \lambda x.f \ x \ \text{ where } x \text{ is not free in } f
\]

These are all of all the basic principles of the \( \lambda \)-calculus we will need, and one may now be confused about how anything useful can be computed using such a simple system. The solution may not be immediately obvious: in order to do this, we first require some notion of data, but obviously the syntax of the \( \lambda \)-calculus does not provide for this. Instead, data can be encoded in the \( \lambda \)-calculus using the Church encoding. For example, a Church numeral represents a natural number \( n \in \mathbb{N} \) using \( n \)-many applications of some variable \( f \) to some other variable \( x \):

\[
\begin{align*}
\text{ZERO} & \equiv \lambda f.\lambda x.x \\
\text{ONE} & \equiv \lambda f.\lambda x.f \ x \\
\text{TWO} & \equiv \lambda f.\lambda x.f \ (f \ x) \\
\end{align*}
\]

Lists, pairs, etc. can be encoded in a similar fashion. We are now able to perform computations using higher-order functions, which expect their arguments to be data in this encoding:

\[
\text{PLUS} \equiv \lambda m.\lambda n.\lambda f.\lambda x.m \ f \ (n \ f \ x)
\]

This term can be used to add up two natural numbers in the Church encoding to produce the resulting Church numeral. As an example, suppose we want to calculate \( 1 + 1 \) in the \( \lambda \)-calculus, then we can do this simply using \( \beta \)-reduction:
A compiler for Cada

\( PLUS \text{ ONE ONE} \)
\[ \equiv \{ \text{definition of } PLUS \} \]
\[ (\lambda m.\lambda n.\lambda f.\lambda x.m \ (n \ f \ x)) \ \text{ONE ONE} \]
\[ \Rightarrow \beta \{ \beta\text{-reduction for } m/\text{ONE and } n/\text{ONE} \} \]
\[ \lambda f.\lambda x.\text{ONE} \ f \ (\text{ONE} \ f \ x) \]
\[ \Rightarrow \beta \{ \text{ONE} \ f \ x \text{ reduces to } f \ x \} \]
\[ \lambda f.\lambda x.\text{ONE} \ f \ (f \ x) \]
\[ \Rightarrow \beta \{ \text{ONE} \ f \ (f \ x) \text{ reduces to } f \ (f \ x) \} \]
\[ \lambda f.\lambda x.f \ (f \ x) \]
\[ \equiv \{ \text{definition of TWO} \} \]

TWO

Note that definitions are merely used for our convenience and do not form part of the syntax of the \( \lambda \)-calculus. Thus, our definitions cannot be recursive and must be acyclic. However, we know that recursion is essential to construct loops which, in turn, are required for Turing completeness.

2.1.1 Recursion

Recursion can be encoded in the \( \lambda \)-calculus using a fixed-point combinator. In other words, we can describe a higher-order function which calculates the fixed-point of some other function. The most commonly used of these fixed-point combinators is the Y-combinator shown below:

\[ Y \equiv \lambda f.(\lambda x.f \ (x \ x)) \ (\lambda x.f \ (x \ x)) \]

Conceptually, recursion requires a function to be able to reference itself. The Y-combinator accomplishes this by providing the specified function with a continuation which allows it to do this. To see how it works, consider the example below:

\[ (\lambda f.(\lambda x.f \ (x \ x)) \ (\lambda x.f \ (x \ x))) \ g \]
\[ \Rightarrow \beta \{ \beta\text{-reduction for } f/g \} \]
\[ (\lambda x.g \ (x \ x)) \ (\lambda x.g \ (x \ x)) \]
\[ \Rightarrow \beta \{ \beta\text{-reduction for } x/(\lambda x.g \ (x \ x)) \} \]
\[ g \ ((\lambda x.g \ (x \ x)) \ (\lambda x.g \ (x \ x))) \]
\[ \equiv \{ \eta\text{-expansion, definition of } Y \} \]
\[ g \ (Y \ g) \]
We can observe that \( g \) is now applied to the initial term we had, thus giving it a
continuation using which it may refer to itself. Note that this term is an example of
one which has no \( \beta \)-normal form, because we could choose to repeatedly evaluate \( Y \ g \)
as the evaluation order is unspecified. As a result, we say that the \( \lambda \)-calculus is not
strongly normalising as there exist terms for which evaluation may not terminate.

### 2.1.2 Simply-typed \( \lambda \)-calculus

Previously, we have talked informally about expectations we have about the argu-
ments of functions. For example, we said that \( m \) and \( n \) in \( PLUS \) should be Church
numerals, but there is no way for us to enforce such a restriction. This is where one
motivation for \textit{types} comes from: if we can describe formally what the properties of
terms should be and if there is a way to check this, then our programs are less likely
to get stuck or do something unexpected. In the simply-typed \( \lambda \)-calculus (or \( \lambda \rightarrow \)), we
introduce a new syntax for types which we will use to describe terms:

\[
\tau = \tau \rightarrow \tau \mid T \in B
\]

Terms are either functions from one type to another, or values belonging to elements
of the set of base types \( B \). The term syntax changes slightly such that all abstractions
must be annotated with a type for the variable which they are binding:

\[
e = x \mid \lambda x : \tau . e \mid (e e)
\]

For example, if we choose \( B = \{ \text{Nat} \} \), then we could describe an identity function
which expects its argument to be a natural number:

\[
\text{ID} \text{NAT} \equiv \lambda x : \text{Nat}.x
\]

Of course defining such terms in itself is not good enough as we need a way to
verify the \textit{soundness} of our constructions. For example, we would expect \( \text{ID} \text{NAT} \) to
be described as a function of type \( \text{Nat} \rightarrow \text{Nat} \) and we would want a term such as
\( \text{ID} \text{NAT} \ \text{ID} \text{NAT} \) to be rejected because the argument of the first \( \text{ID} \text{NAT} \) is not of
type \( \text{Nat} \). For this purpose, there exists a set of \textit{typing judgements} which are used to
determine the types of terms in \( \lambda \rightarrow \). Each such rule is given in the following format:

\[
\begin{array}{c}
\text{Premise} \\
\vdots
\end{array}
\]

We write a list of \textit{premises} (pre-conditions) above a line, below which we can find a
\textit{conclusion}. If all premises are met, then we can conclude whatever is written below
the line. If no rule applies, then a term is said to be \textit{ill-typed}. Figure 2.1 contains
the typing judgements for \( \lambda \rightarrow \) where \( \vdash \) denotes the ternary typing relation between
contexts, terms, and types.
(x : τ) ∈ Γ \quad \frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma. e : \sigma \to \tau} \quad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 e_2 : \tau} \\

\text{Figure 2.1: Typing judgements for } \lambda \to

In order to derive that, for example, \textit{IDNAT} is of type Nat \to Nat, we can construct a proof tree by starting with a rule where the term in the conclusion matches the format of our current term. \textsc{TAbs} is our only option here, but in more complicated deduction systems, there may be more choice. Note that all variables in the typing judgements, such as \tau, x, and e_1, above are meta variables and can be replaced with the corresponding terms in our derivation:

\frac{(x : \text{Nat}) \in \Gamma, x : \text{Nat}}{\Gamma, x : \text{Nat} \vdash x : \text{Nat}} \quad \frac{\Gamma \vdash \lambda x : \text{Nat}. x : \text{Nat} \to \text{Nat}}{\text{TAR} \quad \text{TABS}}

\Gamma \vdash \lambda x : \text{Nat} \to \text{Nat}

\Gamma is the typing environment or context, which can be thought of as a set containing all known typings. Initially, we do not make any assumptions about the contents of \Gamma in the derivation above, but add to it when a new typing is introduced in \textsc{TAbs}.

Finally, let us note that the simply-typed \lambda-calculus is strongly normalising as terms which are not guaranteed to have a normal form, such as the Y-combinator, cannot be typed and are thus rejected. However, this means that, unlike the \lambda-calculus, \lambda \to is not Turing complete. Should recursion be required, it is possible to cheat a little bit by adding a typing \textit{fix} : (\tau \to \tau) \to \tau for all types \tau where \textit{fix} has the operational semantics of the Y-combinator.

While \lambda \to in the presentation we have shown here is not particularly well suited for use in programming languages and serves a more theoretical purpose, Plotkin (1977) presented a programming language, similar to the simply-typed \lambda-calculus, with the addition of types and literals for the booleans and natural numbers, as well as recursion in the style outlined above. Readers may find an implementation of this language in Haskell, written by the author of this dissertation, online\footnote{\url{https://github.com/mbg/simply-typed-ext}}.
2.2 System F

Suppose that we are using the simply-typed $\lambda$-calculus as a programming language with $B = \{ \text{Bool}, \text{Nat} \}$ and wish to define the identity function. This task would be easy in the untyped $\lambda$-calculus as we can simply write $\lambda x.x$. However, in $\lambda \to$, we need to specify a type for $x$. If we say that $x$ is of type $\text{Bool}$, then our identity function can only be applied to arguments of type $\text{Bool}$, but not natural numbers. We can attempt to solve this dilemma by defining an identity function for each type:

$$
\begin{align*}
\text{IDBOOL} & \equiv \lambda x : \text{Bool}.x \\
\text{IDNAT} & \equiv \lambda x : \text{Nat}.x \\
\end{align*}
$$

Despite having a solution to our initial problem, we now have to give a definition for every single type. However, there are infinitely-many types and we can observe that the two definitions are almost identical, except for the different types in the abstraction. Girard (1971) and Reynolds (1974) independently discovered a way to get around this problem, through the addition of type abstractions and universally quantified type variables to $\lambda \to$, resulting in a calculus now known as System F or the second-order $\lambda$-calculus. A full reference of this system may be found in appendix A.1.

In order to construct a polymorphic identity function, we need to parametrise it over some type using the newly added type abstractions. We will refer to this style of polymorphism as parametric polymorphism:

$$
ID = \Lambda \alpha.\lambda x : \alpha .x
$$

Here, $\Lambda$ is a type abstraction which can bind a type variable to some type, in the same way an abstraction binds a variable to some term. Let us demonstrate how the type of the above term may be derived:

$$
\begin{array}{c}
(x : \alpha) \in \Gamma, x : \alpha \quad \text{TVAR} \\
\Gamma, x : \alpha \vdash x : \alpha \quad \text{TAbs} \\
\Gamma \vdash \lambda x : \alpha .x : \alpha \to \alpha \quad \alpha \notin \text{TV}(\Gamma) \quad \text{TTyAbs} \\
\Gamma \vdash \Lambda \alpha .\lambda x : \alpha .x : \forall \alpha .\alpha \to \alpha
\end{array}
$$

We conclude that $ID : \forall \alpha .\alpha \to \alpha$ which can simply be read as “for all types $\alpha$, $ID$ is a function from $\alpha$ to $\alpha$”. For example, we can now reconstruct our earlier definitions of $\text{IDBOOL}$ and $\text{IDNAT}$ using $ID$ by applying it to the respective types:

$$
\begin{align*}
\text{IDBOOL} & \equiv ID \text{ Bool} \\
\text{IDNAT} & \equiv ID \text{ Nat}
\end{align*}
$$
It is easy to show that these terms yield the previous types, using the TTYApp rule:

\[
\frac{\Gamma, \text{id} : \forall \alpha. \alpha \rightarrow \alpha \vdash \text{id} : \forall \alpha. \alpha \rightarrow \alpha}{\Gamma, \text{id} : \forall \alpha. \alpha \rightarrow \alpha \vdash \text{id} \text{ Bool} : \text{Bool} \rightarrow \text{Bool}} \quad \text{TTYApp}
\]

Similarly to our previous experience with the simply-typed \( \lambda \)-calculus, our brief presentation of System F is largely of theoretical use to understand how we may describe terms using polymorphic types. In order to make this calculus practical for use as the foundation of a programming language, it would be beneficial to get rid of type annotations in terms. Consider the following example, where we define Church booleans and functions on them:

\[
\text{TRUE} \equiv \lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. x
\]

\[
\text{FALSE} \equiv \lambda \alpha. \lambda x : \alpha. \lambda y : \alpha. y
\]

\[
\text{AND} \equiv \lambda x : \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha. \lambda y : \forall \alpha. \alpha \rightarrow \alpha \rightarrow \alpha. x (\forall \alpha. \alpha \rightarrow \alpha) y \text{ FALSE}
\]

As we can see, the type annotations in the example above make the definition of \text{AND} incredibly verbose. If programmers were required to write programs in this style, they may easily become frustrated.

### 2.2.1 Church vs Curry

Our definitions of \( \lambda \rightarrow \) and System F are in the Church-style, where \( \lambda \)-abstractions must annotate the variable they are binding with a type. For use in programming languages, we prefer the Curry-style, where we take terms from the pure \( \lambda \)-calculus and assign types from e.g. the simply-typed \( \lambda \)-calculus to them through a process known as type reconstruction or type inference.

Unfortunately for us, Wells (1998) proved that this process is undecidable for System F types whose ranks are greater than 2. The rank of a type is determined by the greatest depth of a quantifier to the left of an arrow:

- \( f : \forall \alpha. (\alpha \rightarrow \alpha) \) \quad \text{Rank-1 type}
- \( g : \forall \alpha. (\alpha \rightarrow (\forall \beta. \beta)) \) \quad \text{Rank-1 type}
- \( h : (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \text{Bool} \) \quad \text{Rank-2 type}
- \( i : ((\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \text{Bool}) \rightarrow \text{Bool} \) \quad \text{Rank-3 type}

To better understand the intuition behind this problem, let us consider the following term where we use some syntactic sugar to denote pairs:

\[
\text{FUN} \equiv \lambda g. (g \ 3, \ g \text{ True})
\]

What should the System F type of this term be? There are multiple permissible types, including \((\forall \alpha. \alpha \rightarrow \alpha) \rightarrow (\text{Nat}, \text{Bool})\) and \((\forall \alpha. \alpha \rightarrow \text{Nat}) \rightarrow (\text{Nat}, \text{Nat})\). This in itself
is not a problem, because we could just arbitrarily choose one type. However, if this term is part of a larger one and fails to match another type further down the line, we need to go back and choose differently. Of course, there are infinitely-many choices to try, one of which may be right, such that the problem becomes undecidable.

2.2.2 Hindley-Milner

Instinctively, one may now conclude that it would make sense to design a programming language based on the pure $\lambda$-calculus with System F types which are restricted to rank-2. However, the result probably would not be all that useful because, for starters, we would have to let the type inference algorithm choose arbitrary typings in some cases such as our previous example. Additionally, it may be difficult to decide when a term should be given a polymorphic type. For example, if the identity function occurs deeply nested within some larger expression, would it be safe to assign a polymorphic type to it?

Hindley (1969) and later Milner (1978) described a system which now forms the foundation of many functional programming languages, including Haskell and ML. Their idea was to make a clear distinction between polymorphic and monomorphic types, thus restricting polymorphic types to at most rank-1. The syntax of this system, which we shall refer to as Hindley-Milner from here on, is shown in appendix A.2. We can note that a distinction carries through to the typing judgements, where polytypes are only permitted if we use $\sigma$.

However, we need to solve the issue of having to decide when terms should have a polymorphic types and when they should not. Hindley-Milner solves this problem using what is now known as let-polymorphism. Polymorphic typings may be introduced at any point using let-expressions with the help of $\text{TGen}$ which generalises a monomorphic type to a polymorphic type by adding quantifiers for the free type variables. As an example for this, let us show how the polymorphic type of the identity function may be derived:

\[
\begin{align*}
(x : \alpha) \in \Gamma, x : \alpha & \quad \text{TVar} \\
\Gamma, x : \alpha \vdash x : \alpha & \quad \text{TAbs} \\
\Gamma \vdash \lambda x. x : \alpha \rightarrow \alpha & \quad \text{TGen} \\
\Gamma \vdash \lambda x. x : \forall \alpha. \alpha \rightarrow \alpha & \quad \text{TLet}
\end{align*}
\]

As we can see, the rule $\text{TLet}$ expects a polytype $\sigma$ to be added to the type environment, thus forcing us to use $\text{TGen}$ first in this case because most other typing judgements, such as $\text{TAbs}$, expect and provide monomorphic types. Indeed, type variables will never be instantiated with polytypes. This is accomplished with the help of $\text{TInst}$.
which allows us to instantiate type variables using the specialisation rule shown in Figure A.4. In the following example, we assume that \((id : \forall \alpha. \alpha \to \alpha) \in \Gamma\) to derive the type of id applied to itself:

\[
\begin{align*}
\text{TVAR} & \quad \frac{(id : \forall \alpha. \alpha \to \alpha) \in \Gamma}{\Gamma \vdash id : \forall \alpha. \alpha \to \alpha} \\
\text{TINST} & \quad \frac{\forall \alpha. \alpha \to \alpha \sqsubseteq \beta \to \beta} {\Gamma \vdash id : \beta \to \beta}
\end{align*}
\]

\[
\begin{align*}
\text{TVAR} & \quad \frac{(id : \forall \alpha. \alpha \to \alpha) \in \Gamma}{\Gamma \vdash id : \forall \alpha. \alpha \to \alpha} \\
\text{TINST} & \quad \frac{\forall \alpha. \alpha \to \alpha \sqsubseteq (\beta \to \beta) \to \beta \to \beta} {\Gamma \vdash id : (\beta \to \beta) \to \beta \to \beta}
\end{align*}
\]

\[
\frac{\Gamma \vdash id : (\beta \to \beta) \to \beta \to \beta} {\Gamma \vdash id\ id : \beta \to \beta}
\]

We carefully choose to instantiate \(\alpha\) with \(\beta \to \beta\) in the left sub-derivation so that it later matches the type returned by the right sub-derivation. In order for Hindley-Milner to become truly useful, we need a sound and complete algorithm which is able to make such choices for us.

Luckily, the system was presented together with Algorithm \(\mathcal{W}\) (Figure A.5) which was proven to be sound and complete by Damas and Milner (1982). This algorithm encodes the deductive system used above, with three points of interest: polytypes obtained from \(\Gamma\) are always instantiated with fresh type variables, monotypes are generalised in let-bindings before they are added to the type environment, and unification (Robinson, 1965) is used to find the most-general unifiers for types obtained from the sub-terms in applications, in order to prevent the algorithm from trying to predict the right types during instantiation.

### 2.3 Monads

Both, an advantage and disadvantage, of the \(\lambda\)-calculi we have seen is that everything must be explicit, as functions are pure in the sense that, given the same set of arguments, they will always compute the same result. Moggi (1991) discovered that monads, which are abstract structures in category theory, can be used to model effectful computations.

While we will not concern ourselves too much with category theory in this dissertation, a short introduction to the topic aimed at readers with backgrounds in Computer Science may be found in Basic Category Theory for Computer Scientists Pierce (1991). Category Theory (Awodey, 2006) and Categories for the Working Mathematician (Mac Lane, 1998) provide more comprehensive texts.


### 2.3.1 In Haskell

Suppose that $\text{Hask}$ is the category of all Haskell objects, then an object in this category is just a type. In other words, $\text{Hask}$ corresponds to the *kind* of types. We denote this kind using $\star$ in the syntax of Haskell. An endofunctor over $\text{Hask}$ is thus a function on the type level whose domain and codomain are both $\star$, which is written as $\star \to \star$.

Any type $m$ of this kind can be a monad if there exist functions of types $a \to m \ a$ and $m \ a \to (a \to m \ b) \to m \ b$ which adhere to identity and associativity laws shown below, collectively known as the monad laws:

- **Left identity:** $\text{return } a \gg= f = f \ a$
- **Right identity:** $m \gg= \text{return } = m$
- **Associativity:** $(m \gg= f) \gg= g = m \gg= (\lambda x \to f \ x \gg= g)$

We define an appropriately named type class to represent monads, which suitable types can be made an instance of:

```haskell
class Monad m where
    return :: a \to m a
    (\gg=) :: m a \to (a \to m b) \to m b
```

As an example for such a monad, we will consider a simple expression language which represents the untyped $\lambda$-calculus. To begin, we will define an appropriate data type, parametrised over a type $a$ which we will use for variables:

```haskell
data Expr a = Var a
            | Abs a (Expr a)
            | App (Expr a) (Expr a)
```

This data type has one parameter which is not applied to any other types in the right-hand side of the definition and therefore the kind of $\text{Expr}$ is $\star \to \star$ as we would expect. In order to see whether this type is a monad, we need to create an instance of the $\text{Monad}$ type class and find definitions for the $\text{return}$ and $\gg=\,$ functions which satisfy the monad laws.

