Utrecht University
Faculty of Science
Department of Information and Computing Sciences
Graduate School of Natural Sciences

Verified Compiler Optimisations

Computing Science MSc Thesis

Joris Dral

Supervisors:
Wouter Swierstra
Jacco Krijnen
Gabriele Keller

Acknowledgements

I would first like to thank my supervisors for their guidance throughout the project. Topical discussions and brainstorming sessions during our weekly meetings and handful of presentations offered plenty of handholds for me to progress. The amiable atmosphere and your approachability for questions and advice contributed to the fact that I thoroughly enjoyed working on the project.

In addition, I would like to thank my family and friends for their trust and support. I particularly admire your willingness to listen to my (not always successful) attempts at explaining the topic of the thesis in an accessible-to-all manner. All jokes aside, I am very grateful for all of you.

Abstract

This thesis explores how to formally verify the correctness of a certifying compiler where the correctness of individual compiler runs is specified by translation relations, each of which characterises the admissible behaviour of a single compiler pass. Whereas the correctness of compiler passes follows from the fact that we can recognise compiler behaviour, there is no guarantee that the recognised behaviour is correct. This motivates the need for formal guarantees that express that translation relations themselves are well-behaved.

We use the Coq Proof Assistant to demonstrate how we can prove that a well-behaved translation relation preserves the static and dynamic semantics of programs. In a preliminary study, we show positive results in the context of a simply typed lambda calculus: we define static and dynamic semantics preservation, and we prove both properties for several example translation relations. The main takeaway of this study is that we can use logical relations to express and prove dynamic semantics preservation.

The preliminary study serves as a starting point for scaling up our methods to a more complex lambda calculus that is based on System F_{ω}^{μ}. Though the proof method for dynamic semantics preservation does not scale up easily, we research and demonstrate how *step-indexed* logical relations solve the issues that arise such that we can define and prove dynamic semantics preservation. Furthermore, we prove static and dynamic semantics preservation for a realistic translation relation.

The semantics preservation proof method is developed for use in a certification engine for the Plutus Tx compiler: a smart contract language compiler intended for commercial use that is developed and maintained by IOHK. As such, we show a promising proof method that can be applied in realistic compilers and verification tools.

Contents

Chapter 1

Introduction

The formal verification of compilers is a research area that is the subject of extensive research[12] dating back as far as 1967[27]. This is not surprising, as compilers are pieces of highly advanced software that perform non-trivial tasks. Amongst others, these tasks include code optimisations and translations from and to a multitude of (intermediate) representations and/or programming languages. These tasks are performed in sequence as separate compiler passes, and all passes together characterise the compilation of a source code to some target code.

Crucially, each of the compiler passes must be *correct* in order to state with confidence that the target code is well-behaved with respect to the source code. The exact definition of correctness can differ throughout literature, in part because of the different types of programming languages on which the literature reports. However, the definition generally encompasses the notion that each of the compiler passes preserves a set of pre-determined properties. Formal verification of compilers is concerned with formally proving such correctness properties using rigorous mathematical methods.

In the context of safety-critical software, it can be argued that it is even more vital to formally verify compiler correctness. This becomes clear when we consider a type of safety-critical software that is called *smart contracts*. They are small, compiled programs on a public blockchain that represent legal agreements which often concern themselves with sizable financial assets. Thorough explorations of the Ethereum smart contract system show that the system had a number of security issues that are exploitable in practice[3, 22]. Moreover, there exist examples of successful attacks on smart contract systems that resulted in great financial losses, such as the notorious DAO attack[23] and the Parity Multi-Sig Wallet attacks [40, 6, 39].

Importantly, security vulnerabilities are not exclusively the result of smart contract programming mistakes, but can also be caused by bugs in the compiler. Such bugs can be found even in commercial and open-source compilers[11], including smart contract language compilers such as Vyper[28]. In addition, a security audit of the historical Serpent smart contract language compiler unearthed a range of bugs in the compiler code, amongst which bugs of critical severity[32, 4]. Careful auditing of compiler code is, however, not the whole answer. It could not prevent the second Parity Multi-Sig Wallet attack[39]. It becomes clear that compiler verification is part of the solution.

Compiler verification can also serve an additional purpose: providing a *verifiable link* between source and target code, which allows a user to trust the compiled code. However, there exist a number of compiler verification flavours, and it is not immediately clear which flavour suits the need of a smart contract system best. We distinguish these flavours in two ways and compare them.

A first distinction that we can make is that of full compiler verification versus single-run verification. While whole-compiler proofs offer the strongest guarantees, namely that any compiler run is correct, development of the compiler is more complex. Particularly, a single change in the compiler design may require an update to the proof. Instead, it may be a more manageable alternative to only verify individual runs of the compiler such that verification is not as tightly coupled with compiler design. One alternative is *Translation Validation*[31], where each run of the compiler is followed by a validation phase that automatically checks whether the semantics of a program are preserved by comparing code before and after compilation. Research shows that translation validation can be applied to realistic compilers [25, 33]. An arguably similar alternative is based on *Proof-Carrying Code*[24], where a *certifying compiler* pairs each piece of compiled code with machine-checkable evidence (a certificate) that the compiled code has some desired property [26]. Colby et al. show that it is possible to develop a certifying compiler for realistic languages such as a subset of Java[9]. An advantage over translation validation is that the certificate can double as a verifiable link.

A second distinction is that of on-paper proofs versus mechanised proofs. The use of mechanised proof assistants has led to a number of projects that formally verify properties of programming languages and compilers using machine-checked proofs. Examples of projects that are built in the Coq Proof Assistant[5], which is a popular choice of proof assistant for this type of project[10], include the CompCert project[19, 20, 21], the Lambda Tamer project[8], and the Aura programming language[14]. Projects that are built in other proof assistants exist as well, such as a verified implementation of ML written in HOL4[18], and a verified compiler for a subset of Java written in Isabelle[36]. It can be argued that these projects show that the use of proof assistants has had a positive impact on the process of formally verifying compiler correctness. One argument in favour is that on-paper proofs of compiler correctness can be unconvincing because such proofs are extensive and far from trivial to construct[13]. Conversely, mechanically checked proofs can be more convincing and better maintainable than the alternative on-paper proof, though arguments can be made against the use of proof assistants as well[13].

With the previous considerations about smart contract language compiler verification in mind, this thesis zooms in on the Plutus Tx compiler. The compiler is part of the Plutus Platform[1]: a smart contract platform for the Cardano blockchain[2] that is developed and maintained by IOHK[3]. The Plutus Tx compiler concerns itself with compiling smart contracts, which are written in a subset of Haskell, to Plutus Core. The latter is a version of System F that is further extended with higher-order kinds and arbitrary recursive types[7]. A hash of the compiled code is put on the blockchain, and is then the on-chain representative of the smart contract.

Because on-chain code is nearly impossible to change if a compiler bug is found, there should be a high level of assurance that the compilation of this code is *correct* – i.e., the compiled code behaves according to the specification of the source language. For this reason, development was started on a certification engine for the Plutus Tx compiler using the Coq Proof Assistant[17]. The certification engine is written in the style of a certifying compiler[26]: each piece of compiled code is paired with a machine-generated compilation certificate that describes that the target program is well-behaved with respect to the source program. The certificate gives strong guarantees about the correctness of this single instance of the compiler's execution, and it allows users to trust on-chain code. Note that an argument against whole-compiler verification of the Plutus Tx compiler is that the work would amount to verifying parts of the Glasgow Haskell Compiler, which is a daunting task.

