Dissertation Type: research

DEPARTMENT OF COMPUTER SCIENCE

Programming Language Theory Applications

for Verified Cryptography in EasyCrypt

Nashe Mncube

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Friday 24th April, 2020

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Nashe Mncube, Friday 24th April, 2020

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

List of Definitions

Mitigation Table

Event/Issue	Potential/actual Impact on project	Action(s) taken to mitigate impact on project outcomes	Remaining impact

Table 1: Mitigation Table

Executive Summary

Cryptography is a field of research which can be summarised by the goal of how to ensure privacy, integrity and authenticity when communicating information between multiple parties. To achieve successful results in this field, researchers(primarily mathematicians) construct schemes using mathematics which aim to achieve some if not all of the goals mentioned. The construction of these schemes typically involves constructing lengthy proofs of certain security goals by hand which becomes harder with growing scheme complexity.

Machine verified cryptography is an area of research which seeks to aid some of the difficulty of scheme construction and proofs by allowing for cryptographers to construct their schemes using proof assistants. These allow for verified assurances of security using formal methods and automation of theorem proving. EasyCrypt is one such proof assistant [16] that started being developed in 2009 by the IMDEA software institute and Inria and now also École Polytechnique. It is self described as the following by the authors:

> EasyCrypt is a toolset for reasoning about relational properties of probabilistic computations with adversarial code. Its main application is the construction and verification of game-based cryptographic proofs.

My project focused on applying a concept from programming language theory known as type classes [25] to allow for easier construction of mathematical objects for use in cryptographic proofs. I achieved the following for this dissertation.

- I learnt the theory behind proof assistants including EasyCrypt,
- I learnt the relevant programming language theory as it relates to the problem,
- I became familiar with the EasyCrypt compiler,
- I implemented multi-parameter type classes in EasyCrypt through modifications to said compiler.

Supporting Technologies

This project makes use of the following technologies:

- The EasyCrypt proof assistant version 1.0 [16] as the target language for my work,
- the OCaml programming language version 4.09.0 which the EasyCrypt compiler is implemented in [19] and therefore my work was implemented in.

Notation and Acronyms

PLT	:	Programming Language Theory
PLD	:	Programming Language Design
ATP	:	Automated Theorem Proving
DHM	:	the Damas-Hindley-Milner type system
pRHL	:	the probabilistic relational Hoare logic
pHL	:	the probabilistic Hoare logic
aHOL	:	the ambient higher order logic
$\lambda\!\rightarrow$	:	Simply Typed λ-calculus
$\lambda 2$	:	Polymorphic λ-calculus a.k.a System-F
$\mathcal{O}(n)$	:	Big-O complexity of order n
$e ::= \epsilon \mid Ae$	:	Backus-Naur form expression definition notation
$\Gamma \vdash M : S$	:	type declaration S to term M with set of type assumptions Γ
$Pr[x = X]$	:	Probability of some random variable X equaling x
$\bigcup_{i=1}^{n} S_i$	:	Union of n sets S_i
$k \xleftarrow{\$} \mathcal{K}$	:	Random sampling of variable k from set $\mathcal{K}$
$\exists x \frac{\phi_1 \dots \phi_n}{\phi}(Name)$	:	Natural deduction inference rule with propositions ϕ_i and judgement ϕ, side condition on left and optional label on right

Chapter 1

Contextual Background

This chapter seeks to introduce the ideas that underpin automated theorem proving and it's applications to cryptography. Although not wholly descriptive, it seeks to act as a good overview of the core ideas necessary to understand the problem context. I will briefly explain the relevant literature. I will also outline the problem itself and its importance. It should be stated that the areas of research touched upon are rich and cannot be fully understood in a brief chapter, and the following seeks to link the relevant ideas on a higher level.

1.1 Theory of Computation

Although primarily considered a means to an end in computer science circles, that end being the utilisation of computers, programming languages are an exciting area of research. The focus of research into programming languages ranges from trying to extract the most efficient performance on target hardware to abstract ideas around language design. When relating PLT to ATP we typically are discussing how we can construct and reason about formal proofs of correctness in a programming language, although the reason that we can do this requires exposition. The following seeks to provide that exposition by highlighting the relationship between computation and mathematics.

1.1.1 Computation is an Algorithm Evaluated: Turing Machines

In the 20$^{\text{th}}$ century theoretical computer scientists and mathematicians were considering the theory of computation. Computation at the time was considered to be the "effective" solving of some problem given some laws or axioms to operate by. In 1936, Alan Turing constructed his now famous model of computation called the Turing machine [24].

Definition 1.1.1: Turing Machine

A Turing Machine is a 7-tuple $M = \langle Q, \Gamma, b, \Sigma, \delta, q_0, F \rangle$

- Q is a finite, non-empty set of states,
- Γ is a finite, non-empty set of tape alphabet symbols,
- $b \in \Gamma$ is the blank symbol (the only symbol allowed to occur on the tape infinitely often at any step during the computation),
- $\Sigma \subseteq \Gamma \setminus \{b\}$ is the set of input symbols, that is, the set of symbols allowed to appear in the initial tape contents,
- q_0 is the initial state,
- $F \subseteq Q$ is the set of final states or accepting states. The initial tape contents is said to be accepted by M if it eventually halts in a state from F.
- $\delta : (Q \setminus F) \times \Gamma \nrightarrow Q \times \Gamma \times \{L, R\}$ is a partial function called the transition function, where L is left shift, R is right shift. If δ is not defined on the current state and the current tape symbol, then the machine halts.

The key takeaway from the model of the Turing machine is that computation on a machine could be described as providing a program (or algorithm) that would define the state transitions of the machine. This directly relates to the purpose of a programming language for computers, but in abstract terms where we consider a programming language as a way to write such algorithms for some machine.

1.1.2 The λ-calculus

One of the most influential results of this era of research was the λ-calculus [8] proposed by Alonzo Church in 1936. He wished to construct a model of computation that would allow for defining certain problems in number theory which required defining some function over n positive integers e.g. $f(x_1, ..., x_n) = 2^z$.

Definition 1.1.2: Un-typed λ-calculus

The λ-calculus consists of inductively defined terms and rules for computation.
(Terms) a set, written Λ, is defined over the alphabet $\mathbb{V} + \{\lambda, ., (,)\}$ with the following inductive rules.

$$x \in \mathbb{V} \frac{}{x \in \Lambda} \text{ (Var)} \qquad \frac{M \in \Lambda, N \in \Lambda}{(MN) \in \Lambda} \text{ (App)} \qquad x \in \mathbb{V} \frac{M \in \Lambda}{(\lambda x.M) \in \Lambda} \text{ (Abs)}$$

$(\rightarrow_\beta)$ a function on the set of terms which represents the notion of computation, pronounced β-reduction, with $M, N \in \Lambda$

$$(\lambda x.M)N \rightarrow_\beta M[N/x]$$

where $(\lambda x.M)N$ is called our β-redex and $M[N/x]$ our contraction. $M[N/x]$ means that for all occurrence's of variable x in term M we replace with term N. $\rightarrow_\beta$ therefore defines a rewrite step, where our state of computation is represented as a syntactic expression.

The essential takeaway of Church's work was that if a function is λ-definable(can be defined with the λ-calculus) it is effectively computable.

Definition 1.1.3: Effectively Computable

Let $f : \mathbb{N}^k \rightarrow \mathbb{N}$ be a function from the set of k-tuples of natural numbers to the set of natural numbers, then a function f is said to be effectively computable if there's an algorithm that correctly computes f.

The un-typed λ-calculus can be considered a programming language where operations are performed over the set of variables $\mathbb{V}$. This is not particularly expressive as we may want to consider operations over a set of variables that have a type assignment: as can be seen in modern programming languages. This was the motivation for describing an extension of the λ calculus called the the simply typed λ-calculus, written $\lambda\rightarrow$, proposed by Church in 1940 where terms are allowed to depend on terms, and terms can be assigned a type from a set of type schemes. [9].

Definition 1.1.4: Simply typed λ-calculus

The simply typed λ-calculus, written as $\lambda\rightarrow$, adds the following rules to the λ-calculus

$$x : \sigma \in \Gamma \frac{}{\Gamma \vdash x : \sigma} \text{ (Var)} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \rightarrow \tau} \text{ (Abs)}$$

$$\frac{\Gamma \vdash \lambda x.M : \sigma \rightarrow \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (\lambda x.M)N : \tau} \text{ (App)}$$

Γ is the context of types, and the syntax $M : S$ corresponds to the the type scheme S of term M. The syntax $\Gamma \vdash M : S$ is a type judgement meaning type assignment $M : S$ is derivable from context Γ.

Turing showed as an appendix to his 1936 paper introducing Turing machines that the Turing machine and the λ-calculus were equivalent models of computation[24]. A key result of this equivalence was that a program that can be implemented on a Turing machine is λ-definable and vice versa.

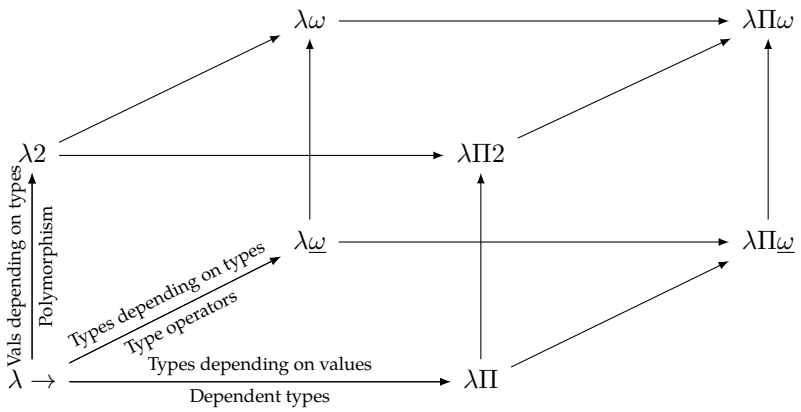

Figure 1.1: The λ-cube with labelled axis

Linking all these ideas of the Turing machine and the λ-calculus we now realise a programming language can be purely described via a λ-calculus, if we consider that a program is just a description of behaviour of some Turing machine. This is not a novel observation and was the reason for using type-theory(the area of research concerned with and reasoning about formal systems where terms have types and operations are defined through types e.g. $\lambda \rightarrow$) to describe the logic of programming languages (type systems) in the 20th and 21st century [1]. Figure 1.1 shows the λ-cube which shows some of the most influential type-theories that have been created as extensions to $\lambda \rightarrow$ and their relationships[2]. I will not go into detail between the difference of each vertex of figure 1.1 (axis are labelled with relationships) although some will be mentioned later when we reference proof assistants. The key takeaway is that as we evolve the λ-calculus we potentially evolve the expressiveness of a programming language.

1.1.3 Proofs as Programs: The Curry-Howard Equivalence

It can be seen that Church's approach to constructing the λ-calculus for reasoning about functions in number theory suggests that there might be a greater general relation between mathematics and computation. This was proven to be the case by Haskell Curry and William Alvin Howard over the course of several decades through observations that culminated in the discovery of the Curry-Howard correspondence formally made explicit in 1980 [15][2]. This states that every system of formal logic has an equivalent encoding in a computational model. For our context it can be proven that natural deduction is equivalent to $\lambda \rightarrow$. What we realise from the correspondence is that two systems of formalisms - one being based in formal logic the other being computation - are equivalent. The power of this result is important for our purposes as it justifies the entire field of ATP: we can construct programming languages which allow us to write formal proofs if we consider the logic of our language to be based in a formal system of computation.

1.2 Proof Assistants and EasyCrypt

The results described in the previous section were discovered in a world where computation was thought of as rote mathematical deduction by clerks. As the 20th century continued, the definition of computing in the real world evolved and described the tasks performed by machines described by programming languages. Type theory also changed with the times, and type systems were used for describing the underlying logic of programming languages.