Finding such definitions is typically just a “game with types” and, by Curry-Howard correspondence, this is not too surprising as we are essentially trying to find proofs for the existence of functions with the given types. There is only one way to wrap a value of type $a$ into $\text{Expr}$ and thus it is easy to give a definition for $\text{return}$:

$$\text{return } = \text{Var}$$
Our goal for $\gg=$ is to apply a function $f :: a \rightarrow Expr\ b$ to all values of type $a$ which are embedded in a value of type $Expr\ a$. This is only possible if we traverse a given expression recursively and use pattern matching on each of $Expr$’s constructors to determine what we should do. Depending on the constructor, we can either apply $f$ to a value of type $a$ or call ourselves recursively on a subexpression:

\[
(Var\ x) \gg= f = f\ x \\
(Abs\ x\ e) \gg= f = Abs\ x\ (e \gg= f) \\
(App\ l\ r) \gg= f = App\ (l \gg= f)\ (r \gg= f)
\]

Interestingly, the effect of this monad is that of substitution, which happens to be the only means by which computation can be performed in the $\lambda$-calculus. However, one last task remains: we need to show that our definitions adhere to all of the monad laws. Luckily, it is very easy to do this using equational reasoning which we gave an introduction to previously. The proofs for the two identity laws are thus trivial as they work by rewriting one side of the equation and induction on $m$, respectively.

Proving that the associativity law holds also works by induction, but is slightly more challenging. To demonstrate how this works and to show that our $Expr$ monad does indeed obey the monad laws, we will give the proof below. As it turns out, the proofs for various other monads have a similar structure so that understanding this proof will enable us to understand the others as well. Inductive proofs always begin with the base case(s) which, in this case, is just $Var\ a$:

\[
(Var\ a \gg= f) \gg= g \\
= \{ \text{applying } \gg= \} f\ a \gg= g \\
= \{ \eta\text{-expansion} \} (\lambda x \rightarrow f\ x \gg= g)\ a \\
= \{ \text{unapplying } \gg= \} Var\ a \gg= (\lambda x \rightarrow f\ x \gg= g)
\]

This was simple, although the $\eta$-expansion step may not be immediately obvious. We now have our induction hypothesis and can quickly proceed with the inductive cases:

\[
(Abs\ a\ e \gg= f) \gg= g \\
= \{ \text{applying } \gg= \} Abs\ a\ (e \gg= f) \gg= g \\
= \{ \text{applying } \gg= \} Abs\ a\ ((e \gg= f) \gg= g)
\]
We omit the final case for \( App e_1 \ e_2 \) as it is mostly the same as for \( Abs \ a \ e \), except that the induction hypothesis needs to be applied twice. However, we shown how easy it is to reason about pure functional programs to prove properties of them.

### 2.3.2 The State monad

As hinted at previously, we are particularly interested in the state monad whose effect is mutable state. We know that pure functions have no side-effects and therefore cannot manipulate mutable variables. While this could easily be done in imperative programs, state needs to be passed explicitly from function to function in a pure language. Let us consider a classic example, in which we wish to relabel a tree of natural numbers with increasing values from the left-most to the right-most leaf of the tree. We begin with a data type for such trees:

\[
\text{data } \text{Tree} = \text{Leaf} \text{Int} \mid \text{Node} \text{Tree} \text{Tree}
\]

In order to keep track of the last number we have assigned to a leaf, we need a counter which has to be increased every time a leaf is updated. Because we are working with a pure language and all values are immutable, we need to keep track of it explicitly:

\[
\text{label} :: \text{Tree} \to \text{Int} \to (\text{Tree} , \text{Int})
\]

\[
\text{label} \ (\text{Leaf} \ _) \ n = (\text{Leaf} \ n,n+1)
\]

\[
\text{label} \ (\text{Node} \ l \ r) \ n = \text{let} \ (l',n') = \text{label} \ l \ n \quad (r',n'') = \text{label} \ r \ n' \ \text{in} \quad (\text{Node} \ l' \ r',n'')
\]

Clearly this approach is rather complicated and even an experienced programmer could easily lose track of the right iteration of \( n \) to use. Obviously this is a simple example, showing that the problem will only become worse in more realistic scenarios.

However, after some thought we can identify a pattern here. Any function which wants to consume a state value needs to accept it as an argument before, optionally, returning an updated version of it in a pair, together with the main result of a computation. In other words, we have functions of form \( s \to (a,s) \) which we will refer to as state transformers, where \( s \) is the type of the state and \( a \) is the type of the main result returned by the function.
It would be good if we could get a monad to take care of this, but we need a type of kind \( \star \rightarrow \star \), while \( s \rightarrow (a, s) \) has kind \( \star \). However, we can define a new data type with a single constructor for this purpose:

\[
\text{newtype } \text{State } s \ a = S \{ \text{runState} :: s \rightarrow (a, s) \}
\]

Data types need to be parametrised over all type variables which appear on the right-hand side, and we therefore end up with a type of kind \( \star \rightarrow \star \rightarrow \star \). Luckily, we can just partially apply this type to only one universally quantified argument in the instance declaration for \textit{Monad}, to end up with a type of the right kind:

\[
\text{instance } \text{Monad } (\text{State } s) \ \text{where}
\]
\[
\text{return } v = S (\lambda s \rightarrow (v, s))
\]
\[
(S \ m) \gg f = S (\lambda s \rightarrow \text{let } (r, s') = m \ s \ \text{in } f \ r \ \text{in } m' \ s')
\]

The definition of \( \gg \) may seem confusing at first, but it is once again just a game with types. We are given two arguments whose types are \( \text{State } s \ a \) and \( a \rightarrow \text{State } s \ b \) which, if we pattern match on the first argument, gives us a function \( s \rightarrow (a, s) \). On the right-hand side we expect a value of type \( \text{State } s \ b \) and the only way to construct a value of type \( \text{State} \) is using the \( S \) constructor which, in turn, requires an argument of type \( s \rightarrow (b, s) \). We then use a \( \lambda \)-abstraction to define a new function, bringing a value of type \( s \) into scope.

Now we have set up the framework required for \( \gg \) to work, and can simply put the puzzle pieces together in the right order. We apply the function of type \( s \rightarrow (a, s) \) to the value of type \( s \), giving us access to its two results. The first result, of type \( a \), can be used as argument for \( f :: a \rightarrow \text{State } s \ b \). This yields a value of type \( \text{State } s \ b \) which lets us retrieve a function of type \( s \rightarrow (b,s) \). Finally, we apply this function to the second result of \( m \ s \), leaving us with the desired value of type \( (b,s) \). We can observe how this structure resembles that in our initial definition of the \textit{label} function.

Using this monad, we can now define \textit{label}' in a cleaner manner. Firstly, we define a helper function named \textit{fresh} which returns the current state and increments it by one for future use:

\[
\text{fresh} :: \text{State } \text{Int } \text{Int}\\
\text{fresh} = S (\lambda s \rightarrow (s,s + 1))
\]

The type of \textit{fresh} is \( \text{State } \text{Int } \text{Int} \), indicating that the type of the state is \( \text{Int} \) and that this state transformer also returns a value of the same type. Our new definition of the relabelling function has type \( \text{Tree} \rightarrow \text{State } \text{Int } \text{Tree} \) to reflect that it needs an additional argument of type \( \text{Tree} \) and also returns a tree:
\[
\begin{aligned}
\text{label'} & : \rightarrow \text{Tree} \rightarrow \text{State Int Tree} \\
\text{label'} (\text{Leaf } n) & = \text{do} \\
& \quad n \leftarrow \text{fresh} \\
& \quad \text{return} (\text{Leaf } n) \\
\text{label'} (\text{Node } l r) & = \text{do} \\
& \quad l' \leftarrow \text{label'} l \\
& \quad r' \leftarrow \text{label'} r \\
& \quad \text{return} (\text{Leaf } l' r')
\end{aligned}
\]

Here we make use of Haskell’s do-notation, which is just syntactic sugar for \(\gg=\) where, for example, the notion \(n \leftarrow \text{fresh}\) translates to \(\text{fresh} \gg= \lambda n \rightarrow \ldots\). Our new definition of \textit{label} does not reference the state any point and allows us to focus on the implementation of the relabelling algorithm itself.

### 2.3.3 Monad transformers

After the previous section, one may be very excited with the gained knowledge that monads can be used to hide ugly machinery with the aim of structuring code in such a way that one may focus on only the important aspects of it. However, once the initial euphoria has worn off and we start playing with monads for other effects, ranging from exceptions to non-determinism, we will inevitably run into the question of how multiple such effects may be used at once.

Sadly, it is not possible to just compose the effects of two or more arbitrary monads with each other. However, if we look back to the definitions of \textit{return} and \(\gg=\) we have seen so far, we can note that they are \textit{pure} functions. \textit{I.e.} those functions do not have any effects themselves. So what if we were to define a monad in terms of functions which already have an effect?

We say that monads which are defined in this way are \textit{monad transformers}, and they allow us to construct stacks of monads and thus effects. In order to define a monad transformer version of, for example, \textit{State}, we define a new data type with an additional parameter for the type of the underlying monad:

\[
\text{newtype StateT } s m a = ST \{ \text{runStateT} : s \rightarrow m (a, s) \}
\]

This type can then be made an instance of \textit{Monad} in a similar fashion as \textit{State}, except that we need to wrap the results of the functions we use as arguments for \textit{ST} into the \textit{m} type. However, the only thing we know about \textit{m} is that it is an instance of \textit{Monad}, such that there are \textit{return} and \(\gg=\) functions for it:
The definition of \( \texttt{return} \) is relatively simple, other than the use of the generic \( \texttt{return} \) function in the definition of \( \texttt{return} \) here. Equally, \( \Rightarrow \) makes use of the generic \( \Rightarrow \) function, although this is hidden in the do-notation. Everything else has remained the same as before.

If we use \( \texttt{StateT} \), we will get both the effect of mutable state as well as the effect(s) of the underlying monads. Of course \( m \) may be a monad transformer itself, such that we can stack as many effects as we like. Note that \( \texttt{State} \) can be defined in terms of \( \texttt{StateT} \) with the help of the identity monad:

\[
\textbf{newtype Identity} \; \texttt{a} \; = \; \texttt{Identity} \{ \text{runIdentity} :: \texttt{a} \}
\]

\[
\textbf{instance Monad} \; \texttt{Identity} \; \textbf{where}
\]

\[
\texttt{return} \; \; = \; \texttt{Identity}
\]

\[
(\texttt{Identity} \; x) \Rightarrow f \; = \; f \; x
\]

Now \( \texttt{State} \) can be defined as a simple type alias, which gives us the benefit that all helper functions such as \( \texttt{get}, \texttt{put}, \texttt{modify}, \texttt{etc.} \) only need to be defined for \( \texttt{StateT} \):

\[
\textbf{type State} \; s \; = \; \texttt{StateT} \; s \; \texttt{Identity}
\]
\section*{\lambda.3 Language design}

In the previous chapter we have learnt about many of the key ideas in functional programming and why they are great for software engineering, but now that we are familiar with them it makes sense to also look at some of the shortcomings. In particular, we will focus on \textit{mutable state}, how it is being supported by languages of other paradigms, and how we might be able to enable functional languages to achieve similar conveniences.

If we consider the Von Neumann architecture, which most modern computers are based on, we know that program execution requires a processing unit and memory to store code as well as data. Computation is then performed through the mutation of data in memory. \textit{I.e.} any data we store in the memory of a computer is \textit{mutable}, meaning that it can be changed by programs whenever they request to do so.

Imperative programming languages, such as C \cite{Kernighan1975}, are designed with this architecture in mind. Therefore, programs written in these languages perform their work through the storage and manipulation of values in memory. Object-oriented languages, such as C++ \cite{Stroustrup1997}, add tools which allow programmers to structure their data in more meaningful and reusable ways.

In practice, languages in these two paradigms have turned out to be very popular for many different types of applications. This makes sense, due to the lack of constraints they impose on programmers. Additionally, object-oriented languages have the benefit that classes are a nice way of modelling objects and their relationships, similarly to how we would describe them in the real world.

However, it is the lack of constraints which has led to many problems. Most of us will have experienced software which crashes, produces incorrect results or misbehaves in some other way. Indeed, these two things are closely related, because it is difficult to reason about all possible configurations a computer may be in when we run a program. This makes it difficult to verify the correctness of such programs, as there are often too many cases to consider. Instead, software engineers depend on testing to see if their programs work roughly as expected.
In some ways, this approach may be seen as the equivalent of getting an architect to design a house, then building it, and finally throwing bricks at it to test that it will not collapse. Of course, this is not what happens in reality, because engineers are able to use the properties of the materials they are using to calculate what their constructions are capable of.

We know that pure languages are not plagued by these problems as we can use equational reasoning to prove facts about our programs, before we unleash them on the world. At the same time, this imposes many more restrictions on how programs may be written. This frequently means that real world problems which are easy to solve in imperative languages are suddenly much more difficult to implement in a language such as e.g. Haskell.

Kiselyov and Lämmel (2005), among others, attempted to encode common features of object-oriented programming languages in Haskell. However, they focused too strongly on the replication of such features, rather than on finding functional approaches to tackling the same goals. This lead them to use impure functions throughout their implementation, thus defeating the point of using a pure language.

We wish to find a pure interpretation of objects, which remains compatible with existing libraries and proven techniques. In other words, we are more interested in an elegant approach to working with the effect of mutable state, than encoding object-oriented principles. However, we shall draw inspiration from such languages to understand how they work and what parallels there are to, for example, the state monad.

As an example, let us consider a class in C++ which implements a vector for two dimensional space, consisting of two coordinates $x$ and $y$ and a function which adds the $x$-coordinate of another vector to itself:

```cpp
class Point2D
{
    public:
        int x;
        int y;

    void add(Point2D* p)
    {
        this->x += p->x;
    }
};
```

The code above is roughly equivalent to the following definitions in C, ignoring any additional data required to support more advanced object-oriented features:
struct Point2D
{
    int x;
    int y;
};

void add(Point2D* this, Point2D* p)
{
    this->x += p->x;
}

As we can see, a class simply consists of some data, which can be represented using a data structure, and procedures with an implicit first argument containing a reference to an object’s data. Objects can thus be viewed as values whose type is a class. However, we should be able to replicate all this in Haskell. Let us begin with a data type to replace the data structure:

data Point2D = Point2D { x :: Int, y :: Int }

We use the record syntax here to get a Haskell compiler to automatically generate appropriate projections for us. Of course, values are now mutable, so that we need to construct new ones if we wish to update them. Maintaining the same naming conventions, we can define the add function as:

add :: Point2D -> Point2D -> Point2D
add this p = Point2D (x this + x p) (y this)

Of course, the type signature seems somewhat reminiscent of the state monad and we can convert our initial definition of add to one which makes use of the monad:

add :: Point2D -> State Point2D ()
add p = modify $ \\
    \this -> Point2D (x this + x p) (y this)

This definition more closely resembles the add method in the C++ version of our example, as the state is now implicit. Sadly, our Haskell version is not nearly as nice as the object-oriented one, because we have to manually construct a new value of type Point2D to update the state. However, it is considered a better practice to define getters and setters for instance variables in object-oriented languages:
class Point2D
{
private:
    int x;
    int y;
public:
    int getX()
    {
        return x;
    }

    void setX(int x)
    {
        this→x = x;
    }
};

One justification for this pattern is that getters and setters provide greater abstraction of the details of the implementation from the interface of a class. Indeed, they are so popular that there exists an abstraction for this pattern in, for example, C# (Hejlsberg et al., 2003) where one may simply write the following to add an instance variable to a class and to automatically generate getters/setters for it:

public int X
{
    get;
    set;
}

The language is then clever enough to work out whether the getter or setter is needed when X is referenced, depending on the context. It seems that this arrangement accomplishes what we are looking for: it hides the gory details of the implementation and it can be generated automatically. To see how it works using the state monad, consider the following definition of a getter for x using the gets function:

gets :: (s → a) → State s a
gets p = S (λs → (p s, s))

getX :: State Point2D Int
g getX = gets x
Indeed, all we need to automatically generate a getter for a field is the name of the corresponding projection function. Setters can be defined using \( \text{modify} \), similarly to our previous definition of \( \text{add} \):

\[
\begin{align*}
\text{set}\,X &: \text{Int} \to \text{State \, Point2D} () \\
\text{set}\,X \, v &= \text{modify} \, \lambda (\text{Point2D} \_ \, y) \to \text{Point2D} \, v \, y
\end{align*}
\]

All we need to know here is the number of parameters the constructor has and the index of the field we are trying to set. We can now give a new definition of \( \text{add} \) which makes use of these getters and setters:

\[
\begin{align*}
\text{add} &: \text{Point2D} \to \text{State \, Point2D} () \\
\text{add} \, p &= \text{do} \\
&\quad \, v \leftarrow \text{get}\,X \\
&\quad \text{set}\,X \, (v + x \, p)
\end{align*}
\]

Another important concept in object-oriented programming is that of \textit{inheritance} using which a class may be derived from another to extend it. For example, a class \textit{Child} which inherits from another class \textit{Parent} will contain all of its instance variables, methods, and so on. This is useful to model relationships between object, but also introduces subtype polymorphism (Cardelli, 1984). This is generally the preferred type of polymorphism in such languages, but is of interest to us in that we wish to model abstractions similarly.

### 3.1 Putting it all together

We have now gained an overview of some of the key concepts in object-oriented language programmers will interact with on a daily basis. In summary, classes provide an easy and reusable method to define data structures as well as operations which manipulate them. Let us now attempt to replicate this idea in a Haskell-like language without side-effects, but with the aim of removing all references to the state monad.

Classes can be thought of as syntactic sugar for data structures and procedures which operate on them. Since we know that their basic functionality can be emulated using the state monad, we can propose the following syntax, inspired by decelerations for algebraic data types, which allows a programmer to declare a state type:

\[
\text{state} \, T \, \alpha_0 \ldots \alpha_n \quad n \geq 0 \\
\{ \\
\quad x_0 :: \tau_0 \quad m \geq 0 \\
\quad \ldots \\
\quad x_m :: \tau_m \\
\}
\]
The intuition here is that a compiler will generate a new ADT $T_{\text{Data}}$ with a single constructor $K$ whose parameters are given by $\tau_0$ to $\tau_m$:

$$\text{data } T_{\text{Data}} \alpha_0 \ldots \alpha_n = K \tau_0 \ldots \tau_m$$

This in itself is little more than a different notation for a data type, but this is changed by the addition of a type alias for an instance of the state monad, whose state is a value of type $T_{\text{Data}}$:

$$\text{type } T \alpha_0 \ldots \alpha_n = \text{State} (T_{\text{Data}} \alpha_0 \ldots \alpha_n)$$

Additionally, we have highlighted the importance of getters and setters in object-oriented languages. Therefore a compiler will generate such, as well as a projection function, for every typing $x_i :: \tau_i$ in a state declaration:

$$x_i :: T_{\text{Data}} \alpha_0 \ldots \alpha_n \rightarrow \tau_i$$

$$x_i (K y_0 \ldots y_m) = y_i$$

$$x_i.\text{get} :: T \tau_i$$

$$x_i.\text{get} = \text{gets} x_i$$

$$x_i.\text{set} :: \tau_i \rightarrow T ()$$

$$x_i.\text{set} v = \text{modify} (\lambda (K y_0 \ldots y_m) \rightarrow K [y_i \mapsto v](y_0 \ldots y_m))$$

All the basic building blocks we need to work with state can now be generated for us and one may use them to write programs. However, let us note that this notation and its translation is not very flexible yet as it does not allow us to make use of the state monad transformer. To resolve this shortcoming, we will once again draw inspiration from C++ to make a small addition to the syntax:

$$\text{state } T \alpha_0 \ldots \alpha_n [: T_p \tau'_0 \ldots \tau'_k] \quad n \geq 0$$

$$\{$$

$$\quad x_0 :: \tau_0 \quad m \geq 0$$

$$\quad \ldots$$

$$\quad x_m :: \tau_m$$

$$\}$$

If the optional addition to the header of the state declaration is present, our translation for the type alias $T$ changes to produce:

$$\text{type } T \alpha_0 \ldots \alpha_n = \text{StateT} (T_{\text{Data}} \alpha_0 \ldots \alpha_n) (T_p \tau'_0 \ldots \tau'_k)$$

In our interpretation, inheritance now corresponds to stacks of effects. We assume that $T_p$ is a monad such that, for instance, a state declaration whose parent is $\text{IO}$ will
possess the underlying effect of impure computation. Equally, if \( T_p \) is another instance of the state monad then one may access its state using \( \text{lift} \) which corresponds to the \textit{parent} or \textit{base} keywords in languages like Java and C#.