The development of the certification engine has, up until now, focused mostly on recognising the behaviour of the Plutus Tx compiler[17]. Each transformation that is executed by the compiler is characterised by a *translation relation*[4]. In summary, a translation relation specifies the admissible behaviour of a compiler pass. However, these translation relations offer no guarantee that the compiler pass is well-behaved. As such, it is an open problem to show that the semantics of terms are preserved across these translations. In fact, semantics preservation can be seen as a vital property of any *correct* compilation pass. This leads to the main research question of this thesis:

Research question: *Can we formally verify that translation relations are semantics preserving?*

Overview To answer the research question, we distinguish three main goals that we also implement in Coq: (1) constructing a formal static and dynamic semantics for the (intermediate) languages, (2) defining static and dynamic semantics preservation and (3) providing machine-checked proofs that show the semantics preservation property for example transformations. Though it is not within the scope of the project to cover all (intermediate) languages and transformations of the Plutus Tx compiler, we show that the results of this thesis project offer a solid basis for verifying the correctness of each individual translation relation. We give an overview of the contents of this thesis below. Throughout the paper, we define everything on paper and then supplement the definitions with Coq proof script snippets that highlight how we implement the definitions in Coq.

- We discuss the scientific background upon which this thesis builds. This background includes a discussion of the Plutus Tx compiler, previous work on the certification engine and the Software Foundations books. (Chapter 2)
- We discuss a preliminary study that was conducted during the proposal phase of this thesis project. We show that we can prove semantics preservation for a few example transformations in the context of a small functional language: a simply typed lambda calculus. Importantly, we showed that we can prove dynamic semantics preservation using logical relations. As the (intermediate) languages of the Plutus Tx compiler, such as Plutus Core, are all based on a simply typed lambda calculus, we believed that this positive result implied that we could scale up our methods to the Plutus Tx compiler. (Chapter 3)
- We formally define the PIR and Plutus Core languages by defining their concrete and abstract syntax, a static semantics that represents a type checking algorithm and a dynamic semantics in the form of a big-step operational semantics. (Chapter 4)
- We make precise what semantics preservation means. Specifically, we define both static and dynamic semantics preservation. In our discussion of dynamic semantics preservation, we adapt Ahmed's work on step-indexed logical relations[1] to our setting and we extend their work such that we can use the logical relation to prove dynamic semantics preservation. (Chapter 5)
- We show that we can prove both static and dynamic semantics preservation for an example translation relation: desugaring of non-recursive `let`-bindings. (Chapter 6)

[1]`https://github.com/input-output-hk/plutus/`
[2]`https://cardano.org/`
[3]`https://iohk.io/`
[4]In this thesis, we often use the concepts of transformations and translations interchangeably.

Chapter 2

Scientific background

The thesis project as a whole builds upon two types of scientific background. On one hand, this scientific background consists of previous work that concerns itself with the Plutus Tx compiler. On the other hand, we consider literature and existing technology that are related to the Coq Proof Assistant and compiler verification. We list important literature below, after which we discuss a few works in more detail.

The Plutus Tx compiler. In section 2.1, we give an overview of the Plutus Tx compiler, and we focus on its (intermediate) languages in particular [7, 15]. Section 2.2 describes how Krijnen et al. have identified the different transformations that are executed in the compiler, and how we can characterise these transformations and specifiy their correctness using translation relations[17].

The Coq Proof Assistant and compiler verification. Section 2.3 discusses the first two books of Software Foundations in Coq[29, 30]. Most of the results of the preliminary study in chapter 3 directly draw from the second book, and they serve as a source of inspiration for the remainder of the thesis as well. We also use the `Equations` package[34, 35], which allows us to write recursive functions through dependent pattern-matching much like we would write recursive functions in Haskell. In addition, the `Equations` package simplifies the process of proving well-foundedness of more complex recursive functions in Coq. However, we do not further discuss the package here. Lastly, the results pertaining to dynamic semantics preservation build upon Ahmed's work on step-indexed logical relations[1]. We defer a detailed explanation of the work and our adaptation/extension of it to chapter 5.

A note on citations. In the upcoming chapters, we will not always explicitly reference the literature that we discuss in the next few sections. Instead, we remark that most of this thesis' results depend on and draw from the literature mentioned – i.e., these works are ubiquitous throughout the thesis. As such, we hope that this chapter gives sufficient credit to the authors. Futhermore, we will give concluding remarks in chapter 7 that summarise what we contribute to these previous works.

2.1 The Plutus Tx compiler

The Plutus Tx compiler[1] compiles a program that is written in a subset of Haskell to Plutus Core[7] in two parts: first, by compiling Haskell in several steps to GHC Core, and secondly, compiling GHC Core in several steps to Plutus Core using the Plutus Intermediate Representation (PIR)[15] as an intermediate language. The compilation process consists of a range of compiler passes, such as variable renaming, type checking, inlining of `let`-bound variables, desugaring of `let`-bindings, dead code elimination and datatype encodings. Much of the ongoing development of the certification engine has concentrated on the part that compiles PIR to Plutus Core, so we discuss those languages in more detail.

Plutus Core is designed with simplicity in mind because of a blockchain's inherent property that whatever is put on the blockchain is hard to revise. Therefore, the developers opted for a small core language. They designed it to be a modest extension of System F_ω^μ, which consists of System F with higher-order kinds and iso-recursive types[7]. The inclusion of iso-recursive types allows for the use of the Scott encoding to encode datatypes. Because of its relevance for the Plutus compiler, IOHK funded a research project that resulted in a complete formalisation of System F_ω^μ that was written in Agda using an intrinsically typed syntax[7]. Amongst other contributions, the authors give mechanically checked proofs for a number of properties, such as the properties that Progress and Preservation hold for a small-step operational semantics. These results reinforce the belief that the language specification of Plutus Core is correct in the sense that its semantics are well-behaved.

[1] `https://github.com/input-output-hk/plutus/`

In the process of compiling GHC Core to Plutus Core, the compiler uses the the Plutus Intermediate Representation (PIR) as an intermediate representation. PIR is almost synonymous with FIR, which adds recursive datatypes and recursive functions onto System F_ω^μ through the addition of both (mutually) recursive and non-recursive `let`-bindings[15]. PIR adds the `let`-bindings onto Plutus Core in the same fashion that FIR adds them onto System F_ω^μ. Explicit `let`-bindings have the advantage of making it easier to reason about transformations such as inlining and dead code elimination. Additionally, the use of PIR breaks up the translation from GHC Core to Plutus Core into more managable, less complex pieces. Amongst other contributions, the authors provide a complete specification and implementation of a compiler that compiles FIR to System F_ω^μ[15]. However, the authors do not give a direct operational semantics for the intermediate language.

The implementation of Plutus Core and PIR differs in a few ways from System F_ω^μ and FIR as they are described in the papers. For example, type checking and type inference do not proceed precisely according to the typing rules as defined in the papers. In addition, the implemented languages include additional constructs such as constants and built-in functions. Naturally, we consider the compiler implementation as leading in this thesis. In chapter 4, we re-formalise the static and dynamic semantics of the implemented languages for the sake of precision.

2.2 Translation relations and correctness definitions

Recent research efforts have led to the ongoing development of a certification engine for the Plutus Tx compiler, written using the Coq Proof Assistant in the style of a certifying compiler[26]. Amongst other accomplishments, Krijnen et al. have identified the different transformations that are performed by the Plutus Tx compiler in the process of compiling PIR to Plutus Core[17]. To formally characterise the translations, they introduce the concept of *translation relations*. As the correctness of translation relations is the core subject of this thesis, we repeat the exposition of translation relations here.

Let the AST of code in a compiler's source language be denoted by t_1, and let the AST of the code in a compiler's target language be denoted by t_n. There could be an arbitrary number of intermediate ASTs t_i where $i \in [1, \ldots, n]$ that are possibly inhabitants of intermediate languages. Each translation from an AST t_i to an AST t_{i+1} is characterised by a *translation relation* R_i, which acts as a witness for the compilation of t_i to t_{i+1}. By the transitivity property of these R_i, the composition of all translation relations characterises the compilation of t_1 into t_n through its intermediate ASTs:

$$t_1 \text{ compiles to } t_n \text{ if } R_1(t_1, t_2) \wedge R_2(t_2, t_3) \wedge \cdots \wedge R_{n-1}(t_{n-1}, t_n) \tag{2.1}$$

In essence, each R_i should describe the admissible behaviour of a compiler pass f_i – that is, the relation R_i decribes the different ways in which a term t_i can translate to a term t_{i+1} according to f_i. As such, the graph of admissible transformations of f_i is in R_i, alternatively formulated as $\text{graph}(f_i) \subseteq R_i$.