Automath was one of the first projects started in 1967 that attempted to use the Curry-Howard correspondence with the $\lambda \rightarrow$-calculus for proof-checking [11]. One very key-insight from the creation of Automath was that we start to realise that the type of mathematics that we may want to reason about requires a translation of the mathematical theorems to our computational logic system e.g. one may want to formally prove geometric theorems but this requires a translation of the laws of geometry to a formal

[1] Although type theory has clear applications to computer science, it should be noted that type theory was initially introduced as an alternative basis to mathematics instead of set theory

[2] The initial manuscript was written in 1969, and initial observations made towards it's discovery were made in 1934

set of laws in our system[3]. This exists as one of the limitations of ATP which is designing programming languages that are expressive enough to allow for "easy" translation from mathematics to code. We will see later that an example of this limitation reveals itself in the EasyCrypt proof assistant motivating the results of this dissertation. Proof assistants exist as an applications of type theory for the sole purpose of writing mathematical proofs (via Curry-Howard) as supposed to just a programming language. The following is a brief history of proof assistants relevant to my work.

1.2.1 Coq

Coq[23] is arguably the most successful proof assistant that is still in widespread use today. Developed initially by Thierry Coquand and Gérard Huet in 1984, Coq consists of three main components: the logical language, the proof assistant and the program extractor. The logical language is a type theory called the calculus of inductive constructions [7] (CIC), who's relation to $\lambda \rightarrow$ can be seen in figure 1.1 represented as $\lambda\Pi\omega$ at the top of the λ-cube. The CIC can be used as both a typed programming language and a formal basis of mathematics, thereby motivating it's use in Coq as the method by which we write our axioms and specifications. The proof assistant allows for the development of verified proofs and the program extractor can generate programs that follow aforementioned specifications.

1.2.2 CertiCrypt

Cryptographers seek to construct schemes that allow for secure communication in the presence of some adversary. The language by which they use to write these schemes is mathematics, and they prove the formal notions of security that the schemes provide. However in the 21st century, the complexity of such proofs has grown significantly and has made the verification of such proofs by hand difficult. Cryptographers have argued for dedicated proof assistants that allow them to overcome these types of problems. CertiCrypt is one such proof assitant that has been developed on top of Coq by researchers at the IMDEA Software Institute and INRIA Sophia-Antipolis Méditerranée [5]; it supports common patterns of reasoning in cryptography, and has been used successfully to prove the security of many constructions, including encryption and digital signature schemes, zero-knowledge protocols, and hash functions.

1.2.3 EasyCrypt

EasyCrypt [16] is a proof assistant in the same family of languages as CertiCrypt. As a proof assistant it is unique in that it is not based on a type theory that uses dependent types like most automated theorem provers, and rather is implemented using a polymorphic λ-calculus based on the Damas-Hindley-Milner type system[14, 20, 10] which is a constrained version of System-F[12]($\lambda 2$ in the λ-cube 1.1). EasyCrypt seeks to provide a system where cryptographers can construct adversarial codes that allow for proving game-based cryptographic proofs. Game-based proofs treat notions of security as games played between an adversary and a challenger where we wish to prove that the probability of some result that the adversary wishes to achieve (say guessing encrypted information) doesn't exceed a certain threshold [18]. Figure 1.2 provides a visualisation of what a game may look like in the context of an adversary trying to guess an encryption key k when having access to an oracle that provides a one time encryption c of a chosen plain-text m made with TikZ[17].

For figure 1.2 we would want to prove our encryption scheme, $\mathcal{E}_k$, is 1CPA secure by showing that the probability of the adversaries guess for the encryption key is no better than just guessing a random key from the key-space $\mathcal{K}$ i.e. $Pr[\hat{k} = k] \leqslant \frac{1}{|\mathcal{K}|}$. This just exists as just one notion of security but it provides insight on the problem that EasyCrypt wishes to solve which is allowing for the construction of verifiable game-based proofs for encryption schemes. The initial prototype of EasyCrypt has been used to prove the security of several constructions, including the Cramer-Shoup encryption scheme [4], the Merkle-Damgaard iterative hash function design[1], and of the ZAEP encryption scheme[6]. The EasyCrypt organization has however, since 2012, been working towards a complete re-implementation of the assistant to overcome limitations.

[3] In 1959 Polish-American logician Alfred Tarski constructed an axiom set in first order logic representing a substantial portion of Euclidean geometry [22]

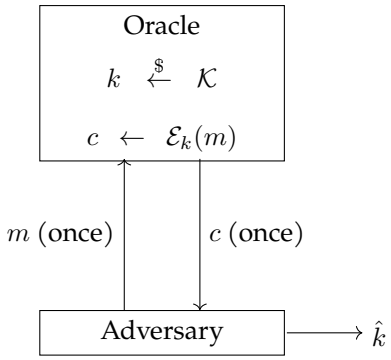

Figure 1.2: One time chosen plain-text attack (1CPA)

1.3 Programming Language Design Paradigms

I will now depart from formal theory and mathematics and discuss briefly the idea of programming language design. I will discuss two paradigms of PLD known colloquially as objected oriented and functional. The following seeks to explain and highlight the key differences between these paradigms. I will then briefly talk about type classes, an innovation in PL(D/T) which motivated the results of this thesis.

1.3.1 Functional

Functional programming is the paradigm that treats code as the evaluation of functions, and it's basis is found in the λ-calculus. When I described the λ-calculus, I introduced the idea of β-reduction which allowed us to reduce terms via the following rule $(\lambda x.M)N \rightarrow_\beta M[N/x]$. The relevancy to this paradigm is that we consider this to be an abstraction of function application, where the application of said function is defined through the rules of β-reduction on a function ,$\lambda x.M$, that takes in an input ,N. To gain more expressiveness one can extend the definition of terms, how they are constructed, and even introduce types such as through the $\lambda\rightarrow$. At the core we still have a formal definition of a functional programming language.

1.3.2 Object Oriented

Object oriented programming introduced the idea of objects as a way of reasoning about code. Objects (or classes) retain data in what are called fields, and functions for object behavior are defined within the object called methods. Through references to objects, one can access fields and methods. A programmer can build a model of some system through defining components and their behaviour through objects and defining how said objects interact. The difficulty of this model is defining a system through object decomposition and interaction, but one could argue the expressive power is immense. This idea became one of the most influential paradigms in modern programming, and the majority of modern programming languages usually support some form of object orientation. It's effect on how we think about programs shouldn't be understated.

1.3.3 Type Classes

Modern programming languages don't typically fit into just one paradigm. The majority of most languages borrow ideas from multiple paradigms. The reason for this is that PLD can be considered a form of user-experience problem. The users in this case want to be able to use a language to easily translate their ideas into code and they want to have certain guarantees on behaviour. The most expressive and feature rich language ends up being the most adaptable, but of course can also become unruly and difficult to understand and use if not designed well.

An example of a feature addition to a modern language is the creation of type classes designed to be added to any programming language based on DHM . Standard ML is one such functional programming language that was developed by R.Milner (of Damas-Hindley-Milner fame) starting in 1970 based on DHM. Type classes are a concept that were first introduced in 1989 by P.Wadler and S.Blott for the

Standard ML programming language (and DHM based languages) to resolve problems around object-oriented programming, bounded type quantification, and abstract data types [25]. This was a hugely influential idea that was introduced into other languages such as Haskell[13] and the Coq proof assistant [21] to name a few. In chapter 2 I will introduce formal definitions of type classes and explain practical use cases. What should be taken away is that type classes provide us greater expressiveness in our DHM based language when a functional implementation exists. This was the purpose of my dissertation: to add greater expressiveness to EasyCrypt via type classes.

1.4 Contributions

I have illustrated the context in which we are working in. We now know that the construction of proof assistants is not only feasible but helpful for improving our understanding of mathematics (and cryptography). EasyCrypt is the proof assistant for which all contributions of my thesis were made towards. The primary contribution of this thesis was an implementation of multi-parameter type-classes [25] in a hope to provide a more expressive language design. The primary development language for my work was Ocaml[19], as this is the implementation language of the EasyCrypt compiler. All source code for this thesis can be found at `https://github.com/EasyCrypt/easycrypt/tree/typeclass-draft/` as git commits under my username **nashpotato**.

Chapter 2

Technical Background

This chapter seeks to introduce and explain the main technical ideas necessary for understanding the contributions of this thesis. I will provide a formal description of the Damas-Hindley-Milner type system and the concept of type classes with reference to DHM. I will then introduce the EasyCrypt proof assistant.

2.1 The Damas-Hindley-Milner Type System

Modern programming languages typically have the ability to assign what is called a type to variables. The concept of a type allows us to essentially describe the expected behaviour/use of a variable. In a language such as Haskell, one may assign a variable a type which corresponds to a `Float` or an `Int` or whatever type available. The difference between these types relates primarily to what kind of value that variable can represent and how it can be used. For instance let's say an `Int` type would be a fixed-precision integer type with at least the range $[-2^{29}, \ldots, 2^{29-1}]$, and a `Float` type would correspond to a single-precision real floating point number i.e. represents values which can have up to 7 decimal points. This declaration of typing wouldn't necessarily end at variable declaration, but would also allow for the declaration of functions. In a typical function type declaration we would assign types to the arguments of the function and then assign a type to the return argument of said function. Listing 2.1 shows what this would look like in a Damas-Hindley-Milner type system based language, where our set of base types contains `Int` and `Float`.

```
{-- Variable declaration --}
integer : Int
float   : Float
{-- Function declaration --}
let leftid = λx.λy.x  : Int → Float → Int in
let rightid = λx.λy.y : Int → Float → Float in

let integer = 42 in
let float = 42.1234 in
leftId  integer float   {-- evaluates to 42 --}
rightid  integer float  {-- evaluates to 42.1234 --}

rightId float integer {--Failure, types dont match function signature --}
```

Listing 2.1: Example type declarations in a DHM based language

The rules that govern how types are defined, reasoned about and used are typically referred to as the type system of a language. As mentioned in the previous chapter, type systems typically are defined through a formal system such as the λ-calculus. The formal logic that is used in a language can then be seen to clearly dictate the behaviour and expressiveness of such a language. The Damas-Hindley-Milner type system is a type system that was created as a constrained version of the $\lambda 2$-calculus (or System-F), and is also the type system upon which the EasyCrypt proof assistant is based upon.

2.1.1 Features of the Damas-Hindley-Milner Type System

In chapter 1 I introduced the un-typed and simply typed λ-calculus and I also explained that through a calculi such as that a programming language designer can formally describe the logic of their own type system. I also explained that extensions had been made to the simply typed λ-calculus and briefly described the motivation for doing so to allow for greater expressiveness. Of course a vague term, greater expressiveness in essence describes what we want our language to allow us to do. If one was designing a language that allowed for something like object orientation, they would find that the simply typed λ-calculus wouldn't allow for polymorphism for example and they would extend the type system they are working in to allow for that behaviour. The motivation for the Damas-Hindley-Milner type system was to solve two problems that faced programming languages in the latter half of the 20th century which was polymorphism and type inference.