Lastly, we note that the getters and setters generated in the translation process are never explicitly named in the syntax. It may be unintuitive for programmers to reference functions which they have not named. Therefore, we will also design new types of statements which can be used to reference them using the name of the field they manipulate. These may be used in a setting like Haskell’s do-notation:

\[
p \leftarrow x_i \rightsquigarrow x_i\text{get} \Rightarrow \lambda p \rightarrow \ldots \quad \text{Getter statement}
\]

\[
e \rightarrow x_i \rightsquigarrow x_i\text{set} e \left[\Rightarrow \lambda _\rightarrow \ldots\right] \quad \text{Setter statement}
\]

We assume \( p \) to be a single pattern and \( e \) to be an expression. The translations for both are shown after the \( \rightsquigarrow \) and are noteworthy in that they use the name of the field to reference an appropriate getter or setter. Note that setters may appear at the end of a block of statements, in addition to expressions in e.g. the do-notation.

### 3.2 Related work

Now that we have been able to identify a suitable syntax for our goals and the translation rules required to implement it, we need to think about how this may be implemented. We could design a new language based on our ideas or add them to an existing one. Any open source compiler can of course be modified, but the size and complexity of the source code often makes this a challenging option.

Haskell is a popular platform for research into programming languages, and efforts have been made to simplify the process in which extensions can be added to popular implementations of the language. The Glasgow Haskell Compiler provides various facilities for this purpose, ranging from support for plugins to Template Haskell (Sheard and Peyton Jones, 2002), which can be used to generate code from meta-languages at compile-time. However, only the generated code is type checked by the compiler\(^1\) and error messages which may arise could therefore be uninformative.

Erdweg et al. (2012) designed and implemented SugarHaskell, a tool to apply syntax extensions, defined as regular Haskell modules, to source files to convert them into ordinary Haskell code. While this would make it very easy to implement our abstractions on top of Haskell, it would lead to very confusing error messages as the Haskell compiler is not aware of any transformations which took place before it was invoked.

A reasonably popular library for working with state in Haskell is \textit{lens}\(^2\), which uses

\(^1\)The author notes that changes to Template Haskell were implemented in GHC in May 2013 which change this behaviour. \url{http://www.haskell.org/pipermail/ghc-devs/2013-May/001266.html}

\(^2\)\url{http://hackage.haskell.org/package/lens}
A compiler for Cada

Template Haskell to automatically generate getters and setters for records. It is similar to our designs in this way, but also differs greatly in that the Haskell compiler has no additional knowledge of what the authors of the library refer to as lenses:

\[
\text{data} \: \text{Lens} \: a \: b \: = \: \text{Lens} \{ \text{getter} :: a \rightarrow b, \text{setter} :: b \rightarrow a \rightarrow a \}
\]

This means that programs which use this library need to keep track of the relevant information at runtime, decreasing performance through the need of having to unbox values of type \( \text{Lens} \) whenever a getter or setter is needed. Our design does not suffer from this issue as our compiler will be aware of the names of the relevant functions and can reference them when required by the program. This also makes our approach more suitable for optimisations and better error messages.

Additionally, lenses work on data types, such that they cannot be used to automatically generate type functions for configurations of monad stacks, such that users still need to be aware of the mechanics of monad transformers and the state monad.
One may argue that it is easy to write code for a compiler, but whether or not it will work as one expects depends on the thorough specification of the language which is being implemented. It would be unfortunate if we were to spend months on the implementation of the various components of a compiler, only to discover that, for example, some terms in our language cannot be typed.

In this chapter, our aim will be to give a specification of the Cada language, beginning with the theoretical representation of its type system (section 4.3), followed by the specification of its concrete syntax (section 4.5). Additionally, we will consider more practical aspects of the language design, such as the module system (section 4.2) and the standard library (section 4.7).

4.1 Summary

Before we dive into the details of the language specification, however, let us note that Cada is designed after Haskell98 (Peyton Jones, 2002) and thus shares many similarities. To make it easier for readers with previous knowledge of Haskell or similar functional programming languages to understand the details of this specification, we will briefly summarise the key differences below.

Many of the changes exist in the concrete syntax where we lose Haskell’s layout rules and favour a more C-like syntax instead. The justification for this is mainly one of flexibility and simplicity, since the additional complexity introduced by such a system does not help us to accomplish the goals we set out for this project. Of course we also have the constructs from the previous chapter which are unique to Cada.

Some of the syntactic sugar which can be found in Haskell is also omitted, such as guards, list comprehensions and sections. While these features provide convenience to the users of the language, they are neither interesting to implement nor would they contribute significant advantages over equivalent notations.

Lastly, we do not allow users to declare operator fixities and precedences. Indeed all operators, including those that are built-in, share the same precedence.
4.2 Module system

In order to avoid having to repeatedly write the same code or to copy and paste existing code into a new project, Cada programs consist of modules. Each source file corresponds to one module and may be compiled separately from the rest of a program. Upon successful compilation of a module, an interface file is generated for it in addition to an object file. A module may import other modules, subject to them having existing interface files.

An interface file contains all information relevant for the type system to make use of the corresponding module’s declarations. This includes the typings of value definitions, algebraic data types, type functions, type classes, and class instances. The details of all these structures are discussed in section 4.3.

The name of a module should match that of the source file which contains it. For example, a file Program.cada should be home to a module named Program. Hierarchical names are supported, such that Cada.Class.Eq corresponds to a file at path \Cada\Class\Eq.cada. When a module is imported, its name is converted to a file path using this pattern, except that the extension for interface files, .co, will be used.

Modules automatically export everything defined in them. For simplicity, there are no means by which the list of exports can be restricted. However, it would be easy to add such facilities in the future.

4.3 Type system

Cada’s type system is the same as that of Haskell98. At its heart lies the system described in subsection 2.2.2, with the addition of type classes (Hall et al., 1996) and type functions. Other notable extensions which are not part of Hindley-Milner include kinds to describe types (subsection 4.3.1) and algebraic data types (Burstall et al., 1980) to let users define their own types. Figure 4.1 gives a formal specification of the syntax of the type system, where we use e.g. \(\overline{C}a\) to mean zero or more recurrences of \(C\) \(a\) for different \(C\) and \(a\).

4.3.1 Kinds

All types in Cada are assigned kinds which can be seen as types of types. Their primary purpose is to describe the parameters of, for instance, type functions in more detail than an arity would. There are two ways of constructing kinds:

\[
\begin{align*}
\kappa & \rightarrow \ast & \text{Type} \\
\mid \kappa & \rightarrow \kappa & \text{Kind function}
\end{align*}
\]
Types with no parameters such as *Char* or *Int* have kind $\star$ which we can write more succinctly as, in this case, *Char* : $\star$. It denotes that these types do not need to be applied to any arguments. If we have a type function, on the other hand, it needs to be applied to other types before it becomes a type itself and its kind will reflect this. Like regular functions, they are curried and applying a type argument will either result in a type whose kind is $\star$ if all arguments have been applied or another type function which will accept the next argument if there is one.
As an example of a type with a more interesting kind, let us consider the following definition for a type function which takes two arguments \( f \) and \( a \), applies the second argument to the first and then gives the result as an argument to \( f \):

\[
\text{newtype } \text{Rec} A f a = \text{Rec} A (f \ (a \ f))
\]

What should the kind of \( \text{Rec} A \) be? We know that this type has two arguments, so that we will have at least \( \kappa_f \rightarrow \kappa_a \rightarrow \ast \) where \( f : \kappa_f \) and \( a : \kappa_a \). We can observe that \( f \) and \( a \) are both applied to arguments as well. Their kinds must therefore be functions and cannot just be \( \ast \). Working from the bottom up, \( a \) is applied to \( f \) so that its kind must be \( \kappa_f \rightarrow \ast \). \( f \) is then applied to the result of \( a \ f \), a type of kind \( \ast \), giving us \( f : \ast \rightarrow \ast \). We can now substitute \( \kappa_f \) for this kind, yielding \( a : (\ast \rightarrow \ast) \rightarrow \ast \) and finally \( \text{Rec} A : (\ast \rightarrow \ast) \rightarrow (((\ast \rightarrow \ast) \rightarrow \ast) \rightarrow \ast) \rightarrow \ast \).

### 4.3.2 Kind inference

We have shown informally how the kind of a type may be inferred from its declaration in the section above. It becomes apparent that this process is very similar to HM-style type inference. Indeed, we will later use the same algorithm to infer types and kinds. In order to do this, types will need to be translated to expressions and kinds must be represented as types, a process which we will call type demotion.

Firstly, we need find types which correspond to kinds. These types will be inferred from the expressions we generate from type declarations and ultimately represent the kinds of those types. For clarity we will use \( \rightarrow \kappa \) to denote type arrows and \( \rightarrow \tau \) to denote functions. We use \( \rightsquigarrow \) to denote the demotion relation:

\[
\begin{align*}
\ast & \rightsquigarrow \text{Star : } \ast & \text{Star type} \\
\rightarrow \kappa & \rightsquigarrow \text{KFun : } \ast & \rightarrow \kappa \ast & \rightarrow \kappa \ast & \text{Kind function type}
\end{align*}
\]

Data type declarations, type functions, type classes, and instances can influence the kind of a type. The conversion process for all constructs follows roughly the same scheme: each type becomes an equation, its type variables are turned into variables bound by \( \lambda \)-abstractions, and types are translated into equivalent expressions.

For example, suppose we are given a surface definition of a data type \( \text{List } a \) with two constructors \( \text{Cons } a (\text{List } a) \) and \( \text{Empty} \), then we will construct the following equation to infer the kind of \( \text{List} \):

\[
\text{List} = \lambda a. (a \rightarrow \ulcorner \text{List } a \rightarrow \ulcorner \text{List } a \ulcorner) \uplus (\text{List } a)
\]

Note that \( \rightarrow \ulcorner \) is simply a right-associative function written in infix notation whose type is \( \text{Star} \rightarrow \tau \text{Star} \rightarrow \tau \text{Star} \). Additionally, we need to combine the constructors of the algebraic data type in one expression to ensure that all occurrences of the type
parameters are considered in the same context. For this purpose, we assume that there exists an operator with the following typing:

\[ \oplus :: \text{Star} \rightarrow \tau \text{ Star} \rightarrow \tau \text{ Star} \]

Our choice of notation is due to the type theoretic interpretation of algebraic data types. We also need to combine the typings of methods in a type class, but we cannot use \( \oplus \) because it is too strict about its arguments. The types of methods may introduce type variables which are not declared in the class header and the corresponding quantifiers will thus be translated into abstractions. This means that we may end up with type functions instead of just \( \text{Star} \):

\[ \Diamond :: \forall \alpha : \star. \forall \beta : \star. \alpha \to \beta \to \text{Star} \]

For example, the translation of the \( \text{Monad} \) type class produces the following equation:

\[
\text{Monad} = \lambda m. \ (\lambda a. a \to m a) \Diamond \\
(\lambda a. \lambda b. m a \to (a \to m b) \to m b)
\]

### 4.4 Lexical grammar

In order to translate characters from a source file into terminals for the parser, we have to define the lexical structure of Cada programs which we will later use to guide the implementation of a lexical analyser. We will use the following notational conventions throughout this specification:

- \( \text{pattern} \) non-terminals
- \( \text{symbol} \) terminals (ASCII symbols)
- \( \text{pattern}^* \) zero or more occurrences of \( \text{pattern} \)
- \( \text{pattern}^+ \) one or more occurrences of \( \text{pattern} \)
- \( \text{pat}_0 \mid \text{pat}_1 \) choice between \( \text{pat}_0 \) and \( \text{pat}_1 \)
- \( \text{pat}_0 \rightarrow \text{pat}_1 \) \( \text{pat}_0 \) but not \( \text{pat}_1 \)
- \( (\text{pattern}) \) grouping
- \([\text{pattern}]\) optional

#### 4.4.1 Program structure

The overall lexical structure of Cada programs is shown in Figure A.6. Like most languages, Cada supports single-line comments as well as multi-line comments. Both are modelled after Haskell such that single-line comments begin with \(--\) (two dashes) and run until the end of a line. However, if \(--\) is part of a valid token, such as a string literal or operator, then it will not start a comment.
Multi-line comments may, as the name suggests, span across multiple lines, begin with { -, and end with the matching - }. In most languages, multi-line comments only run until the next closing tag is found, but we allow multi-line comments to be nested such that we need a matching number of opening and closing tags. This is convenient for programmers if they wish to wrap a block of code into a comment which already contains one or more multi-line comments.

### 4.4.2 Identifiers

Identifiers in Cada are constructed with the help of the non-terminals shown in Figure A.7 and Figure A.8. All sequences of lower-case and upper-case characters, digits, and prime symbols may be used for two flavours of identifiers, classed as variables and constructors. The former must start with a lower-case character and may not be the same as the reserved keywords in the rid non-terminal. Constructors must start with an upper-case character.

All identifiers may be qualified if they are preceded by the hierarchical name of a module. Unlike in Haskell, variables may also contain dots in their names for purposes other than qualification. Note that this requires the composition operator to be surrounded by whitespaces. Operators consist of sequences of one or more symbols.

### 4.4.3 Literals

Literals come, initially, in two varieties (Figure A.9): those for strings and those for positive integers. Other types, such as list literals or tuples, will later be constructed by the parser. Integers are simply sequences of one or more digits, while strings consist of zero or more graphical symbols surrounded by quotation marks.

### 4.5 Context-free grammar

In the previous section we described how the text contained in source files may be translated into terminal symbols. We shall now use these to describe the structure of Cada programs. All of the context-free grammar is provided for reference in appendix A.4. Note that we use Symbol* to mean zero or more repetitions of Symbol and ε to represent the empty word.

It is worth remembering that while the CFG specifies the grammar of our language, it may not be represented identically in our implementation later because we need to consider the structure of the abstract representation as well as limitations of the chosen parsing technique. However, we have disambiguated the grammar presented here and included productions which are more “generous” than they should be. We will return to this part of the compiler in more detail in section 5.4.
Cada’s concrete syntax is very similar to that of Haskell minus the features we listed in section 4.1. Notable differences between the two languages will be outlined in the following sections. However we note that the key difference is the lack of a layout rule, meaning that semicolons and curly braces are always explicit in Cada. Additionally we have the syntax described in chapter 3 which we have discussed in depth there.

4.5.1 Algebraic data types

The definition of new types is a key aspect of functional programming and thus we included several different notations for algebraic data types in Cada (Figure A.18): enumerations, newtypes, single constructor types, and complex data types. The former two will be more familiar to those with backgrounds in imperative or object-oriented programming languages.

Enumerations are isomorphic to the natural numbers and can therefore be represented efficiently on stock hardware using \textit{e.g.} words. Their syntax is inspired by C# and represents an algebraic data type with constructors which have no arguments.

Newtypes are used to define new types with a single constructor which has exactly one parameter. This may seem silly at first, because we could just use the type of the parameter, but using this method one may introduce additional type safety over type functions through the need of having to wrap/unwrap values. However, a compiler may optimise this wrapper away after type checking, thus removing the introduced performance overhead.

Single constructor data types resemble structures in imperative languages in that they simply consist of a group of fields. The name of the constructor will be the same as that of the type. This notation primarily serves for convenience as data types which only have one constructor are reasonably common.

Lastly, complex data types may have one or more constructors with zero or more fields each. Constructor with no parameters may be declared without curly braces as long as they are followed by commas.

4.5.2 Type classes and instances

Unlike in Haskell, one may specify the type of a method in an instance declaration although this serves no practical purpose. A compiler may compare the explicit typing with the inferred type in the same fashion as for top-level functions. This addition aims to provide the same benefits which are provided by other explicit typings, allowing programmers to better plan and document their code.
On the other hand, type classes may not contain default methods. The reasoning for this is largely arbitrary, but can be justified as this feature merely exists for convenience and is not vital for the construction of programs. It would be easy to add it in a future iteration of the language, however.

### 4.5.3 Expressions

Expressions are the key building blocks of functional languages and this is of course no different in Cada where they are more advanced than in the \(\lambda\)-calculus but simpler than in Haskell. There are two noteworthy changes however.

Firstly, there is no \texttt{do} keyword in our language. Instead, “do”-blocks are denoted simply using curly braces. This more closely resembles imperative languages and may be more accessible for beginners since they do not have to worry about what the keyword means until they are more familiar with the language.

Secondly, let-expressions may not contain explicit typings. Once again, this choice removes a rarely used feature in favour of simplicity. It would be simple to add this functionality to the language in a future version as it could share code with top-level bindings and typings.

### 4.6 Denotational and operational semantics

Ideally, the specification of programming languages should include formal descriptions of their denotational and operational semantics. However, it is notoriously difficult to give a full specification of such for real programming languages, although some work has been done in the area such as in Peyton Jones and Wadler (1992) and more recently Faxén (2002).

Unlike languages such as, for example, Agda (Norell, 2008) which are total (Turner, 2004), functions in Cada are partial. For example, a function which we would describe using \(\texttt{Bool} \rightarrow \texttt{Bool}\) does not need to map every element of \texttt{Bool} to another value of type \texttt{Bool}. The need for this arises from non-termination (see 2.1.1) which may cause a function to map to no value for some argument. We justify this semantically through the inclusion of \(\perp\) in all basic data types.

Operationally, Cada is a non-strict language like Haskell. In strict languages, the arguments of functions must be evaluated before they can be called, but this is not the case here. We only evaluate terms when their values are needed and enable sharing of the results between different call sites. We therefore speak of \textit{call-by-need} or \textit{lazy evaluation}. Advantages of this technique allow potential performance gains because a value which is never needed will not be computed. Additionally, we can easily represent infinite data structures such as streams.
4.7 Standard library

Cada’s standard library provides commonly used functions and types which aid with the construction of programs. Most importantly, it exports types and functions which cannot be represented in Cada itself because they, for example, violate syntactic conventions, create loopholes in the type system, or are required for the language to function.

All such primitives must be wired into the compiler and there must exist a way to import them into other modules. An implementation is expected to expose at least the following list of types: `Char`, `Int`, `[]`, `()`, `(,)` and `IO`. We require the following impure functions whose behaviour should be the same as that of their counterparts in Haskell: `return` and `<$>` for `IO`, `putStrLn`, `getLine`, and `error`. Additionally, all functions required to make `Int` and `Char` instances of `Eq` and `Show`, as well as to make `Int` an instance of `Num`, should be provided by the compiler.

The standard library itself should be split into modules where one exists for each data type or type class. For the purpose of this project, the following modules ship with the compiler, in order of dependency:

- Cada.Either
- Cada.Class.Functor
- Cada.Class.Collection
- Cada.Bool
- Cada.Class.Eq
- Cada.Class.Num
- Cada.Class.Ord
- Cada
- Cada.Class.Monad
- Cada.Class.MonadTrans
- Cada.Maybe
- CadaMonad.Identity
- Cada.Class.Monoid
• Cada.Class.Show
• Cada.IO
• Cada.List
• Cada.Pair
• Cada.Monad.StateT
• Cada.Monad.WriterT

All of these modules should be brought into the scope of new programs automatically, with the exceptions of Cada.Class.Monoid and Cada.Monad.WriterT.
Even though we now have a formal specification for our language, there are no requirements regarding its implementation. While this may be interpreted as a shortcoming of the specification, this approach to the construction of programming languages is actually quite common due to the advantages it entails with regards to the diversity of implementations of the same language. If we were to mix language and compiler design, then we might run into situations where parts of the language depend on specific decisions we made for the implementation. This is of course undesirable because it might prevent the development of more advanced or efficient compilers for such a language.

Thus our task in this chapter is to answer the questions which arise during the implementation of a compiler for Cada. Many of the topics we will discuss here are also transferable to the implementations of other functional languages where some parts currently lack coverage in literature.

Our implementation language will almost exclusively be Haskell, apart from the majority of the standard library which will be implemented in Cada itself. This choice makes sense for a number of reasons, ranging from our existing familiarity with the language to its modern set of features, such as pattern matching which will enable us to write more elegant and maintainable code. For simplicity and portability, we will mostly be using the parts of the language which are laid out in its standard, with few of the extensions provided by popular implementations like GHC.

5.1 Architecture

In order to understand how the next few sections fit into the overall picture, let us first consider the high-level architecture of our compiler. Figure 5.1 gives an overview of the key steps required to compile each input file. While we will explore each stage in detail later on, let us briefly summarise what happens in the time from first invoking the compiler to receiving a binary.