As an advantage to this approach, each of the translation relations can ignore any concerns about implementation details of the compiler passes. In practice, each f_i could use some heuristic to determine how and where to apply transformations. Decoupling implementation details from each R_i simplifies the verification process. If the heuristic of f_i changes, then the definition of R_i can probably remain the same. Moreover, the definition of any R_i is independent of any of the other translation relations. In summary: the certification engine is robust with respect to changes in the compiler, and its development can also be performed in an incremental fashion.

We define a compiler pass f_i to be correct if we can characterise its behaviour using a translation relation R_i:

Definition 2.1 (Compiler pass correctness). $f_i(t_i) = t_{i+1}$ *implies* $R_i(t_i, t_{i+1})$

Naturally, the characterisation of the behaviour of a compiler pass does not necessarily imply true correctness of the compiler pass if R_i itself is not correct. In fact, since each f_i should be *sound* with respect to the corresponding R_i, whatever we prove about R_i will also hold for f_i. So, to substantiate the correctness of each R_i and by extension the correctness of each f_i, we should prove a measure of correctness properties of R_i with regards to the semantics of the underlying (intermediate) languages. Let $[\![t_i]\!]_i$ denote the semantics of a term t_i. Let $\sim_i$ denote a binary operator that defines a relation between two semantics that captures that two terms are the same. Examples of such a relation include static semantics preservation and dynamic semantics preservation. We define a correctness property of a translation relation as follows:

Definition 2.2 (Translation relation correctness). $R_i(t_i, t_{i+1})$ *implies* $[\![t_i]\!]_i \sim_i [\![t_{i+1}]\!]_{i+1}$

Proving semantics preservation for translation relations is the main aim of this thesis. This means that we aim to prove a number of theorems that are formulated in the style of Definition 2.2. We will make more precise the concepts of static semantics preservation and dynamic semantics preservation in chapter 5. When we report on the results of the preliminary study in chapter 3, we will also discuss a simpler, though similar version of both concepts within the context of the preliminary study.

2.3 Software Foundations

Software Foundations[2] is a series of six[3] books that discusses the field of software technology, and specially the development of reliable software. Through use of the Coq Proof Assistant to write and mechanically check the proofs of theories that are posed, the authors show how proof assistants can make a significant contribution to writing provably correct software.

The first two books[29, 30] are the main source of Coq-related material that will be used in this thesis project. Both books were used as an introduction to Coq, though the second book also serves as a major source of inspiration for this thesis.

2.3.1 Logical Foundations

The first book in the series, called Logical Foundations[29], aims to introduce three main topics and explain how they interact: (1) functional programming using the functional programming language that is built inside Coq, called Gallina, (2) basic theories of logic, and (3) the use of mechanised proof assistants. As the reader progresses through the book, the writers also show how Coq's type system facilitates the use of Coq as a proof assistant.

2.3.2 Programming Language Foundations

Logical Foundations offers a first insight into reasoning about programs through a number of chapters about a simple imperative programming language. The second book, called Programming Language Foundations[30], deals with the same topic in-depth. Amongst others, the book covers reasoning about programs using Hoare Logic, operational semantics, static semantics and advanced proof techniques. Most importantly, the majority of the chapters consider those topics in the context of a simply typed lambda calculus. This an advantage since the Plutus Tx compiler's (intermediate) languages are all based on a simply typed lambda calculus. Consequently, these chapters of the book serve as a robust starting point for achieving the aim of this thesis: using Coq to define a formal semantics and proving semantics preservation for translation relations.

[2]`https://softwarefoundations.cis.upenn.edu/`

[3]Shortly before writing this, a sixth book was newly published: "Separation Logic Foundations"

Chapter 3

Preliminary study: a prototype for a toy language

In the proposal phase of this project, we conducted a preliminary study that culminated in a research proposal. In this proposal, we report on the results of semantics preservation proofs for three example transformations in the context of a small, functional, toy language. This toy language is a simply typed lambda calculus, and it is called `ToyLang`. In the proposal, we formalise `ToyLang` both on paper and in Coq. As the (intermediate) languages of the Plutus Tx compiler, such as Plutus Core, are all based on a simply typed lambda calculus as well, we believed that this positive result implied that we could scale up our methods to the Plutus Tx compiler. As such, the proposal serves both as an exposition of the methods we aimed to use when we started scaling up to the Plutus Tx compiler and also as a proof of concept that shows that these methods work in a self-contained setting. We also surmised that scaling up our methods to PIR and Plutus Core would be subject to substantial technical overhead, which proved to be true. Perhaps not surprisingly, the methods used in the main thesis have evolved away from the methods used in the preliminary study. Still, we consider it useful to repeat an adaptation of the largest part of the research proposal here, because it fits well in the thesis' narrative as a self-contained introduction to solving the problem of semantics preservation. Particularly, the structure and narrative of the main thesis are similar to those of the proposal.

We give an overview of the contents of this chapter below. The overview given here is in fact the overview of the research proposal's contents, though we also describe which parts of the research proposal are omitted in this chapter.

- We give the syntax and static semantics of `ToyLang`. (Sections 3.1 and 3.2)
- We construct and compare three possible styles of dynamic semantics (denotational, small-step operational and big-step operational), and we choose to use the big-step operational semantics in the study of semantics preservation because it suits the project best. In this chapter, we omit a detailed consideration of the denotational and small-step operational semantics. Instead, we briefly motivate our choice for a big-step operational semantics. (Section 3.3)
- We make the concept of semantics preservation more precise by distinguishing between the preservation of static semantics and the preservation of dynamic semantics. In the context of dynamic semantics preservation, we define a notion for the semantic equivalence of terms that incorporates the concepts of syntactic equality and extensional equality. We discuss the simplest definitions for static and dynamic semantics preservation. We omit a detailed discussion of more general definitions for static and dynamic semantics preservation because they will not be used in this chapter (Section 3.4)
- Lastly, we prove the semantics preservation property for three example relations: `let`-binding desugaring, inlining of `let`-bound variables and Call-By-Name simulation. More specifically, we prove that all three preserve both static and dynamic semantics. In this chapter, we only discuss the semantics preservation proofs for a single translation relation: desugaring of `let`-bindings. (Section 3.5)

3.1 Syntax

We start the construction of our prototype by defining the syntax of terms and types in `ToyLang`. `ToyLang` itself is a simply typed lambda calculus (STLC) with some small extensions. We extend the pure simply typed lambda calculus with Boolean constants, an `if-then-else` construct, `let`-bindings and a `unit` constant that can be interpreted as an empty tuple. The language and the semantics that we introduce in later sections is entirely standard.

Figure 3.1 defines a BNF grammar that describes the concrete syntax of types τ in the language. It can be translated to the `ty` datatype.

$$\tau ::= \texttt{Bool} \mid \texttt{Unit} \mid \tau_1 \longrightarrow \tau_2$$

Figure 3.1: Concrete syntax of types (preliminary study)

$$t ::= n \mid \lambda\tau.t \mid t_1\ t_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \mid \texttt{unit} \mid \texttt{let}\ t_1 : \tau\ \texttt{in}\ t_2$$

Figure 3.2: Concrete syntax of terms (preliminary study)

```
Inductive ty : Type :=
  | ty_bool : ty
  | ty_unit : ty
  | ty_arrow : ty -> ty -> ty.
```

Concerning terms in the ToyLang language, we have a choice of how to represent variables. We chose to use De Bruijn indices, which have the inherent property that α-equivalence can be determined through syntactic equality. In fact, this property simplifies a number of upcoming proofs. A De Bruijn index consists of natural number n that counts the number of binders on the path to the binding site. Figure 3.2 defines a BNF grammar that describes the concrete syntax of terms t in ToyLang.