Polymorphism

The idea of polymorphism is to introduce abstraction into our typing. This is contrasted to monomorphism which is what is provided by $\lambda \rightarrow$. With monomorphsim we can provide typing judgements on variables as $x : T$, or we can provide a typing judgement on a function as $f : T \rightarrow T$. The problem with monomorphism is that the type T is taken to be atomic i.e. it can only be one type from our existing set of type schemes. This limitation can show itself when we consider how would we want to define the same operation on different data types

```
{- Operations on Floats-}
f1Float : Float → Float
f2Float : Float → Float
{- A composition function on floats -}
compFloat :(Float→ Float)→ (Float→ Float)→ Float→ Float
let compFloat = λf1.λf2 .λx . f2 (f1 x) in

{- Example Usage -}
compFloat f1Float f2Float 42.12345

{- Operations on Ints-}
f1Int : Int → Int
f2Int : Int → Int
{- A composition function on ints -}
compInt :(Int→ Int)→ (Int→ Int)→ Int→ Int
let compInt = λf1.λf2 .λx . f2 (f1 x) in

{- Example Usage -}
compInt flInt f2Int 42
```

Listing 2.2: Monomorphism in a DHM based language

As should be apparent from listing 2.2, what we have is an incredibly tedious way of constructing programs. With monomorphism the programmer is left with having to redefine operations for every type that they may wish to use, even if they behave in a similar way. As programs like this grow in complexity their flexibility and management decreases. We can see that there's a pattern with our code where we define methods with the the same structure repeatedly. Polymorphism provides us with flexibility as now our type T can exist as a generic placeholder for all existing types in our type scheme. Following from the example in listing 2.2 one may define a composition function on generic types.

```
{- A composition function on generic types -}
comp :∀ α.(α→ α)→ (α→ α)→ α→ α
let comp = λf1.λf2.λx. f2 (f1 x) in

comp f1Float f2Float {- : Float→ Float -}
comp f1Int f2Int {- : Int→ Int -}
comp f1Float f1Int {- Failure, not allowed as types dont match}
```

Listing 2.3: Example usage of polymorphism in a DHM based language

Although polymorphism means that operations can accept any type, in literature it is referred to as parametric polymorphism. The un-typed λ-calculus can be considered "polymorphic" in a non-strict definition as it's completely neutral to all types for example if I defined an identity function $id \equiv \lambda x.x$, all arguments are accepted by id. Parametric polymorphism essentially introduces an explicit type parameter and quantification for that type e.g $id \equiv \lambda x.x : \forall\alpha.\alpha \to \alpha$. A function is only polymorphic if it can have varying types in it's definition.

Type Inference

The previous listings have made explicit the types of defined functions and variables, although this isn't necessary if the type system allows for type-inference. Type inference is an algorithmic method by which a programming language may infer the types of expressions without them having to be explicitly set by a programmer. This is done by taking explicit information on the use of those operations and using that to infer types. Suppose we are given the following expression-type pair $E : T$. Type checking can be described as when given expression E and type T, check whether E really is of type T. When we are talking of type inference we have the case where we have expression E and we wish to determine the type T of said expression. DHM provides two type-inference algorithms, known as Algorithm J and Algorithm W.

Why not use the $\lambda 2$ Type System?

From figure 1.1 we can see that $\lambda\to$ is extended to allow for polymorphism with $\lambda 2$, and therefore would seemingly be a good candidate as the language of choice instead of DHM. This is also known as System-F and is actually DHM with just a bit more freedom. But this freedom comes with a cost as type inference in System-F is undecidable[26] as proven by J.B.Wells. This result is done by showing type-inference in $\lambda 2$'s equivalence to an undecidable problem known as semi-unification and is out of the scope of this dissertation. However this result was proven after the creation of DHM by 30 odd years, thereby implying that computer scientists may have intuitively known this property of System-F's and therefore decided to construct an alternative.

2.1.2 Formally Defining Damas-Hindley-Milner

The following is a formal definition of DHM. We seek to define the type system, and explain how this allows for polymorphism. To begin our definition we first define the language of DHM [10, 20, 14].

Definition 2.1.1: The Damas-Hindley-Milner Language

DHM consists of a syntax of expressions, types, and typing judgements.

(**Expressions**) given a set $\mathbb{I}$ of identifiers, the language of expressions $\mathbb{E}$ is given by the syntax where $x \in \mathbb{I}$ (parentheses may be used to avoid ambiguity)

$$e ::= x \mid ee' \mid \lambda x.e \mid \textbf{let } x = e \textbf{ in } e'$$

this being the same definition of expressions in the λ calculus except with the introduction of the **let** $x = e$ **in** e' expression.

(**Types and type schemes**) assuming a set of type variables and primitive types α, the syntax of types τ and of type-schemes σ is given by

$$\tau ::= \alpha \mid \tau \to \tau$$
$$\sigma ::= \tau \mid \forall\alpha.\sigma$$

A type scheme $\forall\alpha_1 \ldots \forall\alpha_n.\tau$, which can be written as $\forall\alpha_1 \ldots \alpha_n.\tau$, has generic type variables $\alpha_1 \ldots \alpha_n$. Our σ can be referred to as a polytype as it allows for multiple type variables bound by the $\forall$ quantifier, and τ is a monotype.

(**Context and typing judgments**) a context(or set of assumptions), written as Γ, is a list of pairs of variable type assignments, written $x_i : \sigma_i$. A typing judgment, is a triple of context and expression

type assignment. Both are written respectively as follows

$$\Gamma ::= \epsilon \mid \Gamma, x : \sigma$$
$$\Gamma \vdash e : \sigma$$

Definition 2.1.1 defines the syntax of DHM but doesn't provide inference rules. Before being able to do this I need to define what free type variables are as well as typing order.

Definition 2.1.2: Free Type Variables in DHM

For a polytype $\forall\alpha_1 \ldots \alpha_n.\tau$, all variables α_i are called quantified and the occurrence of quantified type variables in τ are called bound. Any types not bound are considered free. We define the $free$ function inductively on the syntax of DHM which constructs the set of free variables for expressions.

$$free(\alpha) = \{\alpha\}$$
$$free(\tau_1 \to \cdots \to \tau_n) = \bigcup_{i=1}^{n} free(\tau_i)$$
$$free(\forall\alpha_1 \ldots \alpha_n.\sigma) = free(\sigma) - \bigcup_{i=1}^{n} free(\alpha_i)$$
$$free(\Gamma) = \bigcup_{x:\sigma\in\Gamma} free(\sigma)$$
$$free(\Gamma \vdash e : \sigma) = free(\sigma) - free(\Gamma)$$

Definition 2.1.3: Specialisation Rule for DHM

The specialisation rule defines a partial ordering on quantified types in a polytype. The rule is defined as follows

$$\frac{\tau' = \{\alpha_i \mapsto \tau_i\}\tau \quad \beta_i \notin free(\forall\alpha_1 \ldots \alpha_n.\tau)}{\forall\alpha_1 \ldots \alpha_n.\tau \sqsubseteq \forall\beta_1 \ldots \beta_m.\tau'}$$

$\{\alpha_i \mapsto \tau_i\}\tau$ corresponds to applying substitution to a monotype τ i.e. replacing each occurrence of α_i in τ with τ_i. If we have the partial ordering $\sigma \sqsubseteq \sigma'$, we know that if we substitute some quantified variable in σ, we gain some σ'.

Now that we have definition 2.1.2 and 2.1.3, we can now define the inference rules of DHM. These rules can be thought of as semantic laws on program construction.

Definition 2.1.4: Inference Rules for DHM

The inference rules of DHM are defined inductively as follows.

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ (\text{Var}) \qquad \frac{\Gamma \vdash e_0 : \tau \to \tau' \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 e_1 : \tau'}\ (\text{App}) \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}\ (\text{Abs})$$

$$\frac{\Gamma \vdash e_0 : \sigma \qquad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \textbf{let}\ \ x = e_0\ \ \textbf{in}\ \ e_1 : \tau}\ (\text{Let}) \qquad \frac{\Gamma \vdash e : \sigma' \qquad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma}\ (\text{Inst})$$

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin free(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma}\ (\text{Gen})$$

The definition of inference rules encode rules around how polymorphism can be used in DHM. This is encoded with the *Inst* and *Gen* rules that allow the generalisation of types in definitions through type parameter bindings in the definition of polytypes(*Gen*) and substitution of type parameters in polytypes

with the specialisation rule (*Inst*). The remaining inference rules relate more to providing rules to the construction of expressions.

Type inference on DHM is defined by two algorithms which are referred to as "J" and "W". Formal definition of these algorithms will be admitted as they aren't necessary to understand the type system. It should be mentioned that Algorithm J and W perform with $\mathcal{O}(|e|)$ complexity i.e. linear time with respect to the size of the expression e. This is powerful as attempts at type inference algorithms tend to either have NP-hard complexity or are undecidable. Algorithm W differs from Algorithm J in that it is a form of the algorithm that deals with side-effects introduced with J's operation.

2.2 Type Classes

2.2.1 A Practical Example

We know now that any language that uses the DHM type system now has the expressive power to use type inference and polymorphism. As previously illustrated in listings 2.2,2.3 this type of expressive power is incredible. Questions are to be asked however on how would we do overloading, bounded type quantification and abstraction on types. To illustrate this, suppose that we wanted to define an operation over two types of numbers, `Int` and `Float`. We want to say that these two number types are in a $class$ of types that support said operation and no other type in our type system supports this operation. The listing below highlights this.

```
{-
    We wish to define a function eq which only takes in
    representation of numbers and performs an equality operation
    which returns a boolean
    (α ∈{Int, Float})
-}
eq :∀α. α → α → Bool
...
{- However, we can use eq on any α ∈ Γ -}
{- We can make eq explcit however to gurantee strictness,
   but must provide explicit operators
-}
eqFloat : Float → Float → Bool
eqInt   : Int → Int → Bool
...
{-
   But suppose we want strictness on double pairs, we have to define another
   function
-}
eqDouble : Double → Double → Bool
...
```

Listing 2.4: Ad-hoc polymorphism without type classes

Listing 2.4 shows the two main problems that we have with our polymorphic system. The first being is that we have no formal way of bounding the types that are allowed to be used in the definition of our polymorphic function, which may violate certain strictness guarantees we may want. Secondly as a consequence of wanting strictness the process of overloading is ad-hoc. If we want those strictness guarantees, we have to explicitly define functions that behave on our allowed types, which is inefficient and defeats the purpose of polymorphism. If we also want to extend our strictness on some other types, we have to add all appropriate operators to behave as expected which only grows in complexity as we add more types. This is problem known as bounded type quantification where we want to define a set $\{\alpha_1, \ldots, \alpha_n\} \subseteq free(\Gamma)$ which we can use as our set of types which we define polymorphic operations over.

The problems associated with ad-hoc polymorphism are solved with type classes [25]. Created in 1989 by S.Blott and P.Wadler for the Standard ML language, type classes are an extension that can be made to any DHM type system. Following from the previous listing we can illustrate how we would define our strict operation using type classes

```
{- Define a type class of equality -}
{- Eq τ is shorthand for the type τ→ τ→ Bool -}
over eq : ∀α.Eq α in
{- Define instances which support Eq -}
inst eq : Eq Int  = ··· in
inst eq : Eq Float = ··· in
inst eq : ∀αβ.(eq:Eq α).(eq:Eq β).Eq(α, β) =
    λp.λq.(eq (leftId p) (leftId q)) ∧ (eq (rightId p) (rightId q)) in
{- We can use eq as an overloaded operator -}
eq 1 2         {- Eq Int, evaluates to false -}
eq 2.0 2.0     {- Eq Float, evaluates to true -}
eq true false {- Eq Bool, failure as theres no Eq Bool instance -}
eq (4, 2.0) (7, 2.0) {- Eq (Int, Float), evaluates to false -}
eq (4, 'a') (4, 'a' ) {- failure as no Eq(Int, Char) instance -}
```

Listing 2.5: Less ad-hoc polymorphism using type classes

There are a few takeaways from listing 2.5:

- We construct an abstract class using *over* using polymorphic parameters that defines what behaviour(valid operations) types in the parameters have,
- when defining an *instance* for a class, we essentially add behaviour onto the existing behaviour of parameter types provided as arguments, but only for that type therefore providing bounded type quantification e.g. *eq* is only valid on the type sets $S = \{Int, Float\}$ and $S_{pair} = \{(\alpha, \beta) | \alpha, \beta \in S\}$ in the above listing,
- we have true overloading of functions, which means we retain polymorphism, through the passing of type classes into our operation definitions which adds context to the behaviour of polymorphic types. This behaviour is checked to be satisfied by the compiler as it looks for type class instances that are declared which have types as arguments matching our monotype when using a type class operation.