As soon as the compiler is run, it will parse the command line arguments to determine what it is supposed to do (5.2). Once we know which files we are meant to compile
and in what way, we generate an abstract syntax tree for each of them using the parser (5.3 and 5.4). This is required so that we can extract the list of module imports from each AST and check that the dependencies do not form a circle. Next, we continue with a single module and set out on the road to type inference. The first stop on this journey involves demoting types to expressions so that we can infer their kinds using the same system we will use later on to infer the types of expressions (5.9). With this gained knowledge, we then move on to sort equations and explicit typings into binding groups which will later dictate the order in which types are inferred (section 5.11). After all this is done, we have performed all required compile-time checks for the input program. The last step for each source file is thus the generation of code in the target language, which will choose to be Haskell (5.12). If everything goes well, we will end up with one interface (.co) and one object file (.o) for each input file. Unless the compiler is told otherwise, it will then link all object files together into an executable.

5.1.1 Directory layout

There are six subfolders in the root source directory for the compiler, each of which contains the Haskell source files for one subsystem. The root directory itself only contains a handful of files which contain the main entry point as well as testing and debugging facilities. A description of each subfolder is given below:

- \Cada contains the front end for the compiler, including data types for the abstract syntax tree, the lexer, and the parser.
- `\CodeGen` provides the compiler’s back end. Specifically, functions to generate Haskell code from the abstract syntax tree and type environments are located here.

- `\Compiler` is comprised of modules which tie together all the other subsystems and manage the different compilation stages.

- `\Internal` houses type environments for Cada modules which are wired into the compiler, such as `NCC.Internal`.

- `\TypeSystem` accommodates the entirety of the type system, its data types, and the type inference algorithm.

- `\Utility` is home to miscellaneous modules which are used throughout the compiler, but do not fit into any of the other categories.

Haskell modules are named after the directories which they are located in, such that e.g. the names of all modules in the `\TypeSystem` folder start with `TypeSystem.*` where `*` corresponds to the name of a source file.

## 5.2 Configuration

Many solutions for parsing command line arguments exist for Haskell, but one which is particularly nice is the one written by Paolo Capriotti\(^1\). This library allows us to define a simple and elegant parser in the applicative style (McBride and Paterson, 2008). We begin with a data type for the compiler configuration:

\[
\text{data Config} = \text{Cfg} \{ \\
\text{cfgInputs} :: [\text{String}], \\
\text{cfgOutput} :: \text{String}, \\
\text{cfgEntry} :: \text{String}, \\
\text{cfgPaths} :: [\text{String}], \\
\text{cfgUseStdout} :: \text{Bool}, \\
\text{cfgTrace} :: \text{Bool}, \\
\text{cfgNoLinking} :: \text{Bool}, \\
\text{cfgStopAfterParser} :: \text{Bool}, \\
\text{cfgKeepHs} :: \text{Bool}, \\
\text{cfgNoImplicitPrelude} :: \text{Bool} \\
\}
\]

\(^1\)https://github.com/pcapriotti/optparse-applicative
Specifying the parser is now as simple as using a handful of combinators to configure how each parameter of the \texttt{Cfg} constructor should be parsed. The code required to parse the first two options is given below:

\[\text{parseCfg} :: \text{Parser Config} \]
\[\text{parseCfg} = \text{Cfg} \]
\[<\$> \text{ arguments str (metavar "FILE(s)")} \]
\[<\ast> \text{ strOption (} \]
\[\quad \text{ long "output" } <> \]
\[\quad \text{ short 'o' } <> \]
\[\quad \text{ metavar "FILE" } <> \]
\[\quad \text{ value "a.exe" } <> \]
\[\quad \text{ help "Name of the output file"} \]
\[\text{)} \]
\[\ldots\]

Explanations of what each combinator does may be found in the library’s documentation and the full code for the parser may be found in the \texttt{Compiler.Configuration} module. We summarise the meaning of each command line option below, where we use upper-case names to represent metavariables and square brackets to indicate optional components:

- \texttt{FILE} [...] is used to specify the paths of the input files. There must be at least one source file.

- \texttt{--output FILE or -o FILE} is used to specify the name of the output executable. This option has no effect if \texttt{--no-link} is active.

- \texttt{--entry MODULE or -e MODULE} is used to specify the name of the main module. This option has no effect if \texttt{--no-link} is active.

- \texttt{--path PATH or -p PATH} is used to specify a path in which the compiler should look for interface files. This option may be used more than once.

- \texttt{--use-stdout} causes the compiler to redirect trace messages to the standard output. This option has no effect unless \texttt{--trace} is active.

- \texttt{--trace} causes the compiler to emit diagnostic messages to \texttt{MCC.log}, unless \texttt{--use-stdout} is active, in which case the messages will be sent to the standard output together with regular compiler messages.

- \texttt{--no-link} skips the linking stage. This is useful for scenarios where no executable is desired or no entry point exists, such as in the standard library.
• **--stop-after-parser** stops compilation after all input files have been parsed. This is useful to verify that there are no syntax errors in a source file.

• **--keep-hs** causes the compiler to keep all intermediate Haskell source files, which would normally be deleted.

• **--no-implicit-prelude** will cause the compiler to not implicitly import the standard library, which is useful for compiling the standard library itself.

### 5.3 Lexical analysis

Traditionally, the first phase of a compiler is concerned with the lexical analysis of the inputs. That is, given a string of characters from a source file, we wish to generate a list of input tokens which correspond to the terminal symbols of the context-free grammar. These tokens are then used in the next stage by the parser to construct the abstract syntax tree. In our compiler, however, these two processes will actually be combined into one where the parser calls the lexical analyser whenever it requires more tokens. We will refer to this style of lexical analysis as **threaded lexing** and the actual component of our compiler is thus a **threaded lexer**. There are three popular approaches for writing lexers in Haskell:

1. Using tools such as Alex\(^2\) it is possible to automatically generate code for a lexical analyser from a grammar specification. This approach would make it very easy to translate the formal specification we have for the lexical syntax into a format which is understood by such a tool. It also means that we do not have to worry about the details of the implementation and that we can easily make changes or additions to the syntax.

2. The Parsec library (Leijen and Meijer, 2001) is one of various packages which aid with the manual construction of parsers. Of course a lexer is just a simple parser, and thus such libraries may be helpful here. Handwritten parsers may be more efficient than those generated by tools, but are also more prone to errors as they cannot be validated as easily. We will return to Parsec in more detail once we are looking at the implementation of our parser.

3. Lastly, we could write the lexical analyser by hand without the help of any tools or libraries. This may justifiable given the relative simplicity of the lexical syntax, but it would also make this task unnecessarily more difficult and decrease the maintainability of our code.

\(^2\)http://www.haskell.org/alex/
We chose to use Alex for this task because there are no significant disadvantages to it, other than maybe performance which we are not too worried about here. Indeed, the Glasgow Haskell Compiler also uses Alex for its lexer. The majority of the specification, found in Lexer.hs, is simply a translation of the lexical grammar to the format required by Alex and does therefore not require much discussion. We will look at the other parts in more detail in the subsequent sections.

5.3.1 Tokens

Before we can specify the grammar for Alex, we will need to define the data types it depends on. Most importantly, we require a representation for input tokens. One approach for this might be to define a data type which enumerates all terminal symbols, except those which come in infinitely many forms such as, for example, variables and literals. In our implementation, located in the Cada.Token module, there are two data types for this purpose. Firstly, we have one type named TokenT which enumerates different groups of tokens:

```haskell
data TokenT = TEof
             | TVar
             | TCtr
             | TVarSym
             | TRes
             | TRop
             | TSpecial
             | TInt
             | TStr
             deriving (Eq, Show)
```

Secondly, Token combines a value of the above type with a String value which we will use to store additional information about the token such as the name of a keyword or the value of a literal:

```haskell
data Token = T {
  tType :: TokenT,
  tVal :: String
}
```

This design allows us to make changes to the lexical syntax more easily, because we will only have to add extra productions to the specification for Alex, rather than in addition to a corresponding constructor for some data type.

5.3.2 Multi-line comments

Like Haskell, we allow multi-line comments to be nested within each other. At first, this may seem like a rather trivial thing to do, but upon inspection we will find
that it is actually interesting both in terms of an implementation as well as from a theoretical point of view. If it was not for this feature, each production in our lexical grammar would be regular. In other words, a deterministic finite automaton would be able to decide whether an input is accepted by the language corresponding to each production. This is good, because a DFA only needs to look at each input symbol once and therefore runs in $\Theta(n)$ time where $n$ is the size of the input.

However, in order to keep track of how deeply nested the multi-line comments are, we need a counter. This cannot be accomplished with a DFA, but requires at least a pushdown automaton. Alex specifications are based on regular expressions which, like DFAs, represent regular languages. Therefore it is impossible to define appropriate rules for nested multi-line comments. Luckily, we can get around this dilemma by manually writing a function to parse multi-line comments which counts how deeply nested the current comment is and returns control to Alex as soon as that value reaches zero.

The actual implementation of this feature is neither particularly pretty nor interesting, as it simply involves converting bytes read by Alex to values of type `Char` and tests for various cases. This whole procedure is contained in the `Cada.Lexer` module where Alex invokes the semantic action `mlComment` if the start of a multi-line comment is encountered. Most of the work is done then by `recComments`, which is invoked straight away with an argument of 1 to indicate the current comment depth. Note that it only needs to look at each input character once until it reaches the end of the multi-line comment at depth 1. Thus it still runs in linear time, even though it could not be implemented using a DFA.

### 5.4 Parser

Using the lexer we have constructed in the previous section, we can now build the parser. Its task is to take the tokens to construct an abstract syntax tree using the productions of the context-free grammar. As we have discussed previously, this process will be threaded such that the parser dynamically requests more tokens from the lexer when they are needed. Once again, we will consider three different approaches to accomplishing this task:

1. **Happy**\(^3\) is a tool similar to Alex, which takes a grammar specification and turns it into code for a LARL(1) parser (DeRemer, 1969). The translation from Cada’s context-free grammar to such a specification would be easy and because we do not need to worry about the details of the implementation, changes can easily be made.

---

\(^3\)http://www.haskell.org/happy/
2. We have previously considered Parsec for the construction of the lexer, but it also allows us to develop recursive descent parsers for much more complicated languages. It would require us to write all code by hand, but the resulting parser could exceed the capabilities of those generated by Happy at the cost of being less computationally efficient.

3. It would be possible to write a parser without the help of any libraries using, for example, pattern matching on sequences of tokens. However, this approach could easily lead to an ugly and unmaintainable codebase, unless we spend a significant amount of time on abstractions and helper functions.

For evaluation, we have prototyped parsers using all three approaches, leading us to conclude that the first two methods are both equally suitable for this task. However, in the end we decided to settle with Happy, because of its flexibility to accommodate changes to the syntax and the simplicity with which those can be implemented.

### 5.4.1 Parser monad

It is easy to write a parser using Happy which, upon success returns the abstract syntax tree, and upon failure evaluates to ⊥. This style of parser does not permit us to generate parser messages of types other than String, errors can not be caught in a functional style, non-fatal messages are not supported, and there is no way for us to return more than one message. Luckily, Happy allows us to use monadic parsers which can solve all of these issues. To begin, let us introduce the following types for parser messages:

```hs
data MsgLevel = Info | Warn | Error
  deriving Eq

data ParserMsg = PM {
  msgLevel :: MsgLevel,
  msgPos :: Pos,
  msgDesc :: String
}
```

*MsgLevel* enumerates different severities of parser messages, ranging from the lowest level, *Info*, to the highest, *Error*. A single parser message consists of a severity, position, and description. Note that we use *Pos* instead of the *AlexPosn* type used by Alex, because we would like to accommodate for messages which are not caused directly by part of a source file:
In order to allow for messages to be generated in the order in which they occur, we define multiple messages using a difference list, which are described in subsection 5.4.2:

\[
\text{type } \text{ErrorS} = \text{Accum ParserMsg}
\]

Before we can define the parser monad, we need a suitable way of dealing with failure which does not discard the list of compiler messages. Normally, we would a monad transformer stack using \(\text{Maybe, Either, or ErrorT}\) for this purpose, but all of them can only distinguish between success or failure. In our case, we would like to be able to have a mix of both where we can continue despite the potential presence of error messages. To do this, we will define a variation of the state monad transformer whose result is wrapped into \(\text{Maybe}\):

\[
\text{newtype StateMT s m a} = \text{SM (s} \rightarrow \text{m (Maybe a,s))}
\]

Turning this type into an instance of \(\text{Monad}\) follows the general pattern for state monads and is fairly trivial:

\[
\text{instance Monad m }\Rightarrow\text{ Monad (StateMT s m) where}
\]
\[
\text{return x} = \text{SM (}\lambda s \rightarrow \text{return (Just x,s)})
\]
\[
(\text{SM m}) \gg f = \text{SM (}\lambda s \rightarrow \text{do}}
\]
\[
(r, s') \leftarrow m s
\]
\[
\text{case r of}
\]
\[
\text{Nothing} \rightarrow \text{return (Nothing, s')}
\]
\[
(\text{Just x}) \rightarrow \text{let (SM m') = f x in m' s')
\]

We note that failure is preserved, while computations are still executed to update the state which is also maintained in all cases. \(\text{Utility.StateM}\) is the module which contains this monad as well as miscellaneous helper functions, including instances of other classes such as \(\text{MonadTrans}\) and \(\text{MonadState}\). The parser is now simply a \(\text{Reader-like monad whose result is a computation of type StateM, where StateM is StateMT on top of the Identity monad:}\)

\[
\text{newtype Parser a} = P \{ \text{runParser :: AlexState } \rightarrow \text{StateM ErrorS a} \}
\]

\(\text{AlexState}\) is the type used by Alex to represent the lexer’s state which we need to pass on to parsers so that they can request tokens. We note that \(\text{Parser}\) could be defined in terms of \(\text{ReaderT}\) as \(\text{ReaderT AlexState (StateM ErrorS)}\), but use the above definition to gain more control about the way we handle failure:
instance Monad Parser where
{- usual definitions of return and >>= -}
fail msg = P (λs → addError s msg >>= failMT)

This definition shows us the important property we desire: raising an error by itself does not result in failure. We have to call failMT whose definition is shown below to indicate failure in the parser’s result, although even this will not stop consecutive parsers from running so that we can find as many errors as possible during one invocation of the compiler:

failMT :: Monad m ⇒ StateMT s m a
failMT = SM (λs → return (Nothing, s))

Adding errors using addError is also of interest due to our use of Accum. As stated previously, the specifics of this type will be discussed in subsection 5.4.2 to aid with the construction of a left-recursive grammar, but conceptually our goal is simply to add elements to the end of a list in \(O(1)\), which would normally be in \(O(n)\) if we were to use ++. Thus, we have the following definitions:

\[
\begin{align*}
\text{makeMsg} & : \text{MsgLevel} \to \text{AlexState} \to \text{String} \to \text{ParserMsg} \\
\text{makeMsg} \ l \ s \ m & = \text{PM} \ l \ (\text{FilePos} (\text{alex_pos} \ s)) \ m \\
\text{addAux} & : \text{MsgLevel} \to \text{AlexState} \to \text{String} \to \text{StateM ErrorS} () \\
\text{addAux} \ l \ s \ m & = \text{modify} \$ \ flip \ contA \ (\text{makeMsg} \ l \ s \ m) \\
\text{addError} & : \text{AlexState} \to \text{String} \to \text{StateM ErrorS} () \\
\text{addError} & = \text{addAux} \ \text{Error}
\end{align*}
\]

For warnings and information messages, we have addWarning and addInfo which are defined similarly. The key function here is addAux which updates the parser state with a new, partially applied function flip contA (makeMsg l s m) which effectively adds the message constructed using makeMsg to the end of the list of parser messages.

Lastly, we define a function lexer to wrap the lexical analyser into the parser, which will use this function whenever it needs more non-terminals:

\[
\begin{align*}
\text{lexer} & : (\text{TokenP} \to \text{Parser} \ a) \to \text{Parser} \ a \\
\text{lexer} \ f & = \ P \ (λs → \text{case} \ \text{runAlexFrom} \ \text{alexMonadScan} \ s \ of) \\
(\text{Left} \ err) & → \ \text{lexicalError} \ err \ s \ >>= \text{failMT} \\
(\text{Right} \ (s', t)) & → \ \text{let} \ (P \ m) = f \ t \ \text{in} \ m \ s')
\end{align*}
\]

We evaluate runAlexFrom with the current lexer state s, which will either result in an error or a pair consisting of a new state and an input token. In the latter case, we call
the continuation $f$ with the new token to produce a parser which is then applied to the new lexer state.

5.4.2 Left-recursion

LALR parsers like the ones generated by Happy are more efficient for left-recursive grammars, because we can reduce the parse stack more quickly. Let us consider the following, right-recursive, production for a list of comma-separated types and its semantic action which we may encounter in a naïve Happy grammar:

```plaintext
types :: { [Type] }

| types -> type0 { $1 } |
| types', types { $1 : $2 } |
```

This production is simple and easy to understand, but inefficient. The parser will not be able to reduce the parse stack until the end of the list has been found. We can improve this by changing the rules to use the preferred left-recursion:

```plaintext
types :: { [Type] }

| types -> type0 { $1 } |
| types', type0 { $2++[$1] } |
```

However, now we have an $++$ in one of the semantic actions, which is also inefficient as it runs in $O(n)$ each time we add an element to the list! We could improve this by constructing the list in reverse and then calling `reverse` on the result to restore the right order, but there is a better solution.

Wadler (1987) showed how functions which use $++$ can be transformed to use an accumulating parameter. Doing this will enable us to construct lists in the right order without $++$ or `reverse`. Since there are many productions in Cada’s grammar which produce lists of things and we have already encountered several other such scenarios as well, we define a type for this pattern in the `Utility.Accum` module:

```plaintext
type Accum a = [a] -> [a]
```

This type of list is also known as a difference or Hughes list. Back to our grammar, let us note that there are three different cases when it comes to producing lists in semantic actions: we have parsed a single element (base case for productions which parse one or more elements), we have parsed nothing (base case for productions which parse zero or more elements) or we have parsed an element and, recursively, a function of type `Accum a` (recursive case for all lists).

If we have no element, then finding an appropriate semantic action for a value of type `Accum a` is easy: it is simply the identity function. We take an accumulator as argument and just return it.
For the other base case, we wish to construct a function which takes an accumulator as argument, adds the new element to it and return the resulting list. This is the same as adding a single element to a list using cons, but writing semantic actions like \((:) \, $1\) does not look very nice. Consequently, we define an alias for cons whose type better reflects what we are doing as well:

\[
\text{cons}_A \; \vdash \; a \rightarrow \text{Accum } a \\
\text{cons}_A \; = \; (:) 
\]

Lastly, we need to deal with the recursive case. We define \(\text{cont}_A\) which builds a function which accepts three arguments: a function of type \(\text{Accum } a\), a value of type \(a\), and an accumulator of type \([a]\). We add the single element to the accumulator and then apply the given function to the result:

\[
\text{cont}_A \; \vdash \; \text{Accum } a \rightarrow a \rightarrow \text{Accum } a \\
\text{cont}_A \; f \; x \; xs \; = \; f \; (x : xs)
\]

We can now rewrite the type and semantic actions of the \(\text{types}\) production to make use of our more efficient functions:

\[
types_0 \; \vdash \; \{ \text{Accum Type} \} \\
types_0 \rightarrow \text{type}_0 \quad \{ \text{cons}_A \; $1 \} \\
, \text{types}_0 \rightarrow \text{type}_0 \quad \{ \text{cont}_A \; $1 \; $2 \}
\]

In order to become useful, we need to evaluate the functions of type \(\text{Accum Type}\) which are now being constructed by \(\text{types}_0\) into values of type \([\text{Type}]\). For convenience, we write a final helper function which does this for us:

\[
\text{un}_A \; \vdash \; \text{Accum } a \rightarrow [a] \\
\text{un}_A \; f \; = \; f \; []
\]

This function is then used in an additional production whose semantic action applies \(\text{un}_A\) to a function which was constructed by \(\text{types}_0\):

\[
types \; \vdash \; \{ [\text{Type}] \} \\
types \rightarrow \text{types}_0 \quad \{ \text{un}_A \; $1 \}
\]

### 5.4.3 Ambiguities

Unfortunately, there is an ambiguity in Haskell’s grammar for type signatures and this issue is inherited by Cada. Type contexts can be indistinguishable from types until \(\Rightarrow\) is encountered, because their structure resembles that of type tuples where each element is a type constructor applied to a variable.
If we wanted to write a parser for this, we would require infinite lookahead in order to determine whether there is a context or not. Consider the following examples of valid type signatures in Cada where $\triangleright$ indicates the current position of the parser:

$$
\begin{align*}
  f & :: \triangleright (\text{Foo } a, \text{Bar } b) \Rightarrow a \rightarrow b \\
  g & :: \triangleright (\text{Foo } a, \text{Bar } b) \rightarrow b
\end{align*}
$$

Even with one symbol look-ahead, we are unable to choose an appropriate production for the right-hand side of the $\triangleright$ until we reach an arrow. Of course, in theory, there can be infinitely-many elements in a tuple and the parser would therefore have to be capable of looking ahead at a potentially infinite number of symbols.