Note that both λ-abstractions and `let`-bindings are annotated with a type. This is mostly necessary for type checking purposes, which we will describe in section 3.2. Even with the small amount of type information in it, the abstract syntax of terms is *extrinsically typed*. It is well-known that, next to extrinsically typed abstract syntax, there is one other main approach of representing the abstract syntax: an intrinsically typed abstract syntax. We chose to use an extrinsically typed abstract syntax because that would more closely follow the Programming Language Foundations book[30]. We now define the `tm` datatype in Coq.

```
Definition ref := nat.
Inductive tm : Type :=
  | tm_var : ref -> tm
  | tm_abs : ty -> tm -> tm
  | tm_app : tm -> tm -> tm
  | tm_true : tm
  | tm_false : tm
  | tm_if : tm -> tm -> tm -> tm
  | tm_let : tm -> ty -> tm -> tm
  | tm_unit : tm.
```

With the `Notation` command in Coq, we can introduce custom parsing rules. This allows us to write terms and types in the style of the concrete syntax from Figures 3.1 and 3.2, although with some minor differences. We can write terms using concrete syntax if we put them between <{ and }>, and we can write types using concrete syntax if we put them between <{{ and }}>. The full script containing all `Notation` rules can be found in Appendix A.

```
Declare Custom Entry stlc.
Declare Custom Entry stlc_ty.
Notation "<{ e }>" := e (e custom stlc at level 99).
Notation "<{{ e }}>" := e (e custom stlc_ty at level 99).
(* hidden *)
```

3.2 Static semantics

With the abstract syntax of terms and types defined, we define the first of two elements that make up a well-defined semantics for ToyLang: a static semantics.

The final aim of this project is to scale up from the ToyLang language to the Plutus Tx compiler. Because the Plutus Tx compiler has a type inference algorithm, it is safe to assume that we do not have to write such a type inference algorithm ourselves. Instead, we could simply let the compiler output the type information with which we annotate the abstract syntax. In other words, we assume that we have all necessary type information already at our disposal. It is then straightforward to define typing judgements that characterise a type *checking* algorithm.

$$\Gamma ::= \tau :: \Gamma \mid \emptyset$$

Figure 3.3: Concrete syntax of type contexts (preliminary study)

$$\frac{\Gamma(n) = \tau}{\Gamma \vdash n : \tau}\ \textbf{T-Var} \qquad \frac{\tau_2 :: \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash \lambda\tau_2.t_1 : \tau_2 \to \tau_1}\ \textbf{T-Abs} \qquad \frac{\Gamma \vdash t_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau_1}\ \textbf{T-App}$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathtt{Bool}}\ \textbf{T-True} \qquad \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{Bool}}\ \textbf{T-False} \qquad \frac{\Gamma \vdash t_1 : \mathtt{Bool} \qquad \Gamma \vdash t_2 : \tau \qquad \Gamma \vdash t_3 : \tau}{\Gamma \vdash \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 : \tau}\ \textbf{T-If}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \tau_1 :: \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \mathtt{let}\ t_1 : \tau_1\ \mathtt{in}\ t_2 : \tau_2}\ \textbf{T-Let} \qquad \frac{}{\Gamma \vdash \mathtt{unit} : \mathtt{Unit}}\ \textbf{T-Unit}$$

Figure 3.4: Typing of terms (preliminary study)

First, we define type contexts. To take advantage of the implicit scoping of De Bruijn variables, a type context is defined to be a cons-list of types. Figure 3.3 defines a BNF grammar that describes the concrete syntax of type contexts. Looking up the type of a variable n in a type context consists of looking up the n^{th} element in the context. We denote such a lookup as if Γ is a function: $\Gamma(n) = \tau$. In Coq, contexts are easily represented by built-in lists. The `nth_error` function on lists represents the lookup as we defined earlier.

```
Definition context := list ty.
```

It is now straightforward to define the typing judgements for the type checking algorithm. Figure 3.4 defines the inference rules that represent the typing judgements. The `has_type` datatype is a straightforward translation of these inference rules. Again, the `Notation` command allows us to write types of `has_type` instances similarly to how we would write a typing judgement in Figure 3.4.

```
Reserved Notation "Gamma '|-' t '\in' T" (at level 40, t custom stlc,
  T custom stlc_ty at level 0).
Inductive has_type : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T, nth_error Gamma x = Some T -> Gamma |- {tm_var x} \in T
  | T_Abs : forall Gamma T1 T2 t1, (T2 :: Gamma) |- t1 \in T1 -> Gamma |- \T2, t1 \in (T2 -> T1)
  | T_App : forall T1 T2 Gamma t1 t2,
      Gamma |- t1 \in (T2 -> T1) -> Gamma |- t2 \in T2 ->
      Gamma |- t1 t2 \in T1
  | T_True : forall Gamma, Gamma |- true \in ty_bool
  | T_False : forall Gamma, Gamma |- false \in ty_bool
  | T_If : forall t1 t2 t3 T Gamma,
      Gamma |- t1 \in ty_bool -> Gamma |- t2 \in T -> Gamma |- t3 \in T ->
      Gamma |- if t1 then t2 else t3 \in T
  | T_Let : forall Gamma t1 t2 T1 T2,
      Gamma |- t1 \in T1 -> (T1 :: Gamma) |- t2 \in T2 ->
      Gamma |- let t1 : T1 in t2 \in T2
  | T_Unit : forall Gamma, Gamma |- tm_unit \in ty_unit
where "Gamma '|-' t '\in' T" := (has_type Gamma t T).
```

3.3 Dynamic semantics

We can now turn our attention to constructing a dynamic semantics for `ToyLang`. The three most common styles of defining a dynamic semantics are axiomatic semantics, denotational semantics and operational semantics. An axiomatic semantics is not well-suited for our project, as it is a semantics that deals with programs that are annotated with assertions about program state, which is not a feature of any of the (intermediate) languages of the Plutus Tx compiler. This leaves us with a choice of constructing a denotational or operational semantics.

In the proposal, we consider three implementations: a denotational semantics based on an evaluator for a simply typed lambda calculus written in Agda[37], a small-step operational semantics and a big-step operational semantics. We picked the implementation that best suits our needs by comparing the three implementations. While the proposal contains a more detailed comparison, we summarise the two main takeaways:

- The denotational semantics implementation works well for simple languages such as `ToyLang`, but it would have been complex to extend it to the Plutus Tx compiler and to reason about the semantics. This is mainly due to the fact that the (intermediate) languages of the Plutus Tx compiler feature arbitrary recursion, which adds considerable complexity to the denotational semantics.
- The small-step operational semantics offers fine-grained control in the sense that we can inspect individual computation steps. However, we are often more interested in the *results* of computations in the context of this thesis project, and less in the intermediate computation steps.

Because a big-step operational semantics is focused on results of computations, we decided that the big-step operational semantics suits our needs best, and we therefore use it in both the proposal and the main thesis project. The big-step operational semantics is a substitution semantics, because that is the same approach that is used by the authors of the Programming Language Foundations book[30]. This allows us to adapt many of the theorems, lemmas and proof methods from the book to our setting.

3.3.1 Shifting and substitution

The operational semantics is a substitution semantics, so we need to implement a substitution function. To define a substitution function on free variables that are represented as De Bruijn indices, we need an additional, unary *shift* operation to move terms that are to be substituted below binders such as λ-abstractions and `let`-bindings. A shift is necessary if the term that is to be substituted for a variable contains free variables: every time the term is moved below a binder, all its free variables should be incremented by 1.