2.2.2 Formally Defining Type Classes for Damas-Hindley-Milner

We can provide a formal definition of type classes by extending our definition of DHM following Blott and Wadler's paper [25].

Definition 2.2.1: Type Class Syntax for DHM

The formal syntax of DHM is modified as follows to allow for type classes

(**Expressions**) We introduce new syntax terms $over$ and $inst$ that correspond to class and instance declarations.

$$\begin{aligned} e ::= & x \mid ee' \mid \lambda x.e \mid \textbf{let } x = e \textbf{ in } e' \\ & \mid \textbf{over } x : \sigma \textbf{ in } e \\ & \mid \textbf{inst } x : \sigma = e \textbf{ in } e' \end{aligned}$$

over $x : \sigma$ translates to declaring an overloaded identifier x. In the **inst** $x : \sigma' = e$ **in** e', the type σ' is an instance of the type σ which will be defined later. $over$ and $inst$ have caveats in that bound variables in their expressions cannot be re-declared in a smaller scope, and they must contain explicit types.

(**Predicated types**) to facilitate type classes, a third typing group is introduced on top of types ,τ, and type schemes ,σ, called predicated types, ρ, defined as below. We also redefine σ

$$\begin{aligned} \rho &::= (x : \tau).\rho \mid \tau \\ \sigma &::= \forall\alpha.\sigma \mid \rho \end{aligned}$$

we typically will refer to $x : \sigma$ as the predicate and $(x : \sigma)\rho$ as the predicated type

Definition 2.2.2: Translation

We define translation as the mapping of an expression e ,with class and instance declarations, to another expression $\bar{e}$ where class and instance declarations are removed therefore converting e to vanilla DHM. This is written as follows

$$e : \sigma \setminus \bar{e}$$

rules of translation will be defined in the inference rules. We also label three specific forms of binding for identifiers that we assign the following syntax.

$$x :_o \sigma \tag{2.1}$$
$$x :_i \sigma \setminus x_\sigma \tag{2.2}$$
$$x : \sigma \setminus \bar{x} \tag{2.3}$$

These correspond to binding of overloaded identifiers(2.1), declared instances of overloaded identifiers(2.2) and $lambda$ and let bound variables as well as assumed instances of overloaded identifiers(2.3).

Definition 2.2.3: Specialisation Rule 2

The partial order $\sqsubseteq$ is extended to apply to predicated types

$$\frac{\rho' = \{\alpha_i \mapsto \tau_i\}\rho \quad \beta_i \notin free(\forall\alpha_1 \ldots \alpha_n.\rho)}{\forall\alpha_1 \ldots \alpha_n.\rho \sqsubseteq \forall\beta_1 \ldots \beta_m.\rho'}$$

where we define $\rho \sqsubseteq \rho'$ iff the type part of ρ equals the type part of ρ' and for every predicate $x : \tau$ in ρ either

- there is a predicate of the form $x : \tau$ in ρ'; or
- the predicate can be eliminated under Γ

A predicate can be eliminated under Γ iff either

- $x : \tau \setminus \bar{x} \in \Gamma$; or
- $x :_i \sigma' \setminus \bar{x} \in \Gamma \wedge \sigma' \sqsubseteq \tau$

Definition 2.2.4: Unifiability

We say two type schemes σ and σ' are unifiable if there exists type τ and valid context Γ such that

$$\sigma \sqsubseteq \tau \wedge \sigma' \sqsubseteq \tau$$

we write $\sigma \# \sigma'$ if the two type schemes aren't unifiable.

Definition 2.2.5: Context

The set of assumptions Γ must also be redefined to take into account predicated types. We define it as follows.

$$\Gamma = \epsilon \tag{2.4}$$
$$\mid \Gamma, x : \sigma \setminus x \tag{2.5}$$
$$\mid \Gamma, x :_o \sigma, \quad x :_i \sigma_1 \setminus x_{\sigma_1}, \ldots x :_i \sigma_n \setminus x_{\sigma_n}, \quad x : \tau_1 \setminus x_{\tau_1}, \ldots, x : \tau_m \setminus x_{\tau_m} \tag{2.6}$$

where we have conditions on branch labelled 2.6 that state

- $\sigma \sqsubseteq \sigma_i$ for $i \in [1, \ldots, n]$, and
- $\sigma \sqsubseteq \tau_i$ for $i \in [1, \ldots, m]$, and

- $\sigma_i \# \sigma_j$ for $i \neq j$ and $i, j \in [1, \ldots n]$

Definition 2.2.6: Inference Rules for DHM with Type Classes

The inference rules of DHM are defined inductively as follows.

$$\frac{}{\Gamma, x : \sigma \setminus \bar{x} \vdash x : \sigma \setminus \bar{x}} \text{(Taut1)} \qquad \frac{}{\Gamma, x :_i \sigma \setminus \bar{x} \vdash x : \sigma \setminus \bar{x}} \text{(Taut2)} \qquad \frac{\Gamma \vdash e : \forall \alpha.\sigma \setminus \bar{e}}{\Gamma \vdash e : \{\alpha \mapsto \tau\}\sigma \setminus \bar{e}} \text{(Spec)}$$

$$\frac{\Gamma \vdash e : \sigma \setminus \bar{e} \qquad \alpha \notin free(\Gamma)}{\Gamma \vdash e : \forall \alpha.\sigma \setminus \bar{e}} \text{(Gen)} \qquad \frac{\Gamma \vdash e : \tau \to \tau' \setminus \bar{e} \qquad \Gamma \vdash e' : \tau \setminus \bar{e}'}{\Gamma \vdash e e' : \tau' \setminus \bar{e}\bar{e}'} \text{(Comb)}$$

$$\frac{\Gamma, x : \tau' \setminus x \vdash e : \tau \setminus \bar{e}}{\Gamma \vdash \lambda x.e : \tau' \to \tau \setminus \lambda x.\bar{e}} \text{(Abs)} \qquad \frac{\Gamma \vdash e : \sigma \setminus \bar{e} \qquad \Gamma, x : \sigma \setminus x \vdash e' : \tau \setminus \bar{e}'}{\Gamma \vdash \textbf{let } x = e \textbf{ in } e' : \tau \setminus \textbf{let } x = \bar{e} \textbf{ in } \bar{e}'} \text{(Let)}$$

$$x :_o \sigma \in \Gamma \; \frac{\Gamma, x : \tau \setminus x_\tau \vdash e : \rho \setminus \bar{e}}{\Gamma \vdash e : (x : \tau).\rho \setminus \lambda x_\tau.\bar{e}} \text{(Pred)} \qquad x :_o \sigma \in \Gamma \; \frac{\Gamma \vdash e : (x : \tau).\rho \setminus \bar{e} \qquad \Gamma \vdash x : \tau \setminus \bar{e}'}{\Gamma \vdash e : \rho \setminus \bar{e}\bar{e}'} \text{(Rel)}$$

$$\frac{\Gamma, x :_o \sigma \vdash e : \tau \setminus \bar{e}}{\Gamma \vdash (\textbf{over } x : \sigma \textbf{ in } e) : \tau \setminus \bar{e}} \text{(Over)}$$

$$x :_o \sigma \in \Gamma \; \frac{\Gamma, x :_i \sigma' \setminus x_{\sigma'} \vdash e' : \sigma' \setminus \bar{e}' \qquad \Gamma, x :_i \sigma' \setminus x_{\sigma'} \vdash e : \tau \setminus \bar{e}}{\Gamma \vdash (\textbf{inst } x : \sigma' = e' \textbf{ in } e) : \tau \setminus \textbf{let } x_{\sigma'} = \bar{e}' \textbf{ in } \bar{e}} \text{(Inst)}$$

Similar to our description of DHM, inference rules can be partitioned into two groups. The first six rules (*Taut1*, *Taut2*, *Spec*, *Gen*, *Comb*, *Abs*, *Let*) allow us to resolve overloading and explicitly define translation. The remaining four deal with our type class constructs and predicated types. The two rules *Pred* and *Rel* define how to introduce and eliminate predicates respectively. *Pred* translates the introduction of predicates by using a λ expression. *Rel* translates the elimination of predicates via application. The rules *Over* and *Inst* allow us to introduce *over* and *inst* expressions. *Over* adds correct overloading bindings ($:_o$) to expressions and *Inst* adds appropriate $:_i$ bindings.

We have now shown that we can formalise type classes for any type system that is based on Damas-Hindley-Milner. This provides a mathematical framework of how we can think about implementing said type classes in a language that uses DHM. In the next section I will introduce the EasyCrypt proof assistant. It will be explained that although EasyCrypt is based on DHM, it's use as a proof assistant and some of its unique features mean that how we think of type classes isn't exactly the same as the formal definition. This proves to not be a problem and just requires adjustment. Most modern languages that were inspired by type classes didn't strictly follow the formal definition but aimed to achieve the same functionality if not better with a little creativity.

2.2.3 Extensions on DHM

There are a range of extensions to DHM of which I have omitted formal definitions for such as recursion and type constructors. This was done purposefully to to focus on the ideas presented by Hindley, Milner, Blott and Wadler as historically speaking type classes were invented for the vanilla DHM system as we defined, and are the primary focus of this dissertation. I don't believe that the omission of these extensions take away from the formal understanding of my contributions although when looking at EasyCrypt, some of these extensions will be seen in its type system.

2.3 EasyCrypt

To introduce EasyCrypt requires a bit of exposition on why proof assistants differ from traditional programming languages. As explained in chapter 1, one can go about constructing formal proofs using programming language due to the Curry-Howard correspondence[15]. This idea was presented through the lens of the λ-calculus and it's relationship to first order logic. How this relates to programming languages wasn't made explicit, however as the previous sections of this chapter highlighted translation from ideas in formal type systems to concrete languages is intuitive once one considers PLD from an abstract level.

2.3.1 Ambient Higher Order Logic

EasyCrypt does provide a lot in terms of functionality as an automated theorem prover, but we can partition the language into two distinct parts. The first part we can consider is the ambient higher order logic. The second is the module system for constructing game based proofs. There's also other formal logics in EasyCrypt that allow for reasoning about probabilities which will be mentioned later.

The aHOL allows for the construction and proving of mathematical ideas, and is where the contributions of my work are immediately felt. When constructing proofs in a programming language we really want to define expressions like we would do with a regular language, and then be allowed to prove explicit properties of expressions. Relating this to the idea of the λ-calculus as a proof system, inference rules of such a system provide us with a way of deducing programs. As a consequence of the Curry-Howard correspondence, this process of deduction is equivalent to the logical deduction rules of some logic system such as first order logic i.e. a proof is a program, and the formula it proves is the type for the program. When I mentioned type inference and type checking in section 2.1, I introduced the pairing $E : T$ where E is some expression, and T is a typing assigned to E. With proof systems what we have is a way of reasoning about the situation in which we have some typing T, and we wish to see if there's an expression E that satisfies it. So a proof assistant gives us the ability to construct these expressions.