In order to circumvent this problem, we employ a little hack and parse the context as a type. While this allows for invalid definitions to get parsed successfully, it resolves the ambiguity as there is now always a type in this place. Once we know for certain that the type is actually a context, it is converted to a set of type constraints using the `ParserConstr.makeContext` function.

More ambiguities exists between patterns and expressions. In settings where we might expect either of the two, such as in e.g. statements, we will always parse everything as an expression. If we figure out that we need a pattern instead of an expression, we use `ParserConstr.exprToPattern` to convert from a value of type `Expr` to one of type `Pattern`.

### 5.4.4 Locations

For the purpose of generating useful error messages for programmers who will be using our compiler, we wish to keep track of the line and column numbers of objects in the abstract syntax tree. This information may be needed in some situations, but not others. Instead of adding this information to every data type in the AST manually, we define a new parametrised data type which can be wrapped around other types:

```haskell
data Loc a = Loc {
  locValue :: a,
  locPosn :: Pos
}
```

In many cases we will want to inspect or modify the data which has a location associated with it. Instinctively, we could just pattern match on such values, but this would become tedious and ugly. Luckily, `Loc` is trivially a functor in Haskell, such that we can map functions onto the inner value in an elegant manner:

```haskell
instance Functor Loc where
  fmap f (Loc x p) = Loc (f x) p
```
Together with the <$> notation for fmap, we can now give very elegant definitions for location-preserving functions. Consider the following contrived example, which takes an equation and wraps it into a top-level constructor:

\[
\text{makeValDef} :: \text{Loc Equation} \to \text{Parser (LocP Definition)}
\]

\[
\text{makeValDef le} = \text{return}$\text{ValueDef}$<$> le
\]

We will discuss the data types which form Cada’s abstract syntax tree shortly, but let us look at the LocP type for now. Large parts of the AST are parametrised to allow us to annotate it with more or less information, depending on the current compilation stage. This will naturally lead to a pattern, where data types have a parameter \(a\) which is used in the types of their constructors, but where we want to annotate the data types themselves as well. We define two type functions to help us with this – one to capture the general pattern and one specifically for Loc:

\[
\text{type RecA f a} = f(a f)
\]

\[
\text{type LocP a} = \text{RecA Loc a}
\]

Finally, the Cada.Location module contains various helper functions which are simple enough to require no further discussion. Instead, we will describe their behaviour whenever we encounter them later on.

### 5.5 Abstract syntax tree

From a theoretical point of view, a context-free grammar is defined in terms of a set of productions which we can use to derive whether a word is a member of the language represented by the CFG or not. However, a simple “yes” or “no” is obviously not sufficient for us as we need to know the structure of an input program if it is accepted by the parser. The first step is to find an appropriate, abstract representation of the syntax of our language. In other words, we need data types to represent the relevant information obtained from parsing a source file. As an example, we are interested in the names of variables and the composition of expressions, but not the location of semicolons or parenthesis as this information can be represented implicitly. In our implementation, Cada.STypes and Cada.AST contain these data types of which simplified\(^4\) versions may be found in appendix A.5.

The first module is home to the abstract representation of surface types whose need arises from the fact that the concrete syntax of types in the context-free grammar (section 4.5) differs significantly from that in the specification of the type system (Figure 4.1). For example, we do not yet know about the kinds of any of the types in

\[\text{4For simplicity, we have omitted the full record syntax used in the compiler sources}\]
a module and we have not resolved whether type constructors refers to real type constructors or type functions. We will later replace tuple and list types with appropriate type constructors as well.

Let us now briefly consider the data types in the second module, of which there are many, but few require much discussion. Note that we will often use “declarations” and “definitions” interchangeably throughout the following sections.

Firstly, we can observe the use of RecA in the definition of Module (Figure A.22) to both wrap Definition into a and to apply a to Definition. All declarations at the top-level are wrapped into a fitting constructor of Definition.

A value definition is an equation which binds an expression to a name (Figure A.23), optionally using zero or more patterns. We will often encounter patterns and expressions together, such as in let-bindings, prompting us to define a separate type named Alt for this combination.

In the definition of Literal (Figure A.24), we use the String type to represent values of integer literals in order to avoid having to deal with platform-specific issues such as different word sizes. Due to our choice of backend (section 5.12), this does not represent a serious problem, because GHC will take care of it for us using the Integer type if possible or a compile-time error if not.

5.5.1 Construction and validation

In order to construct the abstract representation of a module, we associate semantic actions with each production in our grammar for Happy. These will later be invoked by the resulting parser whenever a production has successfully been applied to reduce the parse stack. To avoid cluttering the grammar with code, we define most semantic actions as functions in a separate module named Cada.ParserConstr. As an example, let us consider the following non-terminal symbol for data fields:

\[
\text{dfield} :: \{ \text{DataField} \} \\
\text{dfield} \rightarrow \text{VAR}' ::' \text{type0} \quad \{\% \ \text{makeField} \ 1 \ 3 \ \} \\
\]

There is only one production, where VAR and ' ::' are terminal symbols, while type0 is another non-terminal symbol. If applied successfully, the semantic action, which is shown to the right in curly braces, will be invoked where $1$ and $3$ will be replaced with the semantic values of VAR and type0 respectively. makeField, which is defined in Cada.ParserConstr then performs the required construction and validation tasks:

\[
\text{makeField} :: \quad \text{TokenP} \rightarrow \text{SType} \rightarrow \text{Parser DataField} \\
\text{makeField} \ (p,n) \ t = \quad \text{do} \\
\quad \text{qualErrH} \ (\text{FilePos} \ p) \ (\text{tVal} \ n) \\
\quad \text{return} \$ \ DField \ (\text{tVal} \ n) \ t \ p
\]
While this is a relatively simple function, we can see the general structure of functions in this module. `qualErrH` is one of many validation functions defined in the `Cada.ParserError` module. Its particular purpose is to raise an error if the specified identifier is qualified:

\[
\text{qualErrH} :: \text{Pos} \rightarrow \text{String} \rightarrow \text{Parser}() \\
\text{qualErrH } p\ n | \text{isUnqualID } n = \text{return}() \\
| \text{otherwise} = \text{addErrorP } p\ (\text{ppQualError } n)
\]

The parser will attempt to perform as many validation tasks as possible, but some are postponed to later stages. For example, we do not yet check for duplicate symbols or circular dependencies in type functions.

In addition to the construction of the AST, `Cada.ParserConstr` also simultaneously removes syntactic sugar from state definitions to turn them into type functions, algebraic data types, getters, and setters. This process is heavily interleaved with other parts of the front end, so that we will focus on the key functions. Let us begin with the main function responsible for this task:

\[
\text{makeState} :: \text{AlexPosn} \rightarrow \text{TokenP} \rightarrow [\text{TypeParam}] \rightarrow \\
\text{Maybe SType} \rightarrow [\text{DataField}] \rightarrow \\
\text{Parser} (\text{LocP Definition})
\]

```
makeState\ p\ tp\ ps\ mt\ fs = let \\
\ t\ n = tkVal\ tp \\
\ d\ n = \ t\ n ++ "Data" \\
\ a\ s = makeAccessors (mkCtrPtr\ \ t\ n\ fs)\ fs\ in\ do \\
\ c \leftarrow buildDataCtr\ \ p\ \ t\ n\ \ fs \\
\ d \leftarrow makeDataDef\ \ State\ \ d\ n\ \ ps\ \ [c] \\
\ t \leftarrow makeStateTy\ \ p\ \ mt\ \ t\ n\ \ d\ n\ \ ps \\
\ returnL\ p\ SState\ SDef\ \ t\ d\ a s
```

The algebraic data type is constructed using `buildDataCtr`, which creates the single data constructor using the list of fields, and `makeDataDef`, which is responsible for the ADT itself. Secondly, we look at the construction of the type function, for which we will first define some helper functions:

\[
\text{makeTyVarsForParams} :: [\text{TypeParam}] \rightarrow [\text{SType}] \\
\text{makeTyVarsForParams} = \text{map} (\text{STyVar} \circ \text{tyParamName})
\]

\[
\text{makeStateDataTy} :: \text{String} \rightarrow [\text{TypeParam}] \rightarrow \text{SType} \\
\text{makeStateDataTy} \ dn = \text{foldl} \text{STyApp} (\text{STyCtr} \ dn) \circ \text{makeTyVarsForParams}
\]
`makeStateDataTy` takes the name of a data type `dn` and a list of type parameters `α` to produce a surface type of the form `dn α`. Next, we define a function which generates the right-hand side of the type alias, given an optional parent type and the result of the functions we have shown above:

```
makeStateTyRHS :: Maybe SType -> SType -> SType
makeStateTyRHS Nothing dt = STyApp (STyCtr "State") dt
makeStateTyRHS (Just t) dt = STyApp (STyApp (STyCtr "StateT") dt) t
```

As we can see, the presence of a parent type dictates whether we use the `State` monad or its transformer variant. Now we can put everything together to define the following function which constructs the type alias for a single state definition:

```
makeStateTy :: AlexPosn -> Maybe SType -> String -> String -> [TypeParam] -> Parser AliasDefinition
makeStateTy p mp tn dn ps = let dt = makeStateDataTy dn ps
                             in return $ ADef tn ps $ makeStateTyRHS mp dt
```

Lastly, we need to generate getters and setters for each field in the state definition using the `makeAccessors` function. Note that its first argument is a pattern of form `K x₀...xₙ`, produced using `mkCtrPtr'`, for the only data constructor of the algebraic data type corresponding to the state definition:

```
makeAccessors :: Pattern -> [DataField] -> [Equation]
makeAccessors pat fs = snd $ foldr (makeAccessor pat) (length fs - 1,[]) fs
```

Before we define `makeAccessor`, we first need two other functions, one for getters and one for setters. We only need to know the name of a field `f` to construct a getter of form `f.get = gets f`:

```
mkGetter :: String -> Equation
mkGetter n = Eq (n ++ ".get") $ Alt [] $ App (Var "gets") (Var n)
```

Setters are slightly more complicated and require the name of the field, the constructor pattern, and the index of the field. We produce a setter of form `f.set = λv -> modify (λ(K x₀...xₙ) → [xᵢ/v](K x₀...xₙ))` for a field `f` using:

```
mkSetter :: String -> Pattern -> Int -> Equation
mkSetter n p i = Eq (n ++ ".set") $ Alt [VarPattern "v"] $
                 App (Var "modify") (Abs $ Alt [p] (modifyCtrVal p i "v"))
```
The `modifyCtrVal` function substitutes the variable pattern at index \( i \) in \( p \) for a variable named \( v \). Finally, `makeAccessor` generates one getter and one setter for each field:

\[
\text{makeAccessor} :: \text{Pattern} \rightarrow \text{DataField} \rightarrow \text{(Int, [Equation])} \rightarrow \text{(Int, [Equation])}
\]

\[
\text{makeAccessor } p (\text{DField } n _) (i, \text{eqs}) = (i - 1, \text{get : set : eqs})
\]

where

\[
\begin{align*}
\text{get} &= \text{mkGetter } n \\
\text{set} &= \text{mkSetter } n p i
\end{align*}
\]

## 5.6 Pretty printing

In order to help with the closely related tasks of debugging the compiler and emitting useful error messages, we require a method to convert the abstract syntax tree, or parts of it, back into a human-readable representation. Conceptually, this is a very easy task which can be solved simply using pattern matching and the concatenation of strings. Though, in practice, it makes sense to design a few combinators to solve this task more elegantly. Indeed, we can even find some discussion about different approaches to this problem in literature (Hughes, 1995; Wadler, 2003). However, it turns out that we do not require such sophisticated solutions which would introduce additional dependencies and make our code more convoluted. Instead, we will base our implementation on the `ShowS` type which is defined in Haskell’s standard library:

\[
\text{type } \text{ShowS} = \text{String} \rightarrow \text{String}
\]

We choose this type rather than `String`, because it allows us to compose different pretty-printers in constant time. Observant readers will have noticed that `ShowS` is equivalent to `Accum String` (subsection 5.4.2), meaning that we can recycle the helper functions we have already defined for the pretty printer.

### 5.6.1 Pretty printing combinators

Pretty printing is not actually a technique specific to the abstract syntax tree and may applied to other problems, such as the construction of compiler messages or the visualisation of other objects. Therefore, we will first define a few generic combinators in `Utility.PrettyPrint`, which can later be used in any setting. As an example of a simple function in this module, consider the following definition where another pretty printer, \( f \), gets wrapped into parenthesis:

\[
\text{ppInParens} :: \text{ShowS} \rightarrow \text{ShowS}
\]

\[
\text{ppInParens } f = \text{showChar } '(' \circ f \circ \text{showChar } ')'
\]
This function can then also be used in other combinators, e.g. to make the parenthesis optional, depending on the value of an additional argument:

\[
ppInOptParens :: \text{Bool} \rightarrow \text{ShowS} \rightarrow \text{ShowS}
\]

\[
ppInOptParens \ True \ f = ppInParens \ f
\]

\[
ppInOptParens \ \_ \ f = f
\]

A more sophisticated, and more frequently used, pretty printer is \(ppDefs\) which formats a list of things using a given function \(f\):

\[
ppDefs :: (a \rightarrow \text{ShowS}) \rightarrow [a] \rightarrow \text{ShowS}
\]

\[
ppDefs \ f = \text{foldr} (\lambda \, x \, xs \rightarrow f \ x \circ xs) \ \text{id}
\]

### 5.6.2 Pretty printing of abstract representations

Using the combinators in \(\text{Utility.PrettyPrint}\) it is now trivial to write pretty printers for all the different types which are used in the abstract representation of Cada programs. Let us consider the following pretty printer for the \(\text{Pattern}\) type which demonstrates how different utility functions are used to reconstruct the corresponding surface syntax of the arguments:

\[
ppPattern :: \text{Pattern} \rightarrow \text{ShowS}
\]

\[
ppPattern \ \text{Wildcard} = \text{showChar} \ '_'
\]

\[
ppPattern \ (\text{VarPattern} \ x) = \text{showString} \ x
\]

\[
ppPattern \ (\text{LitPattern} \ l) = ppLiteral \ l
\]

\[
ppPattern \ (\text{CtrPattern} \ c \ [\]) = \text{showString} \ c
\]

\[
ppPattern \ (\text{CtrPattern} \ " \ [\, x, y]\) = ppInParens \$
\]

\[
ppPattern \ x \circ \text{showString} \ " \ : \" \circ \text{ppPattern} \ y
\]

\[
ppPattern \ (\text{CtrPattern} \ c \ ps) = ppInParens \$
\]

\[
\text{showString} \ c \ \circ
\]

\[
ppSpace \ \circ
\]

\[
ppPatterns \ ps
\]

A more interesting problem is the reconstruction of parenthesis in as few cases as are required to disambiguate the output. We do this using an extra parameter for the respective pretty printers which keeps track of the precedence of the surrounding context. If this value is too large, parenthesis may be required. For example, arrow types are right-associative and bind less strongly than type application. Therefore parenthesis are needed to represent \(\text{Maybe} \ ((a \rightarrow b) \rightarrow c)\), but \(\text{Maybe} \ a \rightarrow (b \rightarrow c)\) can be formatted as \(\text{Maybe} \ a \rightarrow b \rightarrow c\). With the help of the \(ppInOptParens\) combinator, this behaviour can easily be implemented:
We can see how the cases for arrows and type applications behave depending on the precedence $w$ and how they change it for recursive calls. For clarity, we have cheated a little bit in the example above as there is no $STyArr$ constructor for $SType$. Instead, the correct pattern should be:

$$STyApp (STyApp (STyCtr "-»") l) r$$

### 5.7 Type system

The by far most complex part of the compiler is dedicated to the implementation of the type system. Once an abstract syntax tree has been constructed for a module, the compiler begins the lengthy process of preparing it for type inference. We give an overview of the Haskell types used for this component of the compiler in appendix A.6. Let us note that the implementation of our type inference algorithm is a heavily modified version of the code shown in Jones (1999), with the additions of type functions and error handling (section 5.11). Additionally, we make use of more efficient data structures with regards to time complexity, such as balanced binary trees, add the various preparation stages which proceed type inference but are not discussed in Jones’ paper, and integrate the whole system in our compiler.

#### 5.7.1 Type environments

During the compilation of a module, we need to keep track of all the information needed by the type system. For this purpose we use environments which are simply mappings from names to values of some type:

$$\text{type } \text{Env} = M.\text{Map } \text{String}$$

Here we use the $\text{Map}$ type from the $\text{Data.Map}$ module (based on Adams (1993); Nievergelt and Reingold (1973)) which uses balanced binary trees and thus gives us
$O(\log n)$ insertion and lookup time complexities, in contrast to $O(n)$ lookup and insertion (tail) for linked lists. We will make use of this module, as well as `Data.Set` which is part of the same library, throughout this chapter.

There are five different environments in the type system for algebraic data types, type functions, type classes, typings, and class instances respectively:

```
type ADTEnv   = Env ADT
type AlEnv    = Env Alias
type ClEnv    = Env TypeClass
type ExEnv    = Env PolyType
type InEnv    = Env [Instance]
```

All five environments are combined in one data type which holds all the information needed by the type system to function:

```
data Envs    = Envs {
    adtEnv :: ADTEnv,
    alEnv :: AlEnv,
    clEnv :: ClEnv,
    exEnv :: ExEnv,
    inEnv :: InEnv
}
```

Values of this type will be constructed and updated throughout the stages which precede type inference, where it is then used for e.g. context reduction and the expansion of type functions.

### 5.8 Module system

The first stage of the process which prepares a module for type inference is a simple one. We need to extract the list of module imports from the abstract syntax tree, import those modules, and construct type environments from them. There are only three short functions needed to accomplish the first step:

```
extracl' :: Definition Loc → Accum ModuleName
extract' (ImportDef m) = consA m
extract' _ = id

extract :: [Definition Loc] → [ModuleName]
extract = foldr extract' []

collectImports :: [LocP Definition] → [ModuleName]
collectImports = extract ∘ map unL
```
As we can see above, extracting dependencies can be accomplished simply using a fold on the list of declarations obtained from the abstract syntax tree. Next, we need to sort the modules based on their dependencies and ensure that there are no circles in the dependency graph. For this purpose, we will use functions from the `Data.Graph` module, which is part of the `containers` library (based on King and Launchbury (1995)), to sort dependencies into a graph of strongly connected components:

\[
\text{type ModuleNode} = (\text{Context}, \text{ModuleName}, [\text{ModuleName}])
\]

\[
\text{makeNode} :: \text{Context} \to \text{ModuleNode}
\]

\[
\text{makeNode} \; \text{ctx} = (\text{ctx}, \text{moduleName} \; m, \text{collectImports} \; (\text{moduleDefs} \; m))
\]

\[
\text{where}
\]

\[
m = \text{unL} \; (\text{ctxAST} \; \text{ctx})
\]

\[
\text{findDeps} :: [\text{Context}] \to [\text{SCC ModuleNode}]
\]

\[
\text{findDeps} = \text{stronglyConnCompR} \circ \text{map makeNode}
\]

Values of type SCC may be constructed using one of two constructors: `AcyclicSCC` for single nodes in the graph or `CyclicSCC` for a list of nodes which form a circle with each other. The result of `findDeps` will be a list of nodes which are sorted such that we would expect the module with no dependencies to any of the other modules to come first and that all elements of the list are constructed using `AcyclicSCC`. If there is an element which is constructed using `CyclicSCC`, then there is a circle in the dependency graph and an error will be raised.

We cannot compile circular dependencies during one invocation of the compiler, because we would have to arbitrarily decide which module to compile first and it may not be possible to do so if the module we choose requires knowledge about one of the other modules. It may be possible to tackle this problem by compiling multiple modules simultaneously and by resolving dependencies on the declaration level, but this would lead to a very complicated system which does not contribute to our goals.