Let $t \uparrow^k$ denote that the free variables inside a term t are shifted up by k. To define a shift operation in Coq, we define a slightly more general `shift'` function that is parameterised by k and a cutoff value c. The cutoff value serves as an accumulating parameter that describes how many binders have been traversed, such that we can judge whether a variable n is free.

```
Definition shift' : nat -> nat -> tm -> tm :=
  fix shift' k c t :=
    match t with
      | tm_var n => if n <? c then tm_var n else tm_var (k + n)
      | <{ \T, t }> => <{ \T, {shift' k (S c) t} }>
      | <{ t1 t2 }> => <{ ({ shift' k c t1 }) ({ shift' k c t2 }) }>
      | <{ true }> => <{ true }>
      | <{ false }> => <{ false }>
      | <{ if t1 then t2 else t3 }> => <{ if {shift' k c t1} then {shift' k c t2}
                                          else {shift' k c t3} }>
      | <{ let t1 : T1 in t2 }> => <{ let {shift' k c t1} : T1 in ({shift' k (S c) t2}) }>
      | <{ unit }> => <{ unit }>
    end.
```

The `shift'` function applied to an initial cutoff value of 0 computes an actual shift. Lastly, we define some convenient definitions and notations.

```
Definition shift : tm -> tm := shift' 1 0.
Notation "t '^'" := (shift t) (in custom stlc at level 1).
```

With a `shift` operation implemented, we can turn our attention to implementing a substitution function. Using the `Equations` package, we can write the implementation through dependent pattern matching, though it would have been perfectly fine to write substitution as a `Fixpoint` as well. The function `subst j s t`, or alternatively written in concrete syntax as `<{ [j := s] t }>`, replaces a variable j in the term t by a term s.

```
Reserved Notation "'[' n ':=' s ']' t" (in custom stlc at level 20, n constr).
Equations subst : ref -> tm -> tm -> tm :=
  subst j s (tm_var n) := if j =? n then s else (tm_var n) ;
  subst j s <{ \T, t }> := <{ \T, [1+j:=s^] t }> ;
  subst j s <{ t1 t2 }> := <{ ([j:=s] t1) ([j:=s] t2) }> ;
  subst j s <{ true }> := <{ true }> ;
  subst j s <{ false }> := <{ false }> ;
  subst j s <{ if t1 then t2 else t3 }> := <{ if [j:=s] t1 then [j:=s] t2 else [j:=s] t3 }> ;
  subst j s <{ let t1 : T1 in t2 }> := <{ let [j:=s] t1 : T1 in ([1+j:=s^] t2) }> ;
  subst j s <{ unit }> := <{ unit }>;
where "'[' j ':=' s ']' t" := (subst j s t) (in custom stlc).
```

$$\frac{}{\lambda\tau.t \in \mathit{value}}\ \textbf{V-Abs} \qquad \frac{}{\texttt{true} \in \mathit{value}}\ \textbf{V-True} \qquad \frac{}{\texttt{false} \in \mathit{value}}\ \textbf{V-False} \qquad \frac{}{\texttt{unit} \in \mathit{value}}\ \textbf{V-Unit}$$

Figure 3.5: Values (preliminary study)

$$\frac{}{\lambda\tau.t \Downarrow \lambda\tau.t}\ \textbf{E-Abs} \qquad \frac{t_1 \Downarrow \lambda\tau.t_0 \qquad t_2 \Downarrow v_2 \qquad [0 := v_2]\, t_0 \Downarrow v_0}{t_1\ t_2 \Downarrow v_0}\ \textbf{E-App}$$

$$\frac{}{\texttt{true} \Downarrow \texttt{true}}\ \textbf{E-True} \qquad \frac{t_1 \Downarrow \texttt{true} \qquad t_2 \Downarrow v_2}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow v_2}\ \textbf{E-IfTrue}$$

$$\frac{}{\texttt{false} \Downarrow \texttt{false}}\ \textbf{E-False} \qquad \frac{t_1 \Downarrow \texttt{false} \qquad t_3 \Downarrow v_3}{\texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \Downarrow v_3}\ \textbf{E-IfFalse}$$

$$\frac{t_1 \Downarrow v_1 \qquad [0 := v_1]t_2 \Downarrow v_2}{\texttt{let}\ t_1 : \tau_1\ \texttt{in}\ t_2 \Downarrow v_2}\ \textbf{E-Let} \qquad \frac{}{\texttt{unit} \Downarrow \texttt{unit}}\ \textbf{E-Unit}$$

Figure 3.6: Big-step operational semantics (preliminary study)

Though most cases of the pattern match are trivial, take notice of the case for abstractions and `let`-bindings. Moving beneath a binder results in both shifting the term that is to be substituted, and incrementing the index of the variable that is the target of the substitution. This is to account for the implicit scoping of De Bruijn indices.

3.3.2 Big-step operational semantics

A big-step operational semantics is focused on results of computations. A first step in constructing such a semantics is to identify the possible result values that a term can evaluate to. In the case of the `ToyLang` language, such values can be either constants, such as `true`, `false` and `unit`, or λ-abstractions. The inference rules in Figure 3.5 describe all inhabitants of the set of *values*. The `value` datatype is a direct implementation of these rules.

```
Inductive value : tm -> Prop :=
  | v_abs : forall T t, value <{ \T, t }>
  | v_true : value <{ true }>
  | v_false : value <{ false }>
  | v_unit : value <{ unit }>.
```

With a notion of values formalised, we can define the different ways in which a term can evaluate to a value. We define a binary *evaluation* relation on terms, called `eval` and denoted by $\Downarrow$, that fully evaluates terms to values. Figure 3.6 describes the evaluation relation.

When we consider the rules in Figure 3.6, we can observe two categories of rules. On one hand, we have rules that describe how values evaluate to themselves. For example, the **E-Abs** rule states that a λ-abstraction evaluates to the same λ-abstraction. On the other hand, we have rules that describe how components of a constructor must be evaluated and combined to obtain the eventual result value. For example, in the **E-App** rule both terms first have to be evaluated to values after which we can substitute the latter term in the former term, and evaluate the result of the substitution further. It is now straightforward to translate the rules to the `eval` inductive datatype.

```
Reserved Notation "t '==>' v" (at level 40).
Inductive eval : tm -> tm -> Prop :=
  | E_Abs: forall T t, <{ \T, t }> ==> <{ \T, t }>
  | E_App: forall t1 T t0 v0 t2 v2, t1 ==> <{ \T, t0 }> -> t2 ==> v2 ->
     <{ [0 := v2] t0 }> ==> v0 -> <{ t1 t2 }> ==> v0
  | E_True : <{ true }> ==> <{ true }>
  | E_False : <{ false }> ==> <{ false }>
  | E_IfTrue : forall t1 t2 v2 t3, t1 ==> <{ true }> -> t2 ==> v2 ->
     <{ if t1 then t2 else t3 }> ==> v2
  | E_IfFalse : forall t1 t2 t3 v3, t1 ==> <{ false }> -> t3 ==> v3 ->
     <{ if t1 then t2 else t3 }> ==> v3
  | E_Let : forall t1 v1 T1 t2 v2, t1 ==> v1 -> <{ [0 := v1] t2 }> ==> v2 ->
     <{ let t1 : T1 in t2 }> ==> v2
  | E_Unit : <{ unit }> ==> <{ unit }>
where "t '==>' v" := (eval t v).
```

$$\frac{\begin{array}{c}\emptyset \vdash t_1 : \tau \\ \emptyset \vdash t_2 : \tau \\ t_1 \Downarrow v_1 \\ t_2 \Downarrow v_2 \\ v_1 = v_2\end{array}}{t_1 \equiv_\tau t_2}\ \textbf{EQV-Syntactic} \qquad \frac{\begin{array}{c}\emptyset \vdash t_1 : \tau' \to \tau \\ \emptyset \vdash t_2 : \tau' \to \tau \\ t_1 \Downarrow \lambda\tau'.t_3 \\ t_2 \Downarrow \lambda\tau'.t_4 \\ \forall s_1.\forall s_2.\ s_1 \equiv_{\tau'} s_2 \text{ implies } (\lambda\tau'.t_3)s_1 \equiv_\tau (\lambda\tau'.t_4)s_2\end{array}}{t_1 \equiv_{\tau'\to\tau} t_2}\ \textbf{EQV-Extensional}$$

Figure 3.7: Semantic equivalence (preliminary study)

3.4 Semantics preservation

Before we tackle any proofs about translation relations and their correctness, we must define the concept of semantics preservation. Recall from Definition 2.2 that we consider some translation relation R_i to be correct if $R_i(t_i, t_{i+1})$ implies $[\![t_i]\!]_i \sim_i [\![t_{i+1}]\!]_{i+1}$, where $[\![t_i]\!]_i$ is a semantics of an AST t_i, and where $\sim_i$ is a refinement relation. We make the concept of semantics preservation more precise for both static and dynamic semantics.