I will now informally introduce how one may construct a proof in EasyCrypt. The complete specification [16] of such a language is vast so what follows should be understood as an illustration of key ideas.

```
(* We declare some data type corresponding to unary
   representation of natural numbers
   e.g.3 == S (S (S Z)), 0 == Z*)
type unary = [
    | Z
    | S of unary
].
```

EasyCrypt provides support for the construction of custom types using type constructors as well as supporting recursion. These are extensions of the vanilla DHM system.

```
(* We then can define operations over such a data type such as addition *)
op add (x y : unary) : unary =
    with x = Z ⇒ y
    with x = S x_p ⇒ S (add x_p y).
(*i.e 1 + 2 == add (S Z) (S (S Z)) == S (add Z (S (S Z)))  == S (S (S Z))==3*)
```

Like most modern programming languages, the behaviour here is not particularly new. One may construct types, and define functions over types. Now we can digress into what sets apart proof assistants which is the ability to define and prove theorems.

```
(* For simplcity we define add as infix *)
op (+) = add.
(* We define our desired statement to prove as a lemma *)
(* We prove a left identity law *)
lemma leftId : forall (x: unary), Z + x = x. (* E : T *)
proof.
    move ⇒ x.   (*Move x into context, proof goal: Z + x = x*)
    reflexivity. (* Reduce sub-goal and eliminate if equivalence exists *)
qed.
```

We use the *lemma* syntax to declare our lemma which is equivalent to wanting to prove $\forall x \in \mathbb{N}, 0+x = x$. From there we introduce a proof using the proof syntax which allows us to try and find an expression E that satisfies our type $T \mapsto \forall x \in \mathbb{N}, 0 + x = x$.

```
proof.
    tactic_1.
    ...
    tactic_n.
qed.
```

The proof construct allows us to write proofs by modifying the proof goals of some expression. A proof goal can be considered a representation of the lemma state that we want to prove. This is a powerful construct as it allows for us to verify proof statements. Not only that, once a proof is provided for some lemma, that lemma itself can be used in the construction of other proofs thereby allowing us to inductively define more complex theorems. For the listing proving leftId, we use the *reflexivity* tactic. What this tactic does is reduce some proof goal by applying known definitions and determining equivalence. For example the application of reflexivity to the proof goal $Z + x = x$ applies the $(+)$ operation, giving proof goal $x = x$ and checks equivalence.

To explain in full the logic behind tactics requires extensive reading outside of the scope of the dissertation and complexity of tactics varies. What should really be understood is that tactics modify proof goals, which are essentially expressions, and apply some logical rule to get a new proof goal or even eliminating a proof goal. This is akin to something like composition of sub-programs to construct more complete programs, or even the use of axiomatic laws to construct theorems.

Once a programmer is able to understand the above mentioned constructs they are able to verify their programs. Although the formal reasoning behind these constructs is omitted, the relationship to type systems is actually not obscure. What we see with the above is that a proof assistant like EasyCrypt is identical to a programming language. It supports the normal types of constructs such as function declarations and so on. The difference is primarily that proof assistants provide extra in built logic to allow us to check properties of our languages. This could be embedded in a traditional programming language, but requires extensive formalisation and for most use cases, one may wish to rely on something quicker like unit testing to determine correctness. Of course this type of "correctness" isn't verified, it requires far less effort in terms of time spent and constructing proofs. However there are some fields of work where such verification through proof construction is worth the time spent for the guarantees provided such as cryptography.

2.3.2 Modules

The module system allows for the construction of game based proofs by allowing programmers to write games as interactions in the form of modules. An example construction of a module can be shown in the listing below

```
(* Define a module G1,that has method f which returns random boolean value.*)
module G1 = {
    proc f(): bool = {
        var x : bool;
        x <$ {0, 1}; (* Random sampling from set *)
        return x;
    }
}.

(* G2.f() yields the exclusive-or of two randomly chosen booleans*)
module G2 = {
    proc f() : bool = {
        var x, y : bool;
        x <$ {0,1}; y <$ {0,1};
        return x ^^ y;
    }
}.

(* PRHL judgement relating G1.f and G2.f. ={res} means res{1} =
res{2}, i.e., the result of G1.f is equal to (has same
probability distribution
as) result of G2.f *)
lemma G1_G2_f : equiv[G1.f ~ G2.f : true ==> ={res}].
(* proof omitted *)
```

Listing 2.6: EasyCrypt module example

As mentioned in chapter 1, with game based proofs we seek to prove probability bounds of some sort of interaction yielding a certain result. To do this in a programming language requires some system of

reasoning about probabilistic types, and interactions between probabilistic programs. EasyCrypt provides both of these through the probabilistic relational Hoare logic(pRHL) and the probabilistic Hoare logic(pHL)[3]. The pRHL allows for relating pairs probabilistic procedures and the pHL allows for proving properties of those features individually. The study of this part of the type system for probabilistic reasoning isn't necessary for understanding the outcomes of my work as my work primarily dealt with the ambient higher order logic. However it is worth mentioning to provide a whole picture of the EasyCrypt tool. Listing 2.6 illustrates the typical usage of a module system. We construct modules that have methods(processes) which can perform probabilistic computations. We can then formally prove equivalences on these computations. In the above listing proving the lemma *G1_G2_f* is equivalent to proving mathematically $Pr[F_{G1}] \equiv Pr[F_{G2}]$ where F_{G1} and F_{G2} are distinct discrete random variables determined by running *G1.f*() and *G2.f*() respectively.

With both the module system and the aHOL we can see that any cryptographer now has all the tools necessary to prove the security of schemes. This is done by the fact that a cryptographer can now formalise the mathematical primitives which define things such as encryption through the aHOL. Once those primitives are available, the cryptographer can settle for knowing that there scheme is logically correct, or can go on to prove game based security notions.

Chapter 3

Project Execution

In this chapter I will introduce the results of this work. I will first explain the approaches I took towards an implementation touching on the top level. I will then explain what I did in terms of implementation of type classes via changes made to the compiler. I will then seek to provide a formal definition of my implementation.

3.1 Practical Implementation

In this section I will seek to highlight the path I took towards implementing type classes. Initially I aimed to approach the problem by trying to stay on the top level of the language but as time went I became aware that any desired functionality would have to come through extending the compiler. The following is seeks to illustrate this initial approach to the problem and the resulting limitations that came with it. It then seeks to introduce the second approach which exists as a the final work submitted as part of this dissertation.

3.1.1 Initial Approach: Bundling

An initial step to implementing type classes in EasyCrypt is to consider whether we can embed the desired constructs in some library that is implemented within the language itself. This would definitely require less cognitive load as one wouldn't need to make considerations for the internals of the compiler. As this isn't a novel idea, approaches have been made to the construction of type classes on the top level in ATP. The Coq proof assistant has an implementation of type classes, commonly referred to as first-class type classes, which exist purely on the top level of the language [23, 21]. These exploit the dependent type theory that is part of the $\lambda\Pi\omega$ which defines Coq's logic. Although EasyCrypt does not support dependent type theory, I tried to work towards a top level case of type classes using what I refer to as bundling.

```
(* Define an Eq "type class" *)
type α Eq = {
    eq : α → α → bool;
}.
(* Declaring an instance *)
op intEq  : int Eq  = {|eq=···|}.
op charEq : char Eq = {|eq=···|}.
```

Listing 3.1: Bundling towards type classes

Listing 3.1 illustrates how a top level implementation of an Eq type class may look. This uses a data structure known as a record that allow us to define data types which can contain operators as fields. This allows us to "bundle" behaviour into a custom type. Any declaration of these types with custom parameters must be used by declaring an `op`, with parameters and fields defined. Naively one could think that this appears to be a one-to-one correspondence to how we defined type classes in chapter 2, however bundling actually comes with a lot of baggage when we consider usage. In fact three primary problems arise from the usage of bundling some of which limit the ease of use and some which clearly violate our desired behaviour.

```
(* Suppose we wanted to use our Eq type class to declare a lemma/axiom *)
axiom equivalent[α](eqClass: α Eq)(x y : α), x = y ⇒ eqClass.eq x y = true.
axiom reflexive[α](eqClass: α Eq)(x y: α), eqClass.eq x y = eqClass.eq y x.
axiom commutative[α](eqClass : α Eq) (x y z: α),
    eqClass.eq x (eqClass.eq y z) = eqClass.eq (eqClass.eq x y) z.
```

Listing 3.2: Bundling limitation 1: explicit parameters

Lising 3.2 highlights the first issue which is that type class instances are required to be passed explicitly when using them. Not only this, the programmer must make explicit reference to the appropriate type class instance in the input parameters when wanting to use a method in that class. This becomes increasingly difficult to construct more complex programs. Suppose that we wanted to use multiple type classes, a programmer would need to explicitly know which type classes use the desired methods. Our type class implementation for DHM didn't require this explicit nature as the compiler dealt with finding the appropriate type class instance when referencing a method via type inference. This meant that we extended the behaviour of types that were passed as parameters. As can be seen with bundling, declaring a type class instance doesn't extend the behaviour of a type hence why type class methods must be explicitly accessed. This is a violation that goes against our definition of the functionality of type classes.

```
(* We may want to bundle axioms into our type class, but we cant*)
type α Eq = {
    eq: α → α → Bool
    (* We may want to define explicit behaviour of eq *)
    (* law equivalence(x y : α), x=y ⇒ eq x y = true *)
    (* law reflexive(x y: α), eq x y = eq y x *)
    (* law commutative (x y z: α), eq x (eq y z) = eq (eq x y) z *)
}.

op intEq : int Eq = {|eq=···; (* PROVE LAWS *)|}
```

Listing 3.3: Bundling Limitation 2: no axioms

When type classes were introduced they were introduced as a specific extension of languages that didn't exist as verification tools but just a means to an end of programming a computer. EasyCrypt exists in the former and with that we want(and can) provide some way of verifying the behaviour of our data types. Type classes shouldn't exist as an exception to the rule in this reality. When we deal with proof assistants, we have the added ability to define explicit behaviour on types; it would be useful to also add explicit axiomatic laws on the behaviour of methods in a type class. Record data types don't provide this functionality. The idea with bundled axioms, like in listing 3.3, would be that the programmer when declaring an instance would also have to provide a proof of behaviour of the assigned operators by proving the axioms bundled in the type class. This is incredibly powerful as it allows us to declare a verified type class instance, giving us guarantees that can be passed around.

```
(* Suppose we want type classes to depend on other type classes *)
type α Comp = {
    eqClass: α Eq;
    neq    : α → α → bool;
    lt     : α → α → bool;
    (* law neqEq(x y: α), neq x y ≠ (eqClass.eq x y)  *)
    (* law ltEq(x y: α), lt x y  ⇒ !(eqClass.eq x y) *)
}.
...
(* More tiring limitations *)
axiom neqEq[α](compClass: α Comp) (x y: α),
    compClass.neq x y ≠ compClass.eqClass.eq x y.
```

Listing 3.4: Bundling limitation 3: relationships between type classes

From listing 3.4 we see the third limitation that exists with bundled type classes which is that we have no way of passing a type class as a parameter to another type class. The only alternative is to declare type classes as fields of the record. However the consequences of the two previous limitations can be seen

in usage. Because of explicit declaration of type classes as fields, methods that exist in those inherited type classes requires explicit access. If we had support for axioms in bundles those would also require explicit referencing.

The three limitations referenced led me to believe that any solution that required bundling would require extensions to the behaviour of records. However such changes would be extensive enough that we wouldn't realistically be dealing with the same data structure at all. This reveals that we can't rely on top level changes to allow for our type class implementation and must create our own data structures that behave as we want. The Coq proof assistant implements first type classes by the creation of dependent records. These are similar to the presented records but use the dependent type theory in Coq which allows for resolution of the limitations above. One could argue that working towards an implementation of a dependent type theory in EasyCrypt could be a solution. However such a project would require a rewrite of the whole type system. A work of that kind would be in it's own scope and would go beyond our desired goals. I decided to invest my energy into second-class type classes. These are type classes that are implemented by making changes to the functionality of the compiler and extending our type system similar to what was presented in chapter 2 when we extended DHM.

3.1.2 A New Approach

The Compiler

To allow for the formal reasoning of mathematics in a language such as EasyCrypt, there must exist a compiler. The compiler can be split into two primary parts, as most compilers, which is the front-end and the back-end. The front-end primarily exists to take in source programs and translates them to some sort of intermediate representation that can be reasoned about. This involves parsing of source code and creating abstract syntax trees. At this end of the compiler, syntactic errors are also found. The back-end primarily deals with several things mainly reasoning about programs in intermediate representation form, generating native code for hardware and applying optimisations. For the case of EasyCrypt, the back-end is primarily a reasoning engine that performs our verification. Figure 3.1 shows the relationship between the stages of a typical compiler.