Note, however, that one may bootstrap a group of mutually dependant modules if this should be required. Suppose that we have two modules, $A$ and $B$, then we can first compile a basic version of, for example, $A$ which does not import $B$. Once it has been successfully compiled, we compile the other module which imports $A$. Lastly, we modify $A$ to import $B$ and compile it again. This works because our compiler only looks for `.co` interface files when it imports modules and does not ensure that they are recompiled if their corresponding sources have changed.
5.8.1 Interface files

An interface file is generated for every module after type inference has completed successfully. These files contain all information about a module which is relevant during the compilation of other modules. Specifically, it contains serialised versions of all type environments.

When an interface file is generated, the module’s environments are serialised to a binary format. Once we import an interface, this data is deserialised from binary back into a value of type `Envs`. Both processes are contained in the `Compiler.Interface` and `Compiler.Binary` modules, which make use of `Data.Binary`, `Data.Binary.Get`, and `Data.Binary.Put` from the `binary` package\(^5\). All types which need to go through this process are made instances of `Binary` such as, for example, `TyVar` below:

```haskell
instance Binary TyVar where
  put (TyVar n k) = do
    put n
    put k
  get = TyVar <$> get <*> get
```

Instances for other types follow a similar structure, except where there are multiple constructors, in which case we need to serialise an integer tag so that we can later restore a value using the right constructor. Another little subtlety is required when we load interfaces due to the nature of lazy IO:

```haskell
...
  h ← openFile fp ReadMode
  hSetBinaryMode h True
  !envs ← deserialiseEnvs h
  hClose h
  ...
```

Note the use of the bang pattern `!envs`, enabled using GHC’s `BangPatterns` extension, which forces the thunk referenced to by `envs`. This is necessary, because otherwise we would have to keep the file handle `h` open until `envs` is no longer needed. However, there are two reasons why this would not be practical. Firstly, there may be a large number of imported modules and thus file handles, but the number of those which may be open at the same time is usually restricted. Secondly, we would be forced to structure our code such that all further processing takes place before we call `hClose h`, which would clearly be very inconvenient.

\(^5\)http://hackage.haskell.org/package/binary
5.9 Kind inference

In subsection 4.3.2 we have explained that the algorithm needed for kind inference is essentially the same as for type inference. To this end, we described a process using which objects in the type system may be translated to expressions and now we will implement it. As an example to get going, let us consider the conversion of surface types to expressions:

\[
\begin{align*}
demoteType & :: SType \rightarrow Expr \\
demoteType (STyVar x) & = Var x \\
demoteType (STyCtr "->") & = Var "t->" \\
demoteType (STyCtr x) & = Var ('t' : x) \\
demoteType (STyApp f a) & = App (demoteType f) (demoteType a) \\
demoteType (STyList t) & = App (Var "t []") (demoteType t) \\
demoteType (STyTuple ts) & = foldl App (tupleCtr ts) (map demoteType ts)
\end{align*}
\]

This is, unsurprisingly given the correspondence between expressions and types, a rather unremarkable procedure. A noteworthy exception here is that we add 't' as a prefix to the names of all constructors, so that we do not run into name clashes. More complex constructions such as algebraic data types are also easy to translate:

\[
\begin{align*}
demoteDataCtrs & :: [Loc (Typed DataConstructor)] \rightarrow Expr \\
demoteDataCtrs & = foldr1 (<+>) \circ map (demoteDataCtr \circ unL)
\end{align*}
\]

The <+> operator in this definition represents ⊔ and is used to combine two expressions into one as arguments of a function whose kind is Star → Star → Star as described in the specification:

\[
(<>+) :: Expr \rightarrow Expr \rightarrow Expr \\
(<>+) = InfixOp "+"
\]

We require various assumptions about the types of such primitives during kind inference and represent them in the Internal.Kinds module as a list of typings. Each primitive type (section 4.7) must have a kind built into the compiler, in addition to type arrows and combinators. The structure of this file is as follows, with emphasis on the definitions required for ⊔:
All type declarations are translated into values of type \textit{Equation}, with the exception of state declarations which yield two equations and rely on difference lists for efficiency:

\begin{align*}
\textit{demoteState} & :: \text{StateDefinition} \rightarrow \text{Accum Equation} \\
\textit{demoteState} \; s & = \\
& \text{cons} \; A \; (\textit{demoteAlias} \; (\textit{sDefType} \; s)) \circ \\
& \text{cons} \; A \; (\textit{demoteDataTy} \; (\textit{sDefData} \; s))
\end{align*}

The key function exported by the \textit{TypeSystem.STypeDemotion} module is \textit{demoteModule} which composes the relevant functions to coordinate the demotion process:

\begin{align*}
\textit{demoteModule} & :: \text{Module Loc} \rightarrow \text{BindGroup} \\
\textit{demoteModule} & = \text{toBG} \circ \textit{findDeps} \circ \textit{demoteDfts} \circ \textit{moduleDfts}
\end{align*}

Note that \textit{toBG} and \textit{findDeps} sort the resulting equations into groups of strongly connected components (see the explanation in section 5.8) based on other types they reference. This time it is not the presence of cyclic dependencies we are looking for, but in order to derive the most general kinds it is important that we do not consider all types at once as this would restrict kinds unnecessarily.

Once we have groups of equations the normal type inference algorithm, which we will describe later in section 5.11, can be used together with the kind assumptions we have shown above and empty type environments:

\begin{align*}
\textit{inferKinds} & :: \text{Assumps} \rightarrow \text{BindGroup} \rightarrow \text{Either TypeError Assumps} \\
\textit{inferKinds as} & = \text{inferTypes initialEnvs (as <> kindAssumps)}
\end{align*}
5.10 Conversion

Now that we know the kinds of most types in a module, we can convert their representation from surface types (section 5.5) to the internal depiction of types (section 5.7, appendix A.6). In the process, we will also complete the following tasks:

- Check that the definitions of type functions do not form circles
- Check that the hierarchy of type classes does not form any circles
- Identify type functions and check that they are fully applied

There is a lot to do but, in order to identify type functions, we need to construct their internal representations first. Therefore, the conversion process begins by sorting them into strongly connected components and ensuring that there are no cycles using the usual process (section 5.8).

If this test is completed successfully, we can convert the type functions to our internal representation. This serves as an example for the general process followed in the translation of all type declarations:

\[
\text{mk Alias} :: \text{AliasDefinition} \rightarrow \text{EnvsTrans Alias}
\]
\[
\text{mk Alias} (ADef n ps st) = \text{do}
\]
\[
k \leftarrow \text{lift} \, \$ \, \text{lookupKind} \, n
\]
\[
t \leftarrow \text{mk AliasType} \, ps \, (\text{kindArgs} \, k) \, st
\]
\[
return \, \$ \, \text{Alias} \, k \, (\text{length} \, ps) \, t
\]

We begin by looking up the kind of the type function, referred to as alias in the sources, and then we convert the right-hand side of the declaration to a polytype:

\[
\text{mk AliasType} :: [\text{TypeParam}] \rightarrow [\text{Kind}] \rightarrow \text{SType} \rightarrow \text{EnvsTrans PolyType}
\]
\[
\text{mk AliasType} \, ps \, ks \, st = \text{do}
\]
\[
env \leftarrow \text{get}
\]
\[
mt \leftarrow \text{lift} \, (\text{fromType} \, env \, [] \, st)
\]
\[
return \, \$ \, \text{ForAll} \, ks \, (S.\text{empty} :=> \text{paramsToGens} \, ps \, ~=> \, mt)
\]

We have previously obtained the kind of each type parameter from \text{kindArgs} which are now used for the polytype’s quantifiers. The actual translation of the surface type takes place in \text{fromType} which requires the current type environments to look up type functions. Note that we will never encounter a situation where a type function will be converted before one of its dependencies is added to the corresponding type environment, because we maintained the order of dependency obtained from sorting the type aliases earlier on.
The \( \sim \rightarrow \) operator constructs a substitution for, in this case, \( ps \mapsto mt \). Substitutions are represented by the \textit{Map} type we introduced earlier (section 5.7) and can be composed using the \( <\rightarrow \) operator.

Converting polytypes and contexts is a simple procedure, but the translation of monotypes is more difficult. Recall that the surface representation of these represents type constructors using \( STyCtr \ String \). In other words, we do not know how many arguments a constructor is applied to and need to reconstruct this information in case the constructor turns out to be a type function. Type functions must be fully applied because otherwise it would not be possible to decide whether two types are equal:

\[
\text{fromTy}pe :: \text{Envs} \rightarrow [\text{MonoType}] \rightarrow \text{SType} \rightarrow \text{Conversion MonoType}
\]

\[
\text{fromTy}pe \text{ env } ps (\text{STyVar } n) = \text{return }\$
\]

\[
\text{foldl } \text{TApp} (\text{TVar }$ TyVar n KStar) \text{ ps}
\]

\[
\text{fromTy}pe \text{ env } ps (\text{STyCtr } n) = \text{do}
\]

\[
k \leftarrow \text{lookupKind } n
\]

\[
env' \leftarrow \text{getcdEnvs}
\]

\[
\text{case } \text{isAlias } (\text{alEnv env} \cup \text{alEnv env'}) n \text{ of}
\]

\[
(\text{Just } a) \rightarrow \text{mkTyFun } n k (\text{aliasArity } a) \text{ ps}
\]

\[
\text{Nothing} \rightarrow \text{return }\$_{\text{foldl } \text{TApp} (TCtr$ TyCtr n k) \text{ ps}}
\]

\[
\text{fromTy}pe \text{ env } ps (\text{STyApp } f a) = \text{do}
\]

\[
at \leftarrow \text{fromTy}pe \text{ env } [\text{]} a
\]

\[
\text{fromTy}pe \text{ env } (at : \text{ps}) f
\]

Initially \textit{fromTy}pe may look very confusing but the idea is simple: instead of committing to values constructed using \textit{TApp} in the case for \textit{STyApp}, we keep track of all arguments using an accumulator of type \([\text{MonoType}]\) and follow the left branch. If the leaf at the end of this branch is a type variable then the case is clear and we use a left fold to apply the arguments we have accumulated to the variable. However, if we encounter a type constructor in this position then the our action depends on \textit{isAlias}.

If the type constructor represent a type function, then we will generate an appropriate internal representation for it using \textit{mkTyFun} which also ensures that we have accumulated enough (or more) arguments for it. An error is triggered if this is not the case. In the event that the type constructor does not represent a type function, we proceed as with type variables.

Once all type declarations have been converted, we sort type classes based on their superclasses into strongly connected components. If a cycle is detected we will raise an exception and produce an error message informing the user of the names of the relevant type classes. Otherwise we return a new set of type environments.
5.11 Type inference

In addition to Algorithm $W$ (Damas and Milner, 1982; Milner, 1978) which can be seen in Figure A.5, various other algorithms have been proposed to improve upon the shortcomings of $W$. Lee and Yi (1998) formalised Algorithm $M$ which works from the top-down, rather than the bottom-up, by placing constraints on the types of subterms. It has the property that it will always terminate sooner than $W$ if no type can be assigned to a term. Thus $M$ may be favourable if terms are expected to be very large.

Heeren et al. (2002) described a method based on constraint solving in order to produce better error messages. Depending on the order in which the constraints are solved, their algorithm may correspond to $W$ or $M$, but type graphs (Port, 1988) can make this process unbiased. Implementing their algorithm is more challenging than for the other algorithms, however.

As we have indicated previously in section 5.7, our type inference algorithm is based on the code presented by Jones (1999), which is an implementation of $W$. We make this choice because it makes more sense for our purposes to build on top of a working system which is easy to implement. However, we note that a solution based on constraint solving would be more beneficial for general purpose implementations.

However, the presentation of the algorithm in Jones (1999) has two main shortcomings which we will address in this section. Firstly, if an error occurs in the original presentation then the algorithm will evaluate to $\bot$ which provides us with little information about the cause of the error. We will show how error messages can be enhanced with the contexts in which they occur in and how monads can be used as pure abstractions for exceptions. Secondly, we will demonstrate how type functions can be evaluated during unification instead of prior to type inference. This will lead to more informative and concise error messages which reflect the explicit typings in the concrete syntax of a program.

5.11.1 Type errors

Since we use the same algorithm for kind and type inference, we need to find a way to produce sensible error messages for both. For example, referring to an “expression” while we are inferring the kind of an algebraic data type may not be very informative for users of our compiler. Instead of using plain strings to tackle this problem, we will construct values of the data type shown below which we can later convert to an appropriate, human-readable error message which may be different depending on whether the error was raised during kind or type inference:
The recursive cases of this type are used to add context to an error. To understand how this works, let us first define the monad stack used for type inference:

```haskell
data TypeError = UnifyError MonoType MonoType
    | OccursCheck TyVar MonoType
    | ...
    | ExprError Expr TypeError
    | OptionError Option TypeError
    | ...
```

`TE` is very similar to the presentation of the type inference monad in Jones (1999), except that we make use of an existing counter type which is defined in `Utility.Counter` to generate fresh names for type variables. `TI` adds error handling on top of this using the `ErrorT` monad transformer. Errors raised in this monad can be caught using the `catchError` function, thus allowing us to enrich errors by catching them first, then wrapping them into more information, and raising them again in the end:

\[
inContext :: TI a \to (TypeError \to TypeError) \to TI a
inContext m f = m 'catchError' (throwError \circ f)
\]

As an example of how this function is used, consider `tiEquation` which is just a wrapper around `tiEquation'` to catch errors using `inContext`:

```haskell
tiEquation :: Infer Equation MonoType
tiEquation env as e = tiEquation' env as e 'inContext' EqError e
```

Similar definitions exist for the other type inference functions. Once an error has been returned by our algorithm, we we will convert it to a value of type `String` which can be shown to a user. For example, an error constructed using `ExprError` may result in a message similar to "in an expression: . . ." if it was raised during type inference or "in a type: . . ." if it was raised during kind inference.

### 5.11.2 Type functions

We could resolve type functions ahead of type inference as we know that they will always be applied to all of their arguments, but this would be confusing in the event that something goes wrong during type inference. A better solution is to evaluate type functions only when necessary. In other words, we will change the unification algorithm (Robinson, 1965) to try and find a most general unifier without expanding any type functions. Only if that does not succeed will we evaluate one type function at a time. Let us define a function which does that:
A compiler for Cada

\[ \text{resolve} :: \text{Monad } m \Rightarrow \text{Envs } \rightarrow \text{TyFun } \rightarrow \text{[MonoType]} \rightarrow \text{Unify } m \text{ MonoType} \]

\[ \text{resolve env } (\text{TyFun } n \ k) \ ps = \text{case } M.\text{lookup } n \ (\text{alEnv env}) \ of \]

\( (\text{Just } \text{al}) \rightarrow \text{return } \$ \ \text{inst } \ps \ \$ \ \text{mkMono } (\text{aliasTy al}) \]

\( \text{Nothing } \rightarrow \text{throwError } \$ \ \text{OtherError } "\text{Unknown alias in unification."} \]

Given a type function with name \( n \) which is applied to the arguments represented by \( \ps \), we lookup the name in the type environment for type aliases. Note that this lookup should \textit{always} succeed as we would not be aware that \( n \) is a type function unless it exists in the environment. We then obtain the right-hand side of the type function, discard all quantifiers and instantiate it with the arguments. Using this function we can now add three cases to the unification algorithm:

\[ \text{mgu} :: \text{Monad } m \Rightarrow \text{Envs } \rightarrow \text{MonoType } \rightarrow \text{MonoType } \rightarrow \text{Unify } m \text{ Theta} \]

\[ \text{mgu env } (\text{TVar } tv) \ t = \text{bind } tv \ t \]

\[ \text{mgu env } t \ (\text{TVar } tv) = \text{bind } tv \ t \]

\[ \text{mgu env } (\text{TCtr } tc) \ (\text{TCtr } tc') \]

\[ | \ tc == tc' = \text{return epsilon} \]

\[ \text{mgu env } @(\text{TFun } t_f \ ps) \ t'@(\text{TFun } t_f' \ ps') \]

\[ | \ t == t' = \text{do} \]

\[ \xs \leftarrow \text{mapM } (\text{uncurry } \$ \ \text{mgu env}) \ (\text{zip } \ps \ \ps') \]

\[ \text{return } \$ \ \text{foldl } (<> ) \ \text{epsilon } \xs \]

\[ \text{mgu env } (\text{TApp } f \ a) \ (\text{TApp } f' \ a') = \text{do} \]

\[ s1 \leftarrow \text{mgu env } f \ f' \]

\[ s2 \leftarrow \text{mgu env } (s1 \sim > a) \ (s1 \sim > a') \]

\[ \text{return } \$ \ s2 <> s1 \]

\[ \text{mgu env } (\text{TFun } t_f \ ps) \ t = \text{do} \]

\[ t' \leftarrow \text{resolve env } t_f \ ps \]

\[ \text{mgu env } t' \ t \]

\[ \text{mgu env } t \ t' = \text{do} \]

\[ t' \leftarrow \text{resolve env } t_f \ ps \]

\[ \text{mgu env } t \ t' \]

\[ \text{mgu env } t \ t' = \text{throwError } \$ \ \text{UnifyError } t \ t' \]

If both types are type functions and their names are the same then unify their arguments. Note that we do not need to test for a matching number of arguments, because it is an invariant that type functions are always applied to all of their arguments and if we have the same name then this guarantees the same number of arguments.

Should all of the first four equations fail to find a most general unifier then the other new cases come into action which will arbitrarily evaluate type functions one at a time before attempting to unify the resulting types. This guarantees that we will eventually
find a mgu if there is one, although given that we have a bias to the evaluate the first type it may not be optimal in the sense that it may require more type functions to be evaluated than necessary. However, finding a better solution for this problem is not an easy task. It may be possible to construct a tree with branches for all orders in which type functions can be evaluated and to apply breadth-first search to it.

5.12 Backend

Imperative programming languages are easy to compile to machine code for real processor designs such as x86 and ARM, due to the nature of the paradigm which merely provides abstractions for the machine stack, memory access, and so on. Functional languages, on the other hand, are based on the λ-calculus and while they can easily be interpreted by a program, compiling them into machine code for stock hardware is not an easy task. In the case of Cada, we have to compile a range of features for which there is no simple procedural representation, including higher-order functions, partial function application, and laziness.

The usual approach to compiling functional languages is to invent an abstract machine which can be used as the target of compilation. In other words, we try to create a simple machine capable of executing programs constructed in the functional style using a sufficiently small set of instructions. We then implement an interpreter for such a machine on a stock platform and use it to run programs corresponding to the functional source language(s). However, the intermediate interpretation step leads to lower efficiency when compared to native programs for a hardware platform.

Peyton Jones (1992) introduced the Spineless Tagless G-Machine, an abstract machine for non-strict functional languages, designed with runtime efficiency in mind through the use of, for example, unboxed values and tagless heap objects. An implementation of the STG machine has since been used as the target platform for Haskell and has subsequently undergone further performance analysis such as in e.g. Marlow and Peyton Jones (2004).

Ideally, we would like to implement a runtime system based on the STG machine and use it as target for the compilation of Cada programs. However, doing so is not feasible given the scope and time constraints of this project. Marlow and Peyton Jones (2012) give an outline of the components of the runtime system used by GHC and the steps required to produce compatible code. Even if we aim for a simplified system, consisting of only an implementation of the STG machine and a garbage collector, and were to use a compilation framework like LLVM (Lattner and Adve, 2004) in order to avoid having to deal with code generation for different hardware platforms, this task would still represent a significant resource investment.
Instead, we choose to take the approach used by many programming languages in their infancy and target a high-level language similar to ours. This has the obvious disadvantage of introducing a major dependency to our project, but also enables us to generate executables which are likely to be much more efficient and useful than if we were using a simple interpreter.

5.12.1 Code generation

Generating Haskell code for Cada modules is a trivial task in that no significant transformation between the two languages needs to take place. State declarations will have already been desugared into type functions and algebraic data types by the parser, such that we can simply take the abstract syntax tree together with the type environments to pretty-print the corresponding Haskell code. This process does not diverge greatly from the one described in section 5.6, except for the differences in syntax. The corresponding functions can be found in the CodeGen.HsPretty and CodeGen.HsCodeGen modules.

We make use of the NoImplicitPrelude and RebindableSyntax extensions in GHC to get generated programs to use Cada’s standard library instead of Haskell’s. Without the first extension, we would encounter problems with overlapping names in cases where they are the same across the different libraries. The second extension allows us to use functions from our standard library for syntactic sugar in Haskell, such as the do-notation and patterns which would otherwise use functions from the Prelude even though it is not brought into scope.