In the research proposal, we consider two types of translation relations: translation relations that do not change types and translation relations that do change types. For the sake of the narrative, we only consider translation relations that do not change types in this chapter. For an explanation of semantics preservation in the context of translation relations that change types, we refer the reader to the research proposal.

3.4.1 Static semantics preservation

We consider static semantics preservation for translation relations that change only the syntax of terms and not their types. The desugaring of `let`-bindings is an example of such a translation relation. It would be sufficient to prove that both the source and target term have the same type with respect to the same type context:

Definition 3.1 (Static semantics preservation (preliminary study)). *For all t_i and t_{i+1}, if $R_i(t_i, t_{i+1})$ and $\Gamma \vdash t_i : \tau$, then $\Gamma \vdash t_{i+1} : \tau$.*

Still, this formulation of static semantics preservation does not fully describe all theorems that we might need to prove. For example, consider an inlining transformation that inlines `let`-bound variables, such as the one characterised in [17]. The translation relation is parameterised by some environment of `let`-bound terms, and this environment has a correspondence with the type context. Namely, the types of `let`-bound terms are exactly the types that are also in the type context, and we must assume this in some way for the proof to go through. Therefore, it might be that we need additional assumptions to finish a proof of static semantics preservation. Nonetheless, Definition 3.1 sketches the general outline of static semantics preservation.

3.4.2 Dynamic semantics preservation

We interpret preservation of dynamic semantics to mean that the source and target term should be semantically equivalent, i.e., they compute the same result. We use the $\equiv$-sign to denote semantic equivalence, and we define dynamic semantics preservation as follows:

Definition 3.2 (Dynamic semantics preservation (preliminary study)). *For all t_i and t_{i+1}, if $R_i(t_i, t_{i+1})$ then $t_i \equiv t_{i+1}$.*

Now, consider an example: intuitively, a term t_i computes the same result as a term t_{i+1} in which some of the `let`-bindings are desugared. However, we must be more precise about what semantic equivalence means. If a source and target term both terminate with values of `Bool` or `Unit` type, we can determine semantic equivalence by syntactic equality on the values, but this does not necessarily work for terms that compute λ-abstractions as their result. This is a problem in the context of transformations such as desugaring: our dynamic semantics does not reduce below the λ-abstraction value, but it may be the case that the desugaring transformation has desugared `let`-bindings in the body of the λ-abstraction. Consequently, the values corresponding to the source and target term are not syntactically equal. However, even if the abstractions represent the same function and they are not syntactically equal, then they should be extensionally equal. That is, the abstractions are considered equivalent if they have the same observable behaviour.

Furthermore, we must recognize that semantic equivalence should be *type-directed*. The semantically equivalent terms must then be closed and well-typed with respect to the same type τ. This is a sensible requirement, as terms that do not have (α-)equivalent types will not compute the same result. Moreover, the evaluation relation is a substitution semantics, so we always work with closed terms. As such, we now use $\equiv_\tau$ to describe semantic equivalence in the context of a type τ. We formalise semantic equivalence using the two inference rules in Figure 3.7.

We have defined semantic equivalence in such a way that we have defined a *logical relation*. A logical relation is a relation between well-typed terms that is defined by induction on types. The *logical relations proof technique* has been used, for example, to prove the strong normalisation property for a simply typed lambda calculus[38].

We also need a representation of semantic equivalence in Coq. The approach of our choice is inspired by the chapter on Normalisation in the Programming Language Foundations book[30]. This chapter uses the logical relations proof technique in Coq to prove that a simply typed lambda calculus is strongly normalising. We name the logical relation RSE, in the sense that it is a Relation (R) that describes Semantic Equivalence (SE), and it is represented as a `Fixpoint` defined over a `ToyLang` type and two terms. Its definition is akin to the inference rules in Figure 3.7.

```
Fixpoint RSE (T : ty) (t1 t2 : tm) : Prop :=
  [] |- t1 \in T /\ [] |- t2 \in T /\
  (exists v1 v2,
      t1 ==> v1 /\ t2 ==> v2 /\
      (match T with
        | <{{ Unit }}> => True
        | <{{ Bool }}> => v1 = v2
        | <{{ T2 -> T1 }}>=>
          (forall s t,
              RSE T2 f g s t ->
              RSE T1 f g <{ v1 s }> <{ v2 t }>
          )
      end)
  ).
```

3.5 Correctness of transformations

In the proposal, we consider three example translation relations: desugaring of `let`-bindings, inlining of `let`-bound variables and Call-By-Name simulation. We show that all three translation relations are static and dynamic semantics preserving. The goal of these results is exploratory: the fact that we can show that the translations are semantics preserving inspired confidence that these concepts could be applied in the Plutus Tx compiler and that our approach would scale up. We did not aim to perform an exhaustive study of semantics preservation in the context of a simply typed lambda calculus.

In this chapter we will only repeat the semantics preservation results for one of the translation relations: desugaring of `let`-bindings. We refer the interested reader to the research proposal for the results concerning the other translation relations.

As a notational convention, we use $A \vdash t \triangleright t'$ to denote that we can derive from the *contextual information* in A that a term t transforms to a term t'. The environment A stores any information we may need to define the transformation. Naturally, the environment A can also be left out if the transformation does not depend on any information. For example, the translation relation for desugaring does not depend on any contextual information, while the translation relation for inlining requires contextual information about all `let`-bound terms that are in scope.

Note on proof completeness. Crucially, the proofs about semantics preservation rely on a few key properties: (i) Progress, (ii) Preservation, (iii) Determinism, (iv) `eval` preserves RSE, and (v) substitution preserves typing. While we have constructed proofs for (iii) and (iv), we have `Admitted` (i), (ii) and (v) because of time constraints. Since it is known that these last three properties hold for a simply typed lambda calculus with a well-defined semantics, we believe that it is justified to use the `Admitted` theorems and lemmas. We hope that this paragraph has offered sufficient transparency with regards to the completeness of our proofs.

3.5.1 Desugaring of `let`-bindings

The desugaring translation relation, which we denote by R_{DS}, is a simple one. The changes that the translation relation makes to terms are only *local*, because the transformation does not depend on any contextual information. Figure 3.8 defines R_{DS}. Clearly, only the **DS-LetApp** rule is an interesting case, as all other rules are simple congruence cases. Also, note that desugaring is optional, as we have two possible rules that apply to `let`-constructs.