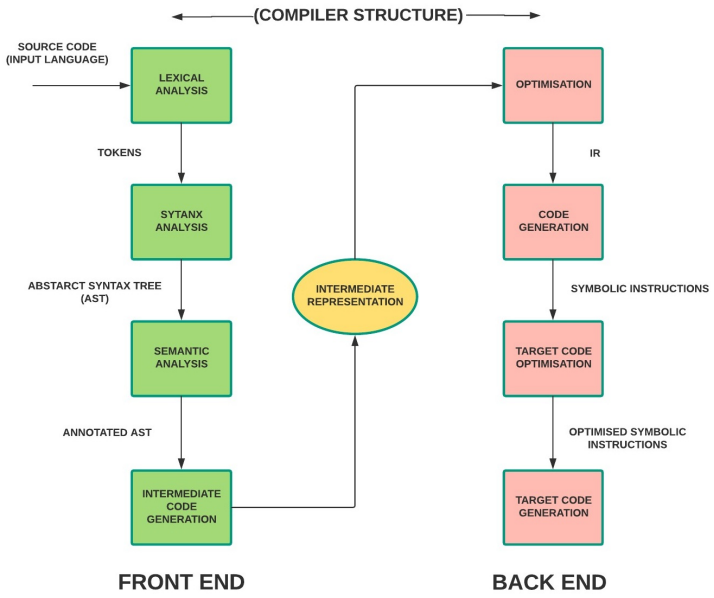

Figure 3.1: Standard compiler stages

In chapter 2 I introduced the concept of translation(definition 2.2.2) when we considered type classes. One can think of translation as turning declarations of type classes and instances to an expression that could be constructed in the absence of type classes, and placing them into the appropriate context of the program where they're used. This allows for type classes to just exist as an extension to the type system that requires a "translation" to the vanilla type system, and leaves any implementation to just

be for a pre-processor that performs that translation operation. This pre-processing can be done by just extending the front-end of the compiler. This was the approach that I made to implement type classes.

Syntax

The first step towards a type class implementation is to consider syntax.

```
type class α Eq = {
    (* Define operators with signatures only *)
    op eq: α → α → bool;
    (* Define expected behaviour via axioms *)
    axiom equivalent(x y : α), x = y ⇒ eq x y = true;
    axiom reflexive(x y: α), eq x y = eq y x;
    axiom commutative (x y z: α), eq x (eq y z) = eq (eq x y) z;
}.

instance intEq with int Eq = {
    (* Define operators of type class *)
    op eq = ···;
    (* Provide proofs that axioms hold *)
    lemma equivalent Eq by ··· ;
    lemma reflexive Eq by ··· ;
    lemma commutative Eq by ···;
}.
```

Listing 3.5: Type class syntax in EasyCrypt

Listing 3.5 shows the syntax that I opted for type class and instance declarations. Regarding type class declaration, I opted for a similar syntax to record types, although there's caveats and we will see usage is different. Similar to record types, type classes contain fields. Fields must be provided with an explicit signature, but cannot be declared. There's also no support for sub-classes, only operators and axioms. A real difference to records is the allowance for axioms to be declared in a type class. The purpose of this is it allows for a clear definition of the behaviour of any named fields; when declaring an instance all relevant axioms require explicit proofs. This brings us to instance declaration. Instances are bound to names, and to declare an instance one must provide concrete parameters which correspond to the polymorphic arguments of the type class declaration. From there all fields from the type class must be defined. I didn't wish to explicitly check the type signatures of provided expressions as that would be resolved with the lemma declarations i.e. incorrect type signatures will not pass type checking of axioms. After binding the operators, of which all must be defined, the programmer is then left to provide explicit proofs of the axioms using the syntax `lemma [axiom_name] [type_class_name] by [tactics];` where all referenced fields in axioms are overloaded with the provided bindings. One can also add their own axioms and lemmas to the instance definition in addition to the compulsory ones.

Usage

Once a syntax is established, I considered what the typical usage of type class instances should be. Although similar to how type classes work in a DHM system, we have to also consider how we can use lemmas provided in instances.

```
lemma int_equiv forall (x y: int), x = y ⇒ eq x y = true.
proof.
    move ⇒ x y.
    apply intEq_equivalent. (* Explicitly reference instance for lemmas *)
qed.

op (==): int → int → bool = eq.
op (==): α → α → bool = eq. (* Failure, no α Eq instance *)

instance charEq with char Eq = {
    (* Define operators of type class *)
    op eq = ···;
```

```
    (* Provide proofs that axioms hold *)
    lemma equivalent Eq by ··· ;
    lemma reflexive Eq by ··· ;
    lemma commutative Eq by ···;
}.

(* correct Eq operations implicitly determined *)
lemma char_equiv forall (x y: char), x = y ⇒ eq x y = true.
proof.
    move ⇒ x y.
    apply charEq_equivalent.
    (* apply intEq_equivalent.  Failure, parameters dont match int ≠ char *)
qed.

(* Failure, no Bool Eq instance *)
lemma bool_eq forall (x y: bool), x = y ⇒ eq x y = true.
...
```

Listing 3.6: Type class usage in EasyCrypt

Listing 3.6 highlights the key features of the usage of my type class implementation in EasyCrypt. Operators that are defined in valid instances are available globally for use in other declarations such as lemmas and operators. Any declaration that makes use of an operator that hasn't got a valid instance fails. The correct operator is resolved when referenced by using the built in type inference algorithm of EasyCrypt. Also operators from type classes cannot be used polymorphically i.e. one cannot reference for example `eq:` $\alpha \to \alpha \to$ `bool`.

Lemmas from instances don't exist in the same way as operators as any reference to a type class lemma must be accessed using a `[instance_name]_[lemma_name]` syntax. The reason for this is that to allow for implicit usage of lemmas and axioms would require the ability to determine the equivalence of two axioms of same form. This can become an increasingly intractable problem with expression complexity as it would require operator resolution and type resolution, and then determining expression equivalence. An approach for similar lemmas/axioms to exist would be to allow for the type class to be passed as an extra parameter to its definition i.e. similar to predicates in predicate types. The provided syntax can be considered similar in that vein as one must make explicit reference to instances.

Multi-parameter Type Classes

When I presented type classes as they were defined by Wadler and Blott they were introduced as single parameter type classes. There didn't exist anyway for type classes to exist with multiple parameters in their definition e.g. `over rel :` $\forall\alpha\beta.\alpha \to \beta$. This is a limitation as we can see that any type class which provides for multiple parameters allows us to define type classes which encapsulate relationships between types. From a mathematical perspective this is a useful expressiveness and as we intend for mathematicians to use EasyCrypt, it would be useful to have this ability. Because of this I have allowed for type classes to allow for multi-parameter definitions and usage

```
(* Define a type class which allows for maps between types *)
type class (α, β) Mappable = {
    op map: α → β
    axiom defined forall (x : α), exists (y: β), map x = y;
}.

instance intToChar with (int, char) Mappable = {
    op map = ···;
    lemma defined Mappable by ··· ;
}.
```

Listing 3.7: Multi-parameter type classes in EasyCrypt

Listing 3.7 highlights how a 2-parameter type class may be declared. It should be noted that when declaring instances, one must bind all parameters, they cannot keep one parameter free in the declaration e.g. `instance _ with int Mappable`.

Type Classes on top of Type Classes

I mentioned previously that we potentially would like to pass type classes to other type classes allowing for some form of inheritance. As of writing this dissertation, I haven't been able to provide for that behaviour. For the use of type classes in other type classes, a programmer would have to follow the procedure shown in listing 3.8, similar to how we would pass records to other records with the bundling approach but slightly less verbose.

```
type class α Ord = {
    op lt: α → α → bool;
    axiom neq forall (x y: α), lt x y ⇒ !(eq x y);
}.

instance intOrd with int Ord = {
    op lt = ···;
    lemma neq Ord by ··· ;
}.

(*
   Failure ⇒ Cannot be proven/instantiated as no bool Eq instance for access
   to eq func as a consequence theres no access to Eq lemmas
*)
instance boolOrd with bool Ord = {
    op lt = ···;
    lemma neq Ord by ··· ;
}.
```

Listing 3.8: Pseudo-inheritance with type classes

This appears to behave as we expected when considering inheritance. This usage exploits the global nature of type class operators and lemmas in instances. The real difference from true inheritance is that this syntax doesn't allow for defining an explicit relationship between type classes[1].

3.2 Formal Definitions

Now that I have introduced in practical terms what the consequences of my work allows us to now do in EasyCrypt, I will now present formal definitions of the logic behind my work. The choice of presenting the formal definitions instead of lines of code, is motivated by the fact that formal definitions allow us to express the relevant ideas in a clear concise way. To show all relevant code would require a deep dive into the EasyCrypt compiler as well as my code, and in the literature for PLD, work is usually presented via formalisation. There's also the issue that presenting disparate code snippets doesn't necessarily illuminate what's happening without the full code context. I have left that responsibility to the reader who I hope that after reading my formal definitions will be able to relate it to the EasyCrypt source code available here https://github.com/EasyCrypt/easycrypt/tree/typeclass-draft as git commits under my username **nashpotato**. For the sake of brevity, we have omitted the definition of the whole of EasyCrypt type system except when relevant to our reasoning and definitions.

3.2.1 Definitions

The following seeks to build an a formal defintion of type classes in EasyCrypt by myself. This takes a similar approach to building out definitions as was shown in 2 where we start with basic definitions that build on complexity. I will seek to link ideas mentioned when necessary and provide extra exposition to remove ambiguities as I see fit.

[1] All limitations mentioned will be discussed in chapter 4

Definition 3.2.1: Type Expressions in EasyCrypt

The set of valid type expressions is defined as follows

$$\tau ::= \epsilon \mid \alpha \mid t \mid \alpha\, t \mid (\alpha_1, ..., \alpha_n) \mid (\alpha_1, ..., \alpha_n)t \mid \tau \to \tau$$

Polymorphism, although not made explicit, is possible due to the fact set α encompasses all valid type variable identifiers(polytypes) and t represents all valid concrete type identifiers(monotypes). For the bracket syntax $(\alpha_1, \ldots, \alpha_n)$ we have the condition $\forall i, j \in [1, \ldots, n] \wedge i \neq j, \alpha_i \neq \alpha_j$. ϵ represents a placeholder type which is used in a case where type need not be declared. We use the DHM syntax for type judgements i.e. $\Gamma \vdash x : \tau$.

Definition 3.2.2: Free for Type Expressions in EasyCrypt

We define a function free which provides the free type variables in an expression τ as,

$$\begin{aligned}
free(\alpha) &= \{\alpha\} \\
free(t) &= \{t\} \\
free(\epsilon) &= \emptyset \\
free(\tau_1 \to \cdots \to \tau_n) &= \bigcup_{i=1}^{n} free(\tau_i) \\
free(\tau\, t) &= free(\tau) + \{t\} \\
free(\ (\alpha_1, ..., \alpha_n)) &= \bigcup_{i=1}^{n} \{\alpha_n\} \\
free(\ (\alpha_1, ..., \alpha_n)t) &= \{t\} + free(\ (\alpha_1, ..., \alpha_n)) \\
free(\Gamma) &= \bigcup_{x:\tau \in \Gamma} free(\tau) \\
free(\Gamma \vdash e : \tau) &= free(\tau) - free(\Gamma) \\
free(S_1, \ldots, S_n) &= \bigcup_{i=1}^{n} free(S_i)
\end{aligned}$$

Definition 3.2.3: Strict Substitution

We define a strict substitution, denoted $\{x_i \mapsto x_i'\}$, between two types σ and σ' if they are tuple types of form $\sigma = (\sigma_1, \ldots, \sigma_n)$ and $\sigma' = (\sigma_1, \ldots, \sigma_m)$ and the following holds

$$\sigma' = \{\sigma_i \mapsto \sigma_i'\}\sigma \wedge n \equiv m$$

we denote shorthand notation for the existence of such substitution as

$$\sigma \mapsto \sigma'$$

Definitions 3.2.1-3.2.3 define a basis from which we can reason about our type class syntax with rules for reasoning about free type variables, substitution and type signatures. The first definition exists to define the type expressions that exist in EasyCrypt as of writing this dissertation. One should notice that these are quite similar to type expressions provided in the definitions of DHM presented in the previous chapter except for the addition of what is known as a tuple type i.e. $(\alpha_1, \ldots, \alpha_n)$. The tuple type represents a multi-type which means it exists as a type that is made of n-distinct types. The second definition is our free function that serves the same purpose of a free function as defined in 2.1.2 which is determine free type variables in a type expression. What is a unique introduction is my definition of strict substitution. The use of this isn't made explicit yet but strict substitution exists to allow us to reason behind the creation of multi-parameter type class instances. We can now define typing judgements in our syntax.