In order to invoke the Glasgow Haskell Compiler from our code, we need to be able to spawn new processes. This can be done using the System.Process module in the process library. To make our lives easier, we define a helper function to obtain the handle of a newly created GHC process where ps is a list of parameters for it:

```haskell
ghcProc :: [String] \rightarrow IO ProcessHandle
ghcProc ps = do
    (_, _, h) <- createProcess $ proc "ghc" ps
    return h
```

Next, we define a wrapper around ghcProc to add search paths, obtained using gets cmpPaths from the compiler configuration, wait for the process to terminate, and to evaluate its exit status:
Each Cada module is compiled separately into an object file using the following function. The $fp$ variable contains the path to the Haskell source file for the module and $\text{swapExtension}$ is used to replace .hs with .o in the name of the file. We then invoke GHC with the -c switch which tells it to only compile the specified module and to not perform any linking:

\[
\text{cmpObjectFile} :: \text{FilePath} \rightarrow \text{Compiler FilePath}
\]

\[
\text{cmpObjectFile} \, fp = \text{do}
\]

\[
\begin{align*}
&\text{let} \\
&\quad \text{obj} = \text{swapExtension} ".o\" \, fp \\
&\quad \text{invokeGHC} \, [fp,\"-c","-o"\,\text{obj}] \\
&\end{align*}
\]

Finally, unless the --no-link option is active, $\text{cmpBinary}$ will be called to link the object files for all modules together into one executable. Note that we leave it up to GHC to choose an appropriate linker for the current platform, because it already has this knowledge and we know that it is readily available:

\[
\text{cmpBinary} :: \text{FilePath} \rightarrow \text{FilePath} \rightarrow \text{Compiler ()}
\]

\[
\text{cmpBinary} \, os \, bf = \text{invokeGHC} \, (os + [+"-o",bf]) \, ()
\]

### 5.12.2 Runtime support

Some types, such as $\text{Int}$ and $\text{IO}$, require special consideration as they can neither be represented in Cada nor Haskell. However, they are part of GHC’s runtime system and can be accessed through the $\text{Prelude}$. In order to bridge the gap, we write a single Haskell module as part of our standard library which, unlike all code generated by our compiler, imports $\text{Prelude}$ and re-exports relevant types and functions. Most functions in this file follow the pattern shown below where we give an explicit typing to select a specific instance of some method which cannot be expressed in Cada:

\[
\begin{align*}
\text{bindIO} :: \text{P.IO} \, a \rightarrow (a \rightarrow \text{P.IO} \, b) \rightarrow \text{P.IO} \, b \\
\text{bindIO} = (\text{P.} \gg=)
\end{align*}
\]
Note that $P$ here refers to Prelude. The types exported by this module are $\text{Bool}$, $\text{Int}$, $\text{Char}$, and $\text{IO}$. All exports are added to the corresponding environments in the $\text{Internal.NCC}$ module which we use as assumptions during kind and type inference.

### 5.12.3 Entry point

Haskell expects a program’s entry point, a function named `main`, to be located inside a module `Main`, but Cada is more liberal and does not have this restriction. While there is an option to tell GHC to use a different module, it needs to be specified during the compilation of that module, which is incompatible with our design where the name of the module containing the entry point may be specified independently. To resolve this issue, our compiler automatically generates a `Main` module in a temporary folder, which contains a single equation where $M$ is a qualified import of the module which contains the real entry point:

$$\text{main} = M.\text{main}$$

### 5.13 Testing

Throughout this dissertation we have asserted that one of the main advantages of pure languages is the ease with which we can prove the correctness of programs written in them. Given that our implementation language is Haskell, doing this may seem like an obvious choice for our compiler. In order to prove the correctness of a compiler, we would need to show that the semantic meaning of a program is the same as the result of executing a compiled version of the program. We can show this idea more easily using the following commutativity diagram:

![commutativity diagram](image)

Sadly, it turns out that even though we are working with a pure language, proving the above is not an easy task. There are many questions which would need to be answered first, ranging from “What is the meaning of a Cada program?” to “How do we deal with module imports and side-effects?”. Leroy (2009) reports on a formally verified compiler for a subset of the C programming language, written in the proof assistant Coq (Bertot and Castéran, 2004). However, the formal verification of compilers is still very much an ongoing research topic. Additionally, we have neither written our compiler in a proof assistant, nor do we want to restrict ourselves to a subset of the
language. Thus we can conclude that formal verification of our compiler would be a project in itself.

A more practical approach to testing comes in the form of *regression tests*. The key idea here is to write an increasing set of programs which will inevitably make use of different features of the language as well as of the compiler. Initially, these test cases help us to check that everything works as expected and can be thought of as simple unit tests. However, as we make changes to the compiler, we can rerun them to ensure that our changes did not break any of the programs. If we encounter a bug while we are using the compiler normally, we can add the respective program to the collection of test programs. The same approach is once again taken for most other compilers, including the Glasgow Haskell Compiler.

In order to ease this task, we write a small program which uses the compiler’s source and can be loaded into GHCi. We first define a data type to represent test cases:

```haskell
data Test = Test [FilePath] [String]
```

The list of values of type *FilePath* will store the list of source files for each test and the list of values of type *String* will store additional command line arguments, other than those which are set by the test environment:

```haskell
testConditions :: [String]
testConditions = ["-trace", "-keep-hs", "-use-stdout", "-path", ".///../lib/"
```

For example, a simple test for the infinite sequence of Fibonacci numbers may now be defined as the following:

```haskell
fibTest :: Test
fibTest = Test
[".//../regression/Fibonacci.cada"]
["-entry", "Fibonacci", ",o", ",fib.exe"]
```

We also define three functions to run tests, where *runTestWith* does all the work by invoking the compiler with the test settings. The other two functions, *runTest* and
A compiler for Cada

parseFile, merely exist for convenience:

\[
\text{runTestWith} :: \text{Test} \rightarrow [\text{String}] \rightarrow \text{IO} ()
\]
\[
\text{runTestWith} (\text{Test } xs \ ys) \ as = \ do
\]
\[
\text{withArgs} (xs + + ys + + as + + testConditions) \ \text{runCompiler}
\]
\[
\text{putStrLn} "\text{Test completed.}\"
\]

\[
\text{runTest} :: \text{Test} \rightarrow \text{IO} ()
\]
\[
\text{runTest} \ t = \ \text{runTestWith} \ t []
\]

\[
\text{parseFile} :: \text{Test} \rightarrow \text{IO} ()
\]
\[
\text{parseFile} \ t = \ \text{runTestWith} \ t ["\text{-stop-after-parser}\"]
\]

There are five “realistic” test programs, i.e. those which one might want write in practice to do something more or less useful. Some of these serve the second purpose of helping us to evaluate the success of our language design, which will return to in the next chapter. Additionally, there are several smaller test cases whose goal is to make use of a variety of language features in unlikely, yet possible arrangements.

Complementary to the approach we have taken, there are two libraries which are worth discussing at this point. HUnit is a framework for writing unit tests in Haskell, similar to those available for most other mainstream programming languages. However, given the size of our codebase, it becomes very difficult to provide adequate coverage of unit tests. QuickCheck (Claessen and Hughes, 2011) offers an alternative approach to testing our code which more closely resembles the idea of formal verification. If we specify properties of functions, QuickCheck will test them with a set of random values each time a test is run. While this does not guarantee correctness like a formal proof would, it is likely to eventually find an error if there is one. For our purposes, it is still too difficult to achieve good code coverage and thus suffers from the same issue as HUnit, albeit not quite as badly. Additionally, because QuickCheck is not guaranteed to find an error during the first time it is run, it may later not be clear whether an error has always been present or if it has been caused by a recent change. Thus we can conclude that regression tests are the right choice for this project at this time.
Conclusion

We set out to design and implement a modern, purely functional programming language which provides abstractions for the effect of mutable state. In this dissertation, we have first established the theoretical foundations required to understand why this work is beneficial to the software engineering community, by learning about the advantages of pure programs and the use of monads as abstractions for effects.

Using inspiration from other paradigms, we combined common patterns in which program state is manipulated with the state monad to yield a simple syntax for our goals. Based on this design and influences from languages such as Haskell, we then designed Cada, a purely functional language with non-strict semantics.

With the specification finished, half of our objectives were accomplished, and we then began to implement a compiler based on it. In the process, we explored the various components of a modern compiler, ranging from the lexical analysis and parser, over the type system, to the code generation. In the end, we produced a working compiler for our language which can be used for the development of real programs, as it is complete, able to help with the diagnosis of errors, and can be used to generate executables or reusable program libraries.

To measure our success, let us briefly consider a small monadic compiler\(^1\) as an example, where we make effective use of our new syntax to transform a part of the implementation, which could be written in a language like Haskell, into a shorter and more elegant version. In order to execute code generated by this compiler, we need an appropriate representation of the corresponding abstract machine, consisting of a stack as well as two types of memory for the instructions and the data.

There are two obvious approaches to implementing this in Haskell: firstly, we could construct a monad stack, using the monad transformer variants of the `Reader` and `State` monads as shown below:

\[
\begin{align*}
\text{type } Memory &= \text{Reader}(\text{State}(\text{Mem})) \\
\text{type } VM &= \text{State}(\text{Stack}(\text{Memory}))
\end{align*}
\]

\(^1\)Based on the specification of [http://www.cs.nott.ac.uk/~gsh/afpcwk2.pdf](http://www.cs.nott.ac.uk/~gsh/afpcwk2.pdf)
However, while this definition is in principle very elegant, it would require us to set up the monad stack first, make use of \texttt{lift} throughout our code, and would not be very accessible to programmers who have no experience with monad transformers.

Alternatively, a simpler solution would be to define a new data type with a single constructor and fields for all three components of the abstract machine. We could then use this type as argument for the state monad:

\begin{verbatim}
data VMData = VMData {
    vmData :: Mem,
    vmProgram :: Code,
    vmStack :: Stack
}

type VM = State VMData

g getData :: VM Mem
g getData = gets vmData

set Data :: Mem \rightarrow VM ()
set Data v = modify $ \lambda (VMData p s) \rightarrow VMData v p s

...
\end{verbatim}

In this case we have to define various helper functions to get or set the values of the individual fields in \texttt{VMData}, which is lengthy, repetitive and not very elegant.

In Cada, we could choose to implement either approach using our new syntax, but the definition of the second option is particularly nice since we can get the compiler to generate all getters and setters, as well as the type declaration for us:

\begin{verbatim}
state VM {
    vmData :: Mem;
    vmProgram :: Code;
    vmStack :: Stack;
}
\end{verbatim}

This piece of code in Cada is equivalent to the above example in Haskell, even though we omitted four more functions there. Here a programmer does not have to worry about any of the details of the implementation as neither the state monad, nor any of its accompanying helper functions are mentioned anywhere.
Additionally, we can make use of Cada’s extended do-notation to easily access the getters and setters which were automatically generated for us:

\[
push :: Int \rightarrow VM();
push v = \{ 
    vs <: vmData;
    (v : vs) :> vmData;
\}
\]

It is evident that the implementation in Cada is much shorter than the equivalent one in Haskell. Indeed, it also does not require any knowledge of the state monad, other than understanding of the idea of mutable state, which may have previously been obtained from imperative or object-oriented programming languages. Our syntax also resembles that of such languages, thus potentially easing the transition from those paradigms to functional programming.

However, while the effect of mutable state is incredibly easy to use in Cada, our new notations can be used together with the other parts of the language, thus remaining true to the idea in functional programming of having many small building blocks which can be used to construct programs in powerful ways. We can conclude that our example demonstrates the successful outcome of our objectives. Subject to the availability of the necessary functions in the standard library, it should even be possible to implement a compiler for Cada in Cada.

### 6.1 Evaluation of the implementation

As with all software projects, the implementation is not perfect. In this section we will discuss the lessons learnt during the construction of Cada’s compiler and from using Haskell for a relatively large project. From an architectural point of view, it seems to be easy to fall into the trap of reinventing the wheel sometimes, if different modules are constructed from the bottom up. For example, there is currently no consistent system for error messages as `Maybe`, `Either`, and `ErrorT` are all used in different incarnations throughout the code and may provide information ranging from type `String` to more sophisticated types such as `TypeError`. The lesson learnt here is that one should design a system in advance, which can then be used coherently by different subsystems.

The structure of Happy’s grammars seems to invoke the urge that abstract representations must be returned by semantic actions, even when this may not be the best thing to do. For example, the current abstract syntax tree consists of a list of declarations, even though they are not all needed in one list. Having two lists, one for type and one for value declarations, may be much more beneficial because there would be no
need to filter them when only one subset is required. However, this would have to be constructed with the help of a state monad as we cannot make any assumptions about the ordering of type and value declarations in a source file. Luckily, it should be fairly easy to make changes to the parser to accommodate such a distinction.

Some parts of the type system, such as kind inference and type classes, are badly documented in literature and thus made the implementation of the respective components more difficult than it should have been. For example, to the author’s knowledge there is no documentation of the type demotion rules required for kind inference, and the only accessible implementation can be found in the Haskell Type Extensions library\(^2\). However, HTE appears to be incomplete in parts and also accommodates various type system extensions which affect kind inference, but are not part of Haskell98, such as constraint kinds. The author hopes that the contents of this dissertation can be used as a reference for this process in the future.

During the implementation, some interesting bugs have shown shortcomings of the specification which then needed to be addressed. For a particularly interesting example, suppose that we wish to give a state declaration where the parent monad is a type variable:

```plaintext
state Test m : m
{
    foo :: Int
}
```

After the syntactic sugar has been removed, we will end up with the following two definitions for a data type and type function as we would expect:

```plaintext
data TestData t0 = Test Int
type Test t0 = StateT (TestData t0) t0
```

In the current implementation, this compiles without issues, although there is actually a subtle error. The kind inference procedure sorts type declarations by dependencies, such that the kind of `TestData` will be inferred before the kind of `Test` can be inferred, as it depends on the previous result. Kind inference will conclude that `TestData` has kind \(\forall a. a \rightarrow \star\). This result may look strange as we do not have polymorphic kinds, but is actually not unexpected as kind variables are simply instantiated with \(\star\) later on. However, in the implementation this takes place after all kinds have been inferred, which means that kind inference will succeed for `Test` because it can instantiate \(a\) with \(\star \rightarrow \star\). Of course, this is incorrect as we will later assign \(\star \rightarrow \star\) to `TestData`.

\(^2\)https://github.com/shayan-najd/HTE/
Fixing the bug in the kind inference process is easy, as we will simply have to instantiate variables with $\star$ at the end of each binding group. However, we should also make a change to the translation process, such that $m$ is guaranteed to be of kind $\star \rightarrow \star$ through, for example, the addition of a kind signature.

6.2 Future work

While we have accomplished everything we set out to do, there are several improvements or additions which can be made on top of our work. Some of these are related to the usability of the compiler, which was not a key issue for us, but become relevant if this project is to be developed further.

6.2.1 Constructors and modifiers

There are two straightforward ways in which our syntax could be extended. During testing of our language, we observed that the following pattern occurred repeatedly within statement blocks, for some data constructor $T$ and a list of arguments $\vec{x}$:

```
  evalStateT { 
    ... 
  } (T \vec{x})
```

Typically, the list of arguments consists of base values such as the empty list or 0, depending on the exact types. It should be possible to borrow the concept of constructors or copy constructors from object-oriented languages to provide a way of specifying the default values of fields in a state declaration.

We also noticed that the two new statement types would often occur in sequence to first retrieve data from a field and to then update it. For example, consider the definition of `push` in the previous example, where we first obtain the current stack, add one element to it, and then store it. It would beneficial to have a single statement which does all of this at once if we specify e.g. $(\cdot) \, v$. A suitable function might be the following variation of `modify`, which additionally returns the old state:

```
  query :: Monad m \Rightarrow (s \rightarrow s) \rightarrow StateT s m s
  query f = get >>= \lambda v \rightarrow put (f v) >>= return v
```

6.2.2 Generalisation

Currently, our notation represents the `State` and `StateT` monads as well as their associated functions. In theory, there are two ways in which this could be generalised to cover a wider range of state monads. Firstly, Haskell’s `mtl` library contains a type class named `MonadState` which could be used to provide implementations of
the \textit{gets} and \textit{modify} functions, thus allowing any instance of this class to be represented by our syntax. However, \textit{MonadState} requires multi-parameter type classes (Peyton Jones et al., 1997) and functional dependencies (Jones, 2000; Sulzmann et al., 2007), neither of which are currently part of Cada.

Secondly, there is an indexed variation of the state monad, where the type of the state may change in a state transformer. In other words, we have the following definition:

\begin{verbatim}
newtype IxStateT i s m a = IxStateT { runIxStateT :: i \rightarrow m (a, s) }
\end{verbatim}

Adding slight changes to our syntax would allow the parameters of the underlying data type to be indexed, thus allowing more interesting applications. For example, a compiler’s representation of a module may have different annotations for the abstract syntax tree throughout the compilation process. Using an indexed state monad, one could run all computations inside the same monad, even if the type of the AST is required to change. Sadly, it is not possible to make \textit{IxStateT} an instance of \textit{Monad}, because we cannot show that \textit{i} and \textit{s} are equivalent in the definition of \textit{return}.

### 6.2.3 Different type inference algorithm

In our current implementation, type inference is based on Algorithm $\mathcal{W}$, but as we have discussed in section 5.11 there exist solutions which have more desirable properties, such as the technique presented in Heeren et al. (2002) which produces better error messages. An implementation of such an algorithm would increase the usability of our compiler, if this should be desired. However, it would also involve having to rewrite the corresponding part of the implementation of the type system.

It might also be interesting to combine this approach with partial type inference for higher-rank polymorphism as shown in Peyton Jones et al. (2007) if this extension to the type system is required, but the author is not aware of any attempts to accomplish this. Additionally, it is unclear whether the constraint-based type inference algorithm would be compatible with type classes.

### 6.2.4 Language extension for Haskell

Constructing a compiler for a general purpose language from scratch is generally not a good idea unless it differs significantly from existing systems, because of the vast amount of work that is required. Cada is very similar to Haskell and, instead of developing this compiler further to include many of the features which are already available in, for example, the Glasgow Haskell Compiler, it may be more sensible to implement the syntax we have presented here as a language extension for Haskell.


A compiler for Cada


Mac Lane, S. *Categories for the working mathematician*, volume 5. springer, 1998.