The inference rules are directly translated into an inductive datatype `desugar` that is indexed by both a source and target term. Again, the only interesting constructor of the datatype is the `DS_LetApp` constructor. Even more so because an observant reader can see that that the desugaring translation justifies annotating `let`-bindings with a type. For type checking purposes, it would have sufficed to only annotate λ-abstractions with a type. However, when we desugar a `let`-binding we suddenly have to annotate the newly constructed λ-abstraction with a type. If we had not annotated `let`-bindings with the type of the bound term, we would have not been able to define the translation relation easily.

$$\frac{}{\vdash n \triangleright n}\ \textbf{DS-Var} \qquad \frac{\vdash t_1 \triangleright t_1'}{\vdash \lambda\tau_1.t_1 \triangleright \lambda\tau_1.t_1'}\ \textbf{DS-Abs} \qquad \frac{\vdash t_1 \triangleright t_1' \quad \vdash t_2 \triangleright t_2'}{\vdash t_1\ t_2 \triangleright t_1'\ t_2'}\ \textbf{DS-App}$$

$$\frac{}{\vdash \texttt{true} \triangleright \texttt{true}}\ \textbf{DS-True} \qquad \frac{}{\vdash \texttt{false} \triangleright \texttt{false}}\ \textbf{DS-False} \qquad \frac{}{\vdash \texttt{unit} \triangleright \texttt{unit}}\ \textbf{DS-Unit}$$

$$\frac{\vdash t_1 \triangleright t_1' \quad \vdash t_2 \triangleright t_2' \quad \vdash t_3 \triangleright t_3'}{\vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 \triangleright \texttt{if}\ t_1'\ \texttt{then}\ t_2'\ \texttt{else}\ t_3'}\ \textbf{DS-If}$$

$$\frac{\vdash t_1 \triangleright t_1' \quad \vdash t_2 \triangleright t_2'}{\vdash \texttt{let}\ t_1 : \tau_1\ \texttt{in}\ t_2 \triangleright \texttt{let}\ t_1' : \tau_1\ \texttt{in}\ t_2'}\ \textbf{DS-Let} \qquad \frac{\vdash t_1 \triangleright t_1' \quad \vdash t_2 \triangleright t_2'}{\vdash \texttt{let}\ t_1 : \tau_1\ \texttt{in}\ t_2 \triangleright (\lambda\tau_1.t_2')\ t_1'}\ \textbf{DS-LetApp}$$

Figure 3.8: Desugaring translation relation (R_{DS}) (preliminary study)

```
Inductive desugar : tm -> tm -> Prop :=
  (* hidden *)
  | DS_LetApp : forall {t1 t1' T1 t2 t2'},
      desugar t1 t1' ->
      desugar t2 t2' ->
      desugar <{ let t1 : T1 in t2 }> <{ (\T1, t2') t1' }>
  (* hidden *).
```

Static semantics preservation. We prove the following theorem:

Theorem 3.3 (R_{DS} preserves static semantics (preliminary study))**.** *For all t_1 and t_2, if $R_{DS}(t_1, t_2)$ and $\Gamma \vdash t_1 : \tau$, then $\Gamma \vdash t_2 : \tau$.*

Proof. We proceed to prove this by induction over the typing judgement. Note that desugaring does not change the type annotations of our terms. Combined with the fact that changes are only local makes the proof of static semantics preservation straightforward. As such, it can be concisely constructed in Coq using proof automation, so we list it fully here.

```
Theorem desugar_preserves_typing : forall t1 t2 T Gamma,
    desugar t1 t2 ->
    Gamma |- t1 \in T ->
    Gamma |- t2 \in T.
Proof. intros t1 t2 T Gamma Hds Ht1. generalize dependent t2.
  induction Ht1; try solve [intros; inversion Hds; subst; eauto].
Qed.
```

□

Dynamic semantics preservation. We prove the following theorem:

Theorem 3.4 (R_{DS} preserves dynamic semantics (preliminary study))**.** *For all t_1 and t_2, if $R_{DS}(t_1, t_2)$ then $t_1 \equiv t_2$.*

Proof. The formulation of Theorem 3.4 differs slightly from its definition in Coq. There, we also assume that t_1 and t_2 are well-typed under the empty context. This is justified because we can derive from just these two assumptions that t_1 and t_2 are reducible by the Progress property. Additionally, we can safely assume that t_2 has the same type as t_1 under the empty context, because we have just proved that desugaring preserves static semantics.

```
Theorem desugar_preserves_semantics : forall t1 t2 T,
    [] |- t1 \in T ->
    [] |- t2 \in T ->
    desugar t1 t2 ->
    RSE_ID T t1 t2.
Proof. intros t1 t2 T Htyp1. generalize dependent t2.
  induction Htyp1.
  (* hidden *).
Abort.
```

If we try to prove the theorem by induction on the first typing judgement, the proof does not work out. This is the case because the first assumption states that `t1` should be a closed, well-typed term, but this is results in a too restrictive induction hypothesis. Instead, we must first prove a more general lemma that is defined over all well-typed, closed *instances* of terms. To this end, we employ an auxiliary `instantiation` datatype that is indexed by a type context and two environments, which are just lists of terms.

```
Inductive instantiation : context -> env -> env -> Prop :=
  | V_nil :
      instantiation [] [] []
  | V_cons : forall T t1 t2 c e1 e2,
      RSE_ID T t1 t2 ->
      instantiation c e1 e2 ->
      instantiation (T :: c) (t1 :: e1) (t2 :: e2).
```

The `instantiation` datatype records that the elements of the two type-index environments are pair-wise semantically equivalent, i.e., every i^{th} element of the first environment and every i^{th} element of the second environment are semantically equivalent with respect to the i^{th} type in the context. We could overload the notion of semantic equivalence and say that the *environments* are semantically equivalent.

Also, consider some function `msubst` that given an environment `ss` and a term `t` performs the left-to-right substitution of all terms in `ss` in `t`.

```
Definition msubst (ss:env) (t:tm) : tm := (* hidden *).
Example msubst_example : forall e0 e1 e2 t,
    msubst [e0; e1; e2] t = <{ [2 := e2] ([1 := e1] ([0 := e0] t)) }>.
Proof. reflexivity. Qed.
```

Now, why are `instantiation` and `msubst` useful? Given that two terms `s` and `t` have some type `T` under the same context `c`, and given that we have some instantiation `instantiation c es et`, then it must be the case that both `msubst es s` and `msubst es t` are closed and well-typed with respect to `T`! Even more so, if we are also given that `s` and `t` are related by the desugaring translation, we can prove that `msubst es s` and `msubst et t` are semantically equivalent. This leads to the more generalised lemma that we need to prove to finish the proof of Theorem 3.4.

```
Lemma msubst_RSE : forall c es et s t T,
    c |- s \in T ->
    c |- t \in T ->
    instantiation c es et ->
    desugar s t ->
    RSE_ID T (msubst es s) (msubst et t).
Proof. (* hidden *). Qed.
```

The proof of the lemma proceeds by induction over the first typing judgement. As the proof is rather longwinded, we do not list it here. Instead, we note that the proof relies on a number of lemmas and theorems about (the interactions between) nearly all previous definitions: the evaluation relation (`eval`), semantic equivalence (`RSE`), instantiations (`instantiation`), substitutions (`subst`), multiple substitutions (`msubst`), shifting (`shift`), typing (`has_type`) and so on. Maybe most importantly, the proof relies on the key properties that are described in the **Note on proof completeness** paragraph at the start of this section.

Finally, we can finish the proof of Theorem 3.4.

```
Theorem desugar_preserves_semantics : forall t1 t2 T,
    [] |- t1 \in T ->
    [] |- t2 \in T ->
    desugar t1 t2 ->
    RSE_ID T t1 t2.
Proof.
  intros t1 t2 T Htyp1 Htyp2 Hds.
  replace t1 with (msubst [] t1) by reflexivity.
  replace t2 with (msubst [] t2) by reflexivity.
  eapply msubst_RSE; eauto.
Qed.
```

□

Chapter 4

PIR and Plutus Core

Plutus Core is an extended version of System F_ω^μ adding higher-order kinds and iso-recursive types[7]. PIR is a superset of System F_ω^μ that includes non-recursive `let`-bindings of terms, types and datatypes, and (mutually) recursive `let`-bindings of terms and datatypes[15]. In this chapter, we formalise the syntax and semantics of the PIR and Plutus Core languages. Since PIR is a superset of Plutus Core, we only formalise PIR and note that we will then have formalised Plutus Core in the same process. We use the aforementioned papers[7, 15] and the Plutus Tx compiler implementation[1] as the basis for this chapter. Naturally, we follow the compiler implementation whenever the papers and the compiler implementation do not agree.

We define and implement the syntax of terms, types and kinds in section 4.1. We define and implement the static semantics of PIR as kind and type checking algorithms in section 4.2, where we design the algorithms such that we only type check for normal types. We define and implement the dynamic semantics of PIR as a big-step operational semantics in section 4.3.

Note on notational conventions. Like Peyton Jones et al.[15], we adopt Guy Steele's suggestions for metalanguage conventions[16]. The most important takeaway from these suggestions is the use and meaning of overlines and underlines to expand sequences. Furthermore, we denote substitution of s for x in t by $[s/x]\ t$.

Note on the use of previous work and our contributions. Most of the work in this chapter is based directly on work by Peyton Jones et al.[15], Chapman et al.[7], the developers behind the Plutus Tx compiler, and previous work by Krijnen et al[17]. For example, most of the figures in this chapter are adaptations of figures from the paper by Peyton Jones et al.[15]. Furthermore, we borrow ideas such as type equality and normalisation of types from both Peyton Jones et al.[15] and Chapman et al.[7]. Moreover, the implementation of type normalisation is more akin to the compiler implementation of type normalisation than it is akin to type normalisation as defined in the two mentioned papers. The implementation of the language syntax in Coq was performed by Krijnen et al[17]. We hope that the reader understands that this dependence on previous work is not surprising, and we hope that this paragraph gives sufficient credit to the authors. Other than adapting and re-iterating previous works to produce a formalisation of PIR and Plutus Core, we do present two novelties: (i) a definition for the formal dynamic semantics of some PIR-specific language constructs, and (ii) a Coq implementation of a static and dynamic semantics for PIR and Plutus Core.

Note on limitations. The static semantics that we present here differs from the compiler implementation in two significant aspects. Firstly, we do not allow type bindings to escape in `let`-bindings, whereas the compiler implementation allows this only if the `let`-binding is top-level. It is a problem for future to work to solve this difference, since allowing escaped type bindings at the top-level requires some non-trivial programming and proof engineering that we were not able to perform due to time constraints. The other aspect in which our static semantics differs from the compiler implementation is that an error can have *any* normalised type as opposed to having the type that it is annotated with. For now, we have sacrificed principal typing for the sake of type Preservation, which would have been jeopardised if we did not allow for errors to have any normalised type. With regards to the dynamic semantics that we present in this chapter, it is not yet complete for the same reason of time constraints. Specifically, we do not give evaluation/reduction rules for datatype bindings, we ignore strictness flags for term bindings, we do not yet model laziness where it is sometimes needed, and the semantics of some built-in functions are not yet correctly modelled. We do not think that the limitations that we have mentioned in this paragraph undermine the positive results of the thesis. Rather, we think that solutions to these problems will mostly consist of technical overhead.

[1]`https://github.com/input-output-hk/plutus/`

kinds	$K, J ::= *$	base kind
	$\mid\ K \to J$	function kind

Figure 4.1: Concrete syntax of kinds

4.1 Syntax

In this section, we define the concrete and abstract syntax of terms, types, kinds and some necessary auxiliary datatypes in PIR by mimicking the Plutus Tx compiler implementation. We do this because at this stage in the development of the certification engine, we dump the ASTs of the compiler. These ASTs are then copied straight into a Coq proof script, and the abstract syntax is designed such that these ASTs are also valid abstract syntax in the certification engine. Because of this, ASTs may contain artifacts that are particular to the Haskell implementation of the compiler, though this has not proven to be an insurmountable problem during development.

We define concrete syntax using BNF grammars, while we define abstract syntax by providing snippets of Coq proof scripts. Because the static semantics, which we will define in section 4.2, depends on kind and type annotations, both our concrete and abstract syntax are explicit about such as annotations.

4.1.1 Variables and binders

The abstract syntax is parameterised by the representation of both (type) variables and (type) binders, meaning that we can instantiate the abstract syntax with custom representations. This level of generalisation allows us to experiment, for example, with De Bruijn indices. We use Coq's `Section` mechanism to generalise over all possible variable and binder representations.

```
Section AST_term.
  Context (name tyname : Set).
  Context (binderName binderTyname : Set).
  (* abstract syntax definitions hidden *)
End AST_term.
```

Throughout this thesis, we use `strings` as both (type) variable and (type) binder representations. We refer to such abstract syntax as `Named`. We do not use De Bruijn indices in order to stay true to the compiler implementation, which uses unique integers to represent variables and binders. We could easily change our abstract syntax to also use integers, which is bound to happen in the future, but it is not relevant for this thesis.

To instantiate the abstract syntax, we define a `Module` containing `Notation` commands that instantiate the abstract syntax with the required representations. As a convention, we define representation-general abstract syntax using lower camelcase identifiers. For example, the representation-general abstract syntax for terms is defined using the inductive datatype `term`. As another convention, we define instantiated abstract syntax using upper camelcase identifiers, such as `Term`.

```
Module NamedTerm.
  (* hidden *)
  Notation name := string. Notation tyname := string.
  Notation binderName := string. Notation binderTyname := string.
  (* hidden *)
  Notation Term := (term name tyname binderName binderTyname).
  (* hidden *)
End NamedTerm.
```

In both the concrete and abstract syntax, we will often use lower-case letters x, y, z and possibly subscripted versions to denote both term-level variables and binders. For type-level variables and binders, we use the upper-case equivalents of those letters.

4.1.2 Kinds

Kinds consist of base kinds and function kinds. Figure 4.1 defines a BNF grammar that represents the concrete syntax of kinds. The Coq implementation of kinds is straightforward.

types	$T, U ::= T \rightarrow U$	arrow type
	$\mid \mathbb{U}_i$	built-in type
	$\mid X$	type variable
	$\mid \forall X :: K.T$	universal type
	$\mid$ `ifix` $T\ U$	fixpoint type
	$\mid \lambda X :: K.T$	lambda abstraction type
	$\mid T\ U$	function application type

Figure 4.2: Concrete syntax of types

```
Inductive kind :=
  | Kind_Base : kind
  | Kind_Arrow : kind -> kind -> kind.
```

4.1.3 Built-in types

Types can be, amongst others, built-in types. The compiler's handling of built-in types is not trivial, and since we mimic the compiler we have to define some additional machinery to handle them. First, we define a *universe* $\mathbb{U}$ as a collection of possible built-in types, where we denote some i^{th} built-in type in the universe by $\mathbb{U}_i$. For example, the default universe in the compiler consists of integers, bytestrings, strings, characters, units and Booleans. The default universe in Coq is just a datatype with empty constructors.

```
Inductive DefaultUni : Type :=
  | DefaultUniInteger
  (* hidden *).
```

We will use the default universe of built-in types for the remainder of this thesis. We now define two additional datatypes: one that represents existential quantification as a datatype, and one that represents a built-in type in the default universe.

```
Inductive some {f : DefaultUni -> Type} := Some : forall {u : DefaultUni}, f u -> some.
Inductive typeIn (u : DefaultUni) := TypeIn : typeIn u.
```

We can now write *some* integer built-in type as `Some (TypeIn DefaultUniInteger)`. This machinery is mostly an artifact that is particular to the Haskell implementation of the compiler, where they use type parameters of datatype definitions to parameterise the abstract syntax with a universe of built-in types. We could have omitted this machinery by using the Coq `Module` system, but we stay true to the compiler implementation. Note that we do not parameterise over universes of built-in types in this thesis, although this parameterisation would not be very hard to implement.

4.1.4 Types

Figure 4.2 defines a BNF grammar that represents the