Definition 3.2.4: Typing Judgements for Type Classes in EasyCrypt

We define judgements on type class and instance identifiers and also define syntax for the relationship between operators and axioms in type classes and instances.

(**Type Class Judgements**) A type class consists of type identifier, written x^{tc} and typing judgement σ, written as the following

$$x^{tc} : \sigma$$

where

$$\sigma ::= \alpha \mid (\alpha_1, ..., \alpha_p)$$

and for $(\alpha_1, \ldots, \alpha_p)$ we have the condition $\forall i, j \in [1, \ldots, p] \wedge i \neq j, \alpha_i \neq \alpha_j$.

(**Type Class Operators**) we define notation which relates an operator declared in a type class $x^{tc} : \sigma$ as the following with typing judgement Σ

$$x_i^{tc \to op} : \Sigma$$

where

- for n operators in type class declaration expression e^{tc}, $i \in [1, \ldots, n]$,
- free$(\Sigma) \subseteq free(\Gamma, \sigma)$

(**Type Class Axioms**) we define notation which relates an axiom declared in a type class $x^{tc} : \sigma$ as the following,

$$x_i^{tc \to ax} : \epsilon$$

for m axioms in type class declaration expression e^{tc}, i$\in [1, \ldots, m]$,

(**Instance Judgements**) an instance of a type class consists of instance identifier x^{in} and typing judgement γ, written as the following

$$x^{in} : \gamma$$

where

$$\gamma ::= \ t \mid (t_1, ..., t_p)$$

(**Instance Operators**) we define notation on instance$(x^{in} : \gamma)$ operators which relates to a type class $x^{tc} : \sigma$ as the following with tying judgement Σ',

$$x_i^{in \to op} : \Sigma'$$

where

- for n operators in type class declaration expression e^{tc}, $i \in [1, \ldots, n]$,
- $|free(\sigma)| = |free(\gamma)|$, and there's a strict substitution $\sigma \mapsto \gamma$, $\{\alpha_i \mapsto t_i\}$,
- the type variables available for Σ' are strictly monotypes obtained from $\Sigma' = \{\alpha_i \mapsto t_i\}\Sigma$.

(**Instance Axioms**) we define a notation on instance$(x^{in} : \gamma)$ axioms which relates to a type class $x^{tc} : \sigma$ as the following,

$$x_i^{in \to ax} : \epsilon$$

for m axioms in type class declaration expression $e^{tc}, i \in [1, \ldots, m]$.

There's a lot to digest with the rules of typing judgements above but it can be split into two distinct definitions which is the type judgements on types classes and the type judgements on instances. Type classes in this definition are treated as a unique identifier in their own right that can exist in Γ. Firstly type classes have a type assignment which corresponds to the polymorphic parameters that they take as arguments when defined and are given a unique identifier. This means that when we reason about a type class we're not reasoning about a type but rather an "object" with axioms and instances. With reference to axioms and operators these are tied to the type class via the identifier. Instances are essentially the

same except that they must exist with type that corresponds to a strict substitution of the type of the type class. There's also the caveat that all referenced type variables in used in operator definitions of an instance must also follow strict substitution as well. Axioms in instances are logically the same as axioms in type classes and just share an identifier. Unlike previous definitions I made reference to typing judgements before declaring our syntax. The primary reasoning is that with our combined approach to type class and instance declarations, it helps to introduce useful notation prior to expressing syntax.

Definition 3.2.5: Type Class Syntax in EasyCrypt

Our type class syntax consists of type class declarations and instance declarations.

(**Type Class Expression**) We define the syntax of a type class expression, e^{tc}, as the following with valid identifier set $\mathbb{X} \ni x, a_i$,

$$\begin{aligned} e^{tc} ::= \textbf{type class}\ \sigma\ x^{tc} = \{ \\ x_1^{tc \to op} : \Sigma; \quad \ldots \quad ; x_n^{tc \to op} : \Sigma; \\ x_1^{tc \to ax} : \epsilon; \quad \ldots \quad ; x_m^{tc \to ax} : \epsilon; \\ \} \end{aligned}$$

where

- x^{tc} is an identifier for our type class,
- $x_i^{tc \to op}$ is a valid operator expression, of form, $e^{op} ::= \textbf{op}\ x_i : \Sigma$, with identifier x_i,
- $x_j^{tc \to ax}$ is a valid axiom expression, of form $e^{ax} ::= \textbf{axiom}\ a_j\ e$, with identifier a_j,
- $\forall i, j \in [1, ..., n] \land i \neq j, x_i \neq x_j$,
- $\forall i, j \in [1, ..., m] \land i \neq j, a_i \neq a_j$.

(**Type Class Instance Expression**) We Define the syntax of a type class instance, e^{in}, as the following

$$\begin{aligned} e^{in} ::= \textbf{instance}\ x^{in}\ \textbf{with}\ \gamma\ x^{tc} = \{ \\ x_1^{in \to op}; \quad \ldots \quad ; x_n^{in \to op}; \\ x_1^{in \to ax}; \quad \ldots \quad ; x_m^{in \to ax}; \\ \} \end{aligned}$$

where

- x^{in}, x^{tc} correspond to instance and type class identifiers respectively,
- $x_i^{in \to op}$ is a valid operator expression of form, $e^{op'} ::= \textbf{op}\ x_i = e$, with identifier x_i,
- $x_j^{in \to ax}$ is a valid lemma expression with proof, of form $e^{le} ::= \textbf{lemma}\ a_j\ x^{tc}\ \textbf{by}\ [\textbf{tactics}]$, with identifier a_j and sequence of proof tactics $[\textbf{tactics}]$.

As can be seen from the definition of our type class expressions, we make reference to our typing judgement syntax. What should be take away from the definition of type class syntax is that we have clear relationship between instances and type classes that relationship being that an instance cannot exist with reference to a type class in scope the provides its blueprint.

Now that we have the expressions that determine typing judgements and our new syntax we can add logic to behaviour. We can define the formal reasoning via inference rules. It should be mentioned that the following inference rules aren't wholly prescriptive of the whole type system behaviour. One can see that from the inference rules, we can simply define the appropriate logic that corresponds to type classes without modifying the entire type system i.e. the consequences of this work can exist as an add on to existing behaviour and not a rewrite.

Definition 3.2.6: Inference Rules for Type Classes in EasyCrypt 1

The following are the inference rules for the EasyCrypt type system in relation to type classes for declaration.

$$\frac{\begin{array}{c}\Gamma \vdash x^{tc} : \sigma \\ \Gamma \vdash e_1^{op} : \Sigma_1, \ldots, e_n^{op} : \Sigma_n \\ \Gamma,\ e_1^{op} : \Sigma_1, \ldots, e_n^{op} : \Sigma_n \vdash e_1^{ax} : \epsilon, \ldots, e_m^{ax} : \epsilon\end{array}}{\Gamma \vdash (\textbf{type class}\ \sigma\ x^{tc} = \{e_1^{op} : \Sigma_1; \ldots; e_n^{op} : \Sigma_n; e_1^{ax} : \epsilon; \ldots; e_m^{ax} : \epsilon\}) : \sigma}\ \substack{free(\Sigma_i) \\ \subseteq free(\Gamma,\sigma)}\ \text{(TypeClass)}$$

where e^{op} and e^{ax} correspond to an operator and axiom expressions meeting our syntax definition shown in 3.2.5.

$$e^{tc} : \sigma \in \Gamma, \sigma \mapsto \gamma\ \frac{\begin{array}{c}\Gamma \vdash x^{in} : \gamma \\ \Gamma, x_i^{tc \to op} : \Sigma_i \vdash e_i^{op'} : \{\alpha_k \mapsto t_k\}\Sigma_i \\ \Gamma, e_i^{op'} : \{\alpha_k \mapsto t_k\}\Sigma_i,\ \ x_j^{tc \to ax} : \epsilon \vdash e_j^{le} : \epsilon\end{array}}{\Gamma \vdash (\textbf{instance}\ x^{in}\ \textbf{with}\ \gamma\ x^{tc} = \{e_i^{op'} : \Sigma_i'; e_j^{le} : \epsilon\}) : \gamma}\ \text{(Instance)}$$

where e_j^{le} corresponds to a valid proof of lemma, following our syntax in 3.2.5, for axiom $x_j^{tc \to ax}$. We obtain $\{\alpha_k \mapsto t_k\}$ as the strict substitution satisfying the side condition $\sigma \mapsto \gamma$.

As mentioned the primary reason that we can constrain the logic of type classes to modifications to the front-end of a compiler is due to the nature of a translation operation. If we can provide translation rules from our extended typ system we can just do the appropriate front-end work to assist in functionality. The challenge here then becomes learning the compiler. I provide a definition of a form of "translation" through the following inference rule.

Definition 3.2.7: Inference Rules for Type Classes in EasyCrypt 2

The following defines inference rules relating to the usage of type classes and instances in EasyCrypt.

$$\frac{e^{in} : \gamma \in \Gamma}{\Gamma \vdash x_i^{in \to op} : \Sigma'}\ \text{(Op)}\qquad \frac{e^{in} : \gamma \in \Gamma}{\Gamma,\ x_i^{in \to ax} : \epsilon \vdash x^{in}\_\, a_i : \epsilon}\ \text{(Axiom)}$$

The two laws, *Op* and *Axiom*, define how ops and axioms respectively can be used in context. From this we can see that my approach to type classes side-steps the need for explicit translation operations by allowing for lemmas and axioms to be embedded in the type system in the same form as the vanilla EasyCrypt type system. This is done by allowing for the creation of type classes and instances from previous inference laws and treating that as a datatype which is brought into scope of the system. From that point if an appropriate instance exists then one can access it's axioms and operations. As a consequence of this my approach leverages the existing compiler. If one was to look at my source code they could see that my works exists as extensions to the compiler which leverages existing functionality while adding new functionality.

The above definitions present the reasoning behind my type classes. The absence of a full description of the type system may be seen as a detriment, but it just requires creativity how such instructions would fit into the regular type system context. To find how all this would fit I had to learn how the EasyCrypt compiler works and then make desired changes. Below is a short semi-descriptive list which links the ideas presented to the compiler source files. This list exists to allow for the reader to have an easier time relating the presented ideas to the compiler itself as it exists just under 150 source files. Although not necessarily a common approach to presenting these PLD research I chose to do this as it allows for a more focused appraisal of contributions. One can look at the source code itself on Github which provides a commit history by date of my contributions. All of this is available at the link that has been previously referenced: https://github.com/EasyCrypt/easycrypt/tree/typeclass-draft

- To allow for the expression definitions I made changes to the parsing stage of the compiler and abstract syntax tree definitions: **ecParser.mly, ecParsetree.ml(i), ecDecl.ml(i)**,
- to allow for strict substitution and checking for valid proofs provided for instance declaration I made changes to the following: **ecTheory.ml(i), ecScope.ml(i)**,
- to check that programs obeyed the: *TypeClass* and *Instance* laws: **ecScope.ml(i), ecSubst.ml(i), ecLocation.ml(i), ecDecl.ml(i)**,
- to allow for the *Op* and *Axiom* laws: **ecEnv.ml(i), ecScope.ml(i), ecTheory.ml(i)**.