APPENDIX

A.1 System F

\[ \begin{array}{ll}
\tau & \to \alpha \mid \tau \to \tau \mid \forall \alpha.\tau \quad \text{Types} \\
e & \to x \quad \text{Variable} \\
| \lambda x : \tau.e & \text{Term abstraction} \\
| e e & \text{Term application} \\
| \Lambda \alpha.e & \text{Type abstraction} \\
| e \tau & \text{Type application}
\end{array} \]

Figure A.1: System F syntax

\[
\begin{array}{ll}
\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} & \text{TVAR} \\
\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x : \sigma.e : \sigma \to \tau} & \text{TABS} \\
\frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1 \ e_2 : \tau} & \text{TAPP} \\
\frac{\Gamma \vdash e : \tau \quad \alpha \not\in TV(\Gamma)}{\Gamma \vdash \Lambda \alpha.e : \forall \alpha.\tau} & \text{TTYABS} \\
\frac{\Gamma \vdash e : \forall \alpha.\tau}{\Gamma \vdash e \sigma : [\alpha \mapsto \sigma]\tau} & \text{TTTYAPP}
\end{array}
\]

Figure A.2: Typing judgements for System F
A compiler for Cada

A.2 Hindley-Milner

\[
\begin{align*}
\tau & \rightarrow \alpha | T \bar{\tau} \quad \text{Mono types} \\
\sigma & \rightarrow \tau | \forall \alpha. \sigma \quad \text{Poly types} \\
e & \rightarrow x \quad \text{Variable} \\
& | \lambda x.e \quad \text{Abstraction} \\
& | e \ e \quad \text{Application} \\
& | \textbf{let} x = e \textbf{ in } e \quad \text{Let-binding}
\end{align*}
\]

\[
\begin{align*}
\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad \text{TVAR} & \quad \frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad \text{TINST} \\
\frac{\Gamma \vdash e : \tau' \rightarrow \tau \quad \Gamma \vdash e' : \tau'}{\Gamma \vdash e \ e' : \tau} \quad \text{TAPP} \\
\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \rightarrow \tau'} \quad \text{TABS} & \quad \frac{\Gamma \vdash e : \sigma \quad \Gamma, x : \sigma \vdash e' : \tau}{\Gamma \vdash \textbf{let} x = e \textbf{ in } e' : \tau} \quad \text{TLET} \\
\frac{\Gamma \vdash e : \sigma \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall \alpha. \sigma} \quad \text{TGEN}
\end{align*}
\]

Figure A.3: Hindley-Milner

\[
\tau' = [\alpha_i \mapsto \tau_i] \tau \quad \beta_j \notin \text{free}(\forall \alpha_1 \ldots \forall \alpha_m. \tau) \\
\forall \alpha_1 \ldots \forall \alpha_m. \tau \subseteq \forall \beta_1 \ldots \forall \beta_n. \tau'
\]

Figure A.4: Specialisation rule
\[ W(\Gamma, x) = (\emptyset, \text{inst}(\sigma)) \text{ where } (x : \sigma) \in \Gamma \]

\[ W(\Gamma, e_1 e_2) = \text{let } (\theta_1, \tau_1) = W(\Gamma, e_1) \]
\[ \quad (\theta_2, \tau_2) = W(\theta \Gamma, e_2) \]
\[ \quad \theta_3 = U(\theta_2 \tau_1, \tau_2 \rightarrow \tau) \text{ where } \tau \text{ is fresh} \]
\[ \quad \text{in } (\theta_3 \theta_2 \theta_1, \theta_3 \tau) \]

\[ W(\Gamma, \lambda x. e_1) = \text{let } (\theta_1, \tau_1) = W(\Gamma \cup \{ x : \tau \}, e_1) \text{ where } \tau \text{ is fresh} \]
\[ \quad \text{in } (\theta_1, \theta_1(\tau \rightarrow \tau_1)) \]

\[ W(\Gamma, \text{let } x = e_1 \text{ in } e_2) = \text{let } (\theta_1, \tau_1) = W(\Gamma, e_1) \]
\[ \quad (\theta_2, \tau_2) = W(\theta_1 \Gamma \cup \{ x : \text{gen}(\theta_1 \Gamma, \tau_1) \}, e_2) \]
\[ \quad \text{in } (\theta_2 \theta_1, \tau_2) \]

\[ \text{gen}(\Gamma, \tau) = \forall \vec{\alpha}. \tau \text{ where } \vec{\alpha} = \text{free}(\tau) - \text{free}(\Gamma) \]

\[ \text{inst}(\forall \alpha_1 \ldots \alpha_n. \tau) = \{ \alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n \} \tau \text{ where } \beta_1, \ldots, \beta_n \text{ are fresh} \]

Figure A.5: Algorithm \( W \)
A.3 Lexical grammar

\[
\begin{align*}
\text{digit} & \rightarrow 0 \mid 1 \mid \ldots \mid 9 \\
\text{lower} & \rightarrow a \mid b \mid \ldots \mid z \\
\text{upper} & \rightarrow A \mid B \mid \ldots \mid Z \\
\text{symbol} & \rightarrow \! | \# | \$ | \% | \& | \* | + \mid . \mid \\
 & \mid / \mid < \mid = \mid > \mid ? \mid @ \mid \backslash \mid ^ \mid \\
 & \mid \mid \mid ^ ~ \\
\text{special} & \rightarrow ( \mid ) \mid , \mid ; \mid [ \mid ] \mid ' \mid \{ \mid \} \\
\text{graphic} & \rightarrow \text{lower} \mid \text{upper} \mid \text{symbol} \mid \text{digit} \mid \text{special} \mid : \mid \" \mid \'
\end{align*}
\]

\[
\begin{align*}
\text{whitechar} & \rightarrow \text{NL} \mid \text{VT} \mid \text{TAB} \mid \text{Space} \\
\text{slany} & \rightarrow \text{graphic} \mid \text{Space} \mid \text{TAB} \\
\text{mlany} & \rightarrow \text{graphic} \mid \text{whitechar} \\
\text{newline} & \rightarrow \text{CRLF} \mid \text{CR} \mid \text{LF} \mid \text{FF} \\
\text{comment} & \rightarrow \-- (\text{slany} \mid \text{symbol})^* \text{newline} \\
\text{opcomment} & \rightarrow \{ - \\
\text{clcomment} & \rightarrow \} \\
\text{mcomment} & \rightarrow \text{opcomment} (\text{mlany} \mid \text{clcomment})^* \text{clcomment} \\
\text{whitespace} & \rightarrow (\text{comment} \mid \text{mcomment} \mid \text{whitechar})^* \\
\text{literal} & \rightarrow \text{integer} \mid \text{string} \\
\text{token} & \rightarrow \text{qvar} \mid \text{qctr} \mid \text{varsym} \mid \\
 & \rightarrow \text{literal} \mid \text{special} \mid \text{rop} \mid \text{rid} \\
\text{program} & \rightarrow (\text{whitespace} \mid \text{token})^*
\end{align*}
\]

Figure A.6: Lexical program structure
\[ \text{var} \rightarrow (\text{lower} \mid \text{upper} \mid \text{digit} \mid ')^* - \text{rid} \\
\text{ctr} \rightarrow \text{upper} (\text{lower} \mid \text{upper} \mid \text{digit} \mid ')^* \\
\text{rid} \rightarrow \text{case} \mid \text{class} \mid \text{data} \mid \text{else} \mid \text{enum} \mid \\
\text{if} \mid \text{import} \mid \text{in} \mid \text{instance} \mid \text{let} \mid \\
\text{module} \mid \text{newtype} \mid \text{state} \mid \text{then} \mid \text{type} \\
\text{varsym} \rightarrow \text{symbol} (\text{symbol} \mid :)^* \\
\text{rop} \rightarrow .. | : | :: | = | \| | l | <- | - > | => | <: | : >: \\
\]

Figure A.7: Identifiers

\[ q\text{var} \rightarrow (\text{ctr} \ .)^* (\text{var} \ .)^* \text{var} \\
q\text{ctr} \rightarrow (\text{ctr} \ .)^* \text{ctr} \\
\]

Figure A.8: Qualified identifiers

\[ \text{integer} \rightarrow \text{digit}^+ \\
\text{string} \rightarrow " ((\text{graphic} - (" \mid \ \)) \mid \text{whitespace} \mid \text{escape})^* " \\
\text{escape} \rightarrow \ \\backslash(a \mid b \mid f \mid n \mid r \mid t \mid v \mid \ | " | ', | \&) \\
\]

Figure A.9: Literals
A.4 Context-free grammar

\[
\begin{align*}
\text{Module} & \rightarrow \text{module qctr \{ Declaration* \}} \\
\text{Declaration} & \rightarrow \text{import qctr;} \\
& | \text{type ctr \textit{var}^* = Type}_0; \\
& | \text{class TypeClass;} \\
& | \text{instance Instance;} \\
& | \text{DataType} \\
& | \text{State} \\
& | \text{Typing} \\
& | \text{ValDecl}
\end{align*}
\]

Figure A.10: Grammar of modules and declarations

\[
\begin{align*}
\text{Op} & \rightarrow \text{‘qvar’} \\
& | \text{varsym} \\
& | : 
\end{align*}
\]

Figure A.11: Grammar of operators

\[
\begin{align*}
\text{Equation} & \rightarrow \text{Name Pattern}^* = \text{Expr}_0; \\
& | \text{Pattern varsym Pattern}^* = \text{Expr}_0; 
\end{align*}
\]

Figure A.12: Grammar of equations
\[
\text{Literal} \quad \rightarrow \quad \text{integer} \\
\quad \quad | \quad \text{string} \\
\quad \quad | \quad \text{ListLiteral}
\]

\[
\text{ListLiteral} \quad \rightarrow \quad [\text{PEexpr}_0]
\]

Figure A.13: Grammar of literals

\[
\text{Pattern} \quad \rightarrow \quad qvar \\
\quad \quad | \quad qctr \\
\quad \quad | \quad (\text{PPattern}) \\
\quad \quad | \quad \text{Literal} \\
\quad \quad | \quad -
\]

\[
\text{PPattern} \quad \rightarrow \quad \text{PPattern}_0 \\
\quad \quad \quad | \quad \text{PPattern}_2 : \text{PPattern}_2
\]

\[
\text{PPattern}_0 \quad \rightarrow \quad \epsilon \\
\quad \quad \quad | \quad \text{PPattern}_1
\]

\[
\text{PPattern}_1 \quad \rightarrow \quad \text{PPattern}_2 \\
\quad \quad \quad | \quad \text{PPattern}_1 : \text{PPattern}_2
\]

\[
\text{PPattern}_2 \quad \rightarrow \quad qvar \\
\quad \quad \quad | \quad qctr \ \text{Pattern}^* \\
\quad \quad \quad | \quad (\text{PPattern}) \\
\quad \quad \quad | \quad \text{Literal} \\
\quad \quad \quad | \quad -
\]

Figure A.14: Grammar of patterns
Figure A.15: Grammar of types

\[
\begin{align*}
\text{Type}_0 & \rightarrow \text{Type}_1 \rightarrow \text{Type}_0 \\
& \quad \mid \text{Type}_1 \\
\text{Type}_1 & \rightarrow \text{Type}_1 \text{ Type}_2 \\
& \quad \mid \text{Type}_2 \\
\text{Type}_2 & \rightarrow \text{var} \\
& \quad \mid (\text{Type}_0, \text{Type}_p) \\
& \quad \mid [\text{Type}_0] \\
& \quad \mid [] \\
& \quad \mid (_,) \\
& \quad \mid (\text{Type}_0) \\
& \quad \mid \text{Type}_3 \\
\text{Type}_3 & \rightarrow () \\
& \quad \mid \text{ctr} \\
\text{Type}_p & \rightarrow \text{Type}_0 \\
& \quad \mid \text{Type}_p, \text{Type}_0
\end{align*}
\]

Figure A.16: Grammar of typings

\[
\begin{align*}
\text{Name} & \rightarrow \text{var} \\
& \quad \mid (\text{varsym}) \\
\text{Typing} & \rightarrow \text{Name}^* :: (\epsilon \mid \text{Type}_1 \Rightarrow \text{Type}_0);
\end{align*}
\]
\( \text{TypeClass} \rightarrow (\varepsilon \mid \text{Type}_1 \Rightarrow \text{Type}_1 \{ \text{Typing}^* \}) \)

\( \text{Instance} \rightarrow (\varepsilon \mid \text{Type}_1) \Rightarrow \text{Type}_1 \{ (\text{Typing} \mid \text{Equation})^* \} \)

Figure A.17: Grammar of type classes and instances

\[
\begin{align*}
\text{CtrList} & \rightarrow \ ctr,\text{CtrList} \\
& \mid \ ctr \\
\text{Field} & \rightarrow \ \text{var} :: \text{Type}_0 \\
\text{Constructors} & \rightarrow \ \text{ctr} \\
& \mid \ \text{ctr} \{ \text{Field}^* \} \\
& \mid \ \text{ctr},\text{Constructors} \\
& \mid \ \text{ctr} \{ \text{Field}^* \} \text{Constructors} \\
\text{DataType} & \rightarrow \ \text{enum} \ ctr \{ \text{CtrList} \} \\
& \mid \ \text{data} \ ctr \ \text{var}^* \{ \text{Field}^* \} \\
& \mid \ \text{data} \ ctr \ \text{var}^* \{ \text{Constructors} \} \\
& \mid \ \text{newtype} \ ctr \ \text{var}^* = \ ctr \{ \text{Field} \}
\end{align*}
\]

Figure A.18: Grammar of algebraic data types

\[
\begin{align*}
\text{Parent} & \rightarrow \ \varepsilon \\
& \mid \ : \ \text{Type}_1 \\
\text{State} & \rightarrow \ \text{state} \ ctr \ \text{var}^* \ \text{Parent} \{ \text{Field}^* \}
\end{align*}
\]

Figure A.19: Grammar of state declarations
\[
\begin{align*}
\text{Alt} & \quad \rightarrow \quad \text{Pattern}^* = \text{Expr}_0 \\
\text{Alts} & \quad \rightarrow \quad \text{Alt} \\
& \quad \quad | \quad \text{Alts}; \text{Alt} \\
\text{Option} & \quad \rightarrow \quad \text{Pattern} \rightarrow \text{Expr}_0; \\
\text{Statement} & \quad \rightarrow \quad \text{Expr}_0; \\
& \quad \quad | \quad \text{Expr}_2 \leftarrow \text{Expr}_0; \\
& \quad \quad | \quad \text{Expr}_2 <: \text{qvar}; \\
& \quad \quad | \quad \text{Expr}_0 >: \text{qvar}; \\
\text{Expr}_0 & \quad \rightarrow \quad \text{Expr}_1 \\
& \quad \quad | \quad \text{Expr}_0 \text{ Op } \text{Expr}_1 \\
\text{Expr}_1 & \quad \rightarrow \quad \lambda \text{Pattern}^* \rightarrow \text{Expr}_0 \\
& \quad \quad | \quad \text{case } \text{Expr}_0 \text{ of } \{ \text{Option}^* \} \\
& \quad \quad | \quad \text{if } \text{Expr}_0 \text{ then } \text{Expr}_0 \text{ else } \text{Expr}_0 \\
& \quad \quad | \quad \text{let } \text{Alts in } \text{Expr}_0 \\
& \quad \quad | \quad \text{Expr}_2 \\
\text{Expr}_2 & \quad \rightarrow \quad \text{Expr}_2 \text{ Atom} \\
& \quad \quad | \quad \text{Atom} \\
\text{Atom} & \quad \rightarrow \quad \text{qvar} \\
& \quad \quad | \quad \text{qctr} \\
& \quad \quad | \quad \{ \text{Statement}^* \} \\
& \quad \quad | \quad \text{Literal} \\
& \quad \quad | \quad (\text{PEExpr}_0) \\
\text{PEExpr}_0 & \quad \rightarrow \quad \epsilon \\
& \quad \quad | \quad \text{PEExpr}_1 \\
& \quad \quad | \quad \text{varsym} \\
\text{PEExpr}_1 & \quad \rightarrow \quad \text{Expr}_0 \\
& \quad \quad | \quad \text{PEExpr}_1, \text{Expr}_0 \\
\end{align*}
\]

Figure A.20: Grammar of expressions
### A.5 Abstract syntax tree

*Figure A.21: Abstract representation of surface types*

```haskell
-- type
type TypeName = String

-- data
data TypeConstraint = TyConstr TypeName String

data TypeParam = TyP String AlexPosn

data SType = STyVar String
  | STyCtr String
  | STyApp SType SType
  | STyTuple [SType]
  | STyList SType
  deriving (Eq, Ord)

data TyQual t = [TypeConstraint] :-= > t

data TyScheme = Scheme [String] (TyQual SType)
```
type ModuleName  =  String

data Module a  =  Module ModuleName [RecA a Definition]

data Definition a  =  ImportDef ModuleName
| TypeDef AliasDefinition
| TyClDef (STypeClass a)
| InstDef (SInstance a)
| DataDef DataDefinition
| StateDef StateDefinition
| TypeDec DecType
| ValueDef Equation

Figure A.22: Abstract representation of modules

data DecType  =  DecTy {
    defSigNames  ::  [String],
    defSigType   ::  TyScheme
}

data Alt  =  Alt {
    altPats  ::  [Pattern],
    altExpr  ::  Expr
}

data Equation  =  Eq {
    eqName  ::  String,
    eqAlt   ::  Alt
}

Figure A.23: Abstract representation of typings and equations
\textbf{data Expr} = Var String \\
| Ctr String \\
| Lit Literal \\
| Abs Alt \\
| App Expr Expr \\
| InfixOp String Expr Expr \\
| Let [Alt] Expr \\
| Cond Expr Expr Expr \\
| Case Expr [Option] \\
| Do [Statement]

\textbf{data Literal} = UnitLit \\
| IntLit String \\
| StrLit String \\
| PairLit [Expr] \\
| ListLit [Expr]

\textbf{data Option} = Option \{ \\
\quad optPattern :: Pattern, \\
\quad optExpr :: Expr \\
\}

\textbf{data Statement} = Statement Expr \\
| Bind Pattern Expr \\
| Getter Pattern String \\
| Setter Expr String

\textbf{data Pattern} = VarPattern String \\
| LitPattern Literal \\
| CtrPattern String [Pattern] \\
| Wildcard

Figure A.24: Abstract representation of expressions
**data** AliasDefinition = ADef { 
  aDefName :: TypeName,
  aDefParams :: [TypeParam],
  aDefType :: SType
}

Figure A.25: Abstract representation of type functions

**data** STypeClass a = TyClass { 
  tyClConstr :: [TypeConstraint],
  tyClName :: String,
  tyClParams :: [String],
  tyClDefs :: [a DecType]
}

**data** InstDef a = InstTyDef (a DecType) 
  | InstValDef Equation

**data** SInstance a = SInst { 
  instConstrs :: [TypeConstraint],
  instClass :: String,
  instParam :: SType,
  instBody :: [InstDef a]
}

Figure A.26: Abstract representation of type classes
data DataField = DField {
  dFieldName :: String,
  dFieldType :: SType,
  dFieldPos :: AlexPosn
}

data DataConstructor = DCtr {
  dCtrName :: String,
  dCtrFields :: [DataField]
}

data DataDefType = Data | Single | Enum
                 | Newtype | State

data DataDefinition = DDef {
  dDefType :: DataDefType,
  dDefName :: String,
  dDefParams :: [TypeParam],
  dDefCtrs :: [Loc (Typed DataConstructor)],
  dDefProjs :: [Equation]
}

data StateDefinition = SDef {
  sDefName :: String,
  sDefParams :: [TypeParam],
  sDefParent :: Maybe SType,
  sDefCtrs :: [DataField],
  sDefType :: AliasDefinition,
  sDefData :: DataDefinition,
  sDefAcs :: [Equation]
}

Figure A.27: Abstract representation of algebraic data types
A.6 Type system

\[
\textbf{data} \quad \text{Kind} \quad = \quad \text{KStar} \\
\quad \quad | \quad \text{KFun} \text{ Kind} \text{ Kind} \\
\quad \quad \quad \text{deriving} \quad (\text{Eq}, \text{Ord}) \\
\]

\[
\textbf{data} \quad \text{TyVar} \quad = \quad \text{TyVar} \text{ String} \text{ Kind} \text{ deriving} \quad \text{Eq} \\
\textbf{data} \quad \text{TyCtr} \quad = \quad \text{TyCtr} \text{ String} \text{ Kind} \text{ deriving} \quad (\text{Eq}, \text{Ord}) \\
\textbf{data} \quad \text{TyFun} \quad = \quad \text{TyFun} \text{ String} \text{ Kind} \text{ deriving} \quad (\text{Eq}, \text{Ord}) \\
\]

\[
\textbf{data} \quad \text{MonoType} \quad = \quad \text{TVar} \text{ TyVar} \\
\quad \quad | \quad \text{TCtr} \text{ TyCtr} \\
\quad \quad | \quad \text{TFun} \text{ TyFun} \text{ [MonoType]} \\
\quad \quad | \quad \text{TApp} \text{ MonoType} \text{ MonoType} \\
\quad \quad | \quad \text{TGen} \text{ Int} \\
\]

\[
\textbf{data} \quad \text{Constr} \quad = \quad \text{In} \text{ String} \text{ MonoType} \\
\quad \quad \quad \text{deriving} \quad (\text{Eq}, \text{Ord}) \\
\]

\[
\textbf{type} \quad \text{Context} \quad = \quad \text{S.Set} \text{ Constr} \\
\]

\[
\textbf{data} \quad \text{Qual} \ t \quad = \quad \text{Context} :=> \ t \text{ deriving} \quad \text{Eq} \\
\]

\[
\textbf{data} \quad \text{PolyType} \quad = \quad \text{ForAll} \ [\text{Kind}] \ (\text{Qual} \ \text{MonoType}) \\
\]

Figure A.28: Kinds, contexts, monomorphic and polymorphic types
data Alias = Alias {
    aliasKind :: Kind,
    aliasArity :: Int,
    aliasTyype :: PolyType
}

Figure A.29: Type functions

data ADT = ADT {
    adtKind :: Kind,
    adtCtrs :: Assumps
}

Figure A.30: Algebraic data types

data Instance = Inst {
    instType :: Qual MonoType,
    instDict :: [Equation]
}

Figure A.31: Instances

data TypeClass = TypeClass {
    clSupers :: Context,
    clKind :: Kind,
    clAssumps :: Assumps
}

Figure A.32: Type classes