Chapter 4

Critical Evaluation

4.1 The Gap: Mathematics to Code

All work on ATP is hinged on the consequence of the Curry-Howard equivalence. The idea being that if we have a formal proof system there's an analogous formal language system that allows us to write code which can prove programs. However the equivalence does nothing in stating the relative ease that such translation requires. Although I haven't presented long proofs of any formal theories in EasyCrypt, from relevant listings one can see a big issue that exists with ATP - writing languages that allow us to develop these proofs is not a trivial task. Any work done on ATP has to take into consideration that to write programs isn't the same as writing formal proofs in practice, and for developers of such languages we must allow for as much expressiveness as possible when considering our language design.

EasyCrypt has the greater task of having to exist as a proof assistant for a subset of mathematics which is cryptography. This subset, although theoretical in definition, concerns itself with very concrete and practical applications. What this means is that any theoretician also has to have a mind for practical applications. With the consequences of this work, I hoped to provide even more expressiveness to allow for programmers/mathematicians to have more power when working on cryptography. This is done in small part by type classes. In chapter 2 I outlined what type classes provide us. They resolve problems around bounded type quantification for polymorphism. The relation to mathematics for our purposes may not be obvious but one has to think how you would construct a mathematical object in EasyCrypt. Mathematical objects can be considered as constructs that define properties of types(sets, integers e.t.c.). This should be seen as the same definition of type classes. These properties are the laws and methods that the type class defines. Valid instances of such objects have clearly defined operations and proofs of laws. An instance thereby shows that our mathematical object can exist for some type or types are provided as arguments. These types of relationships between an abstract objects behaviour on types of some kind is common in mathematics and cryptography e.g. groups, functions, categories. Therefore with type classes EasyCrypt only increases it's ability in a useful direction.

4.2 Alternatives

Although the consequences of this work were made to the EasyCrypt proof assistant, there's little stopping the ideas presented here to be carried forward to other proof assistants that wish to solve the same problem of machine verified cryptography. In this case one could argue that the formalisation's presented could easily be re-contextualised in another proof assistant. The only limits to this generalisation is that proof assistants don't all exist with the same underlying proof system. Because EasyCrypt is based on a variation on DHM, the consequences of this work can only really be fully utilised in a DHM-like system. Any other formal proof system would have it's own challenges with the design of type classes and a more specific application would have to be considered. There would even have to be arguments about whether, for a proof system with polymorphism, type classes of some variety are the best solution. This is again completely dependent on context.

4.3 Related Work: Coq

Work has been done on type classes in proof assistants. This work although contextual to the language highlights some the strengths and weaknesses of any such approach to type classes in any proof assistant. We must consider that type classes were initially created for use in general purpose functional programming languages; any work which shifts that focus to a niche application such as theorem proving requires a new perspective on what we wish to achieve with our implementation.

Mentioned briefly throughout this dissertation is the Coq proof assistant. This exists as a general theory proof assistant which exists to allow for formalisation of mathematics in a range of domains. I mentioned briefly in chapter 3 that this proof assistant allows for the use of type classes through a construct I referred to as dependent records [23, 21]. I also mentioned that these type classes are considered "first-class" which means that the construction of them is made possible through just using the Coq proof assistant itself rather than modifying the underlying Coq logic. Coq's expressiveness as a consequence of the $\lambda\Pi\omega$-calculus allow for this extension in the first-class. We know that the EasyCrypt proof assistant is based on a DHM calculus and so we were limited to only allowing for changes to be made as second-class type classes. Despite this it is worth referring to Coq as it does exist in the same space of proof assistants and therefore can be seen as a source of inspiration(and potentially warning) on the design of type classes in EasyCrypt. There are some similarities and difference that are worth referencing between my type class implementation and Coq's.

- Coq seeks to allow for implicit arguments in the usage of instance methods, as does my implementation. This works by allowing the compiler to perform a search of the appropriate instance method using type inference,
- Coq allows for optional naming of type class instances. This differs from my approach in which naming of instances must be made explicit (and referencing lemmas must be done through names). Sozeau and Oury, authors of the paper outlining these type classes, believe that creating fresh names can be difficult and error prone and they use as much information provided by the programmer to disambiguate [21],
- definitions (analogous to our lemmas) require type classes to be passed as parameters differing from my approach where type classes aren't part of a lemma or axioms arguments.

4.4 Limitations and Further Work

Within the scope of this thesis, any extensions to my work should take the route of resolving any existing limitations of my implementation. There are questions to be asked about whether that should extend to changing the existing functionality of the type system to allow for new logic such as dependent types. The space of programming language design is vast and to consider all existing research that could prove to be relevant for ATP in EasyCrypt is a task fit for a PhD. Below is a list of what I believe are the greatest limitations of my work and what I believe would be the next steps of improvement.

Implicits

Implicit are a way of thinking about type inference on abstract types, or what can be seen in analogy as type classes. The basic idea behind implicits is that when writing programs that make reference to some type class defined method, the programmer is not left with having to explicitly reference those methods via the type class instance, but rather the compiler resolves the correct method "implicitly" using type inference. As can be seen from my implementation of type classes we have implicit behaviour with type class methods. They can be referenced without making explicit call to any related type class instance. We don't have however implicit behaviour when we consider the use of type class lemmas and axioms. The primary reason for this is that lemmas and axioms don't exist in scope with a type that we can perform inference on i.e. $x^{in \to ax} : \epsilon$. This means that when we wish to perform a resolution on the use of lemma or axiom by some type of inference algorithm, we're performing that operation on all expressions in every instance with a matching identifier. Determining the difference between these lemmas and axioms becomes an intractable problem; as expressions defining lemmas and axioms can make reference to any methods in scope in their definition, the number of valid expressions once we perform type inference on referenced methods grows exponentially with expression complexity.

Naming

As per the formal definitions presented in the previous chapter all type class instances require explicit names when declared. This can be seen as a limitation primarily as it creates a cognitive load on the side of the programmer to know explicitly which instance they refer to when they refer to a lemma. This can be seen as a limitation for small programs but one could argue that in the context of a large program requiring explicit naming to instances helps eliminate ambiguity on usage. The real limitation is that to provide names for multiple instances can become tedious.

Related to instance naming is the naming of axioms and lemmas. EasyCrypt provides no type of lemma overloading so all referenced lemmas must have unique identifiers in scope. The consequences of my work are that through the instance referencing, one can effectively use the same lemma in multiple contexts without ambiguity provided by reference to the instance declaring said lemma.

Lemma Parameters

From the example usage provided in the previous chapter we can see that type classes aren't passed as explicit arguments to lemmas or operations. An example usage of what this may look like if supported is below

```
lemma int_equiv{int Eq} forall (x y: int), x = y ⇒ eq x y = true.
proof.
    move ⇒ x y.
    apply equivalent.
qed.
```

We see from the above that an instance is passed in as a type of parameter. The idea behind a syntax like this is that we add extra explicit context to the lemma, in particular with the above syntax we would be saying that int types in this context are also of the Eq type class. This means we require less type inference to refer to appropriate methods, and axioms/lemmas don't need an explicit reference to an instance for use. This can be seen naively as a possible solution to the implicits problem for lemmas and axioms as by adding the extra context we can make explicit reference a lemma in that instance. The problem is that this behaviour isn't implicit at all but rather explicit as all relevant instances are made clear for types. There's also an argument to be made that there's no real logical difference between referencing lemmas via my underscore syntax (`[instance_name]_[ax_name]`) and explicitly passing a type class parameter.

If this was implemented with support for multiple type class parameters, the next step would be to have some way of unification on type classes provided. This is because that although we require unique identifiers on type class axioms, we don't require unique identifiers on type class methods. By bringing multiple methods into scope a type inference algorithm would have to be implemented to determine the correct methods on usage especially when methods share signatures, identifiers but have different behaviours[1]. Any implementation that supports this would provide clearer usage of type classes in proofs thereby easing readability.

Inheritance

Inheritance allows for the composition of type classes.

```
type class α Ord extends α Eq = {
    ...
}.

instance intOrd with int Ord = {
    ...
}.
```

The listing above shows a typical use case. When declaring an instance there would also have to be an extra search to prove that appropriate inherited instances exist and then to unify the results of the two instances i.e. for int Ord, an int Eq instance should exist. There's a lot of hidden complexity that comes with this type of behaviour. The primary thing is that a dedicated unification algorithm would need to be implemented to allow for type resolution and also to determine safe dependencies and cyclic

[1] Whether this problem could be solved is an open question

dependencies. With this behaviour however we allow for programmers to have more explicit control over relationships between different type classes.

Chapter 5

Conclusion

In this dissertation I presented the work I did towards implementing type classes in the EasyCrypt proof assistant. I explained the necessary theory behind proof assistants, type systems and the relationship between programming languages and mathematics.

I presented the consequences of my work by explaining use cases and providing formal definitions. In particular I highlighted the relationship between polymorphism and the expressiveness it provides, but also the limitations that exist without type classes.

I have provided a critique of my implementation when compared to existing work in other proof assistants, as well as limitations of my work. I have explained how these limitations could be potentially rectified and what we gain from doing so.

All source code is available in the above mentioned repositories.

Bibliography

[1] Michael Backes, Gilles Barthe, Matthias Berg, Benjamin Grégoire, César Kunz, Malte Skoruppa, and Santiago Zanella Béguelin. Verified security of merkle-damgård. In *2012 IEEE 25th Computer Security Foundations Symposium*, pages 354–368. IEEE, 2012.

[2] Henk P Barendregt. Lambda calculi with types. 1992.

[3] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Probabilistic relational hoare logics for computer-aided security proofs. In *International Conference on Mathematics of Program Construction*, pages 1–6. Springer, 2012.

[4] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Annual Cryptology Conference*, pages 71–90. Springer, 2011.

[5] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 90–101, 2009.

[6] Gilles Barthe, David Pointcheval, and Santiago Zanella Béguelin. Verified security of redundancy-free encryption from rabin and rsa. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 724–735, 2012.

[7] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[8] Alonzo Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936.

[9] Alonzo Church. A formulation of the simple theory of types. *The journal of symbolic logic*, 5(2):56–68, 1940.

[10] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, 1982.

[11] Nicolaas Govert De Bruijn. A survey of the project automath. In *Studies in Logic and the Foundations of Mathematics*, volume 133, pages 141–161. Elsevier, 1994.

[12] Jean-Yves Girard. The system f of variable types, fifteen years later. *Theoretical computer science*, 45:159–192, 1986.

[13] Cordelia V Hall, Kevin Hammond, Simon L Peyton Jones, and Philip L Wadler. Type classes in haskell. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(2):109–138, 1996.

[14] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[15] William A Howard. The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490, 1980.

[16] Inria, IMDEA Software Institute, and École Polytechnique. EasyCrypt: Computer-Aided Cryptographic Proofs. https://www.easycrypt.info/trac/.

[17] Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2016.

[18] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. CRC press, 2014.

[19] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system release 4.09*. https://caml.inria.fr/pub/docs/manual-ocaml-4.09/.

[20] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[21] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2008.

[22] Alfred Tarski. What is elementary geometry? In *Studies in Logic and the Foundations of Mathematics*, volume 27, pages 16–29. Elsevier, 1959.

[23] The Coq Development Team. The coq proof assistant, version 8.10.0, October 2019.

[24] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[25] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 60–76, New York, NY, USA, 1989. Association for Computing Machinery.

[26] Joe B Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1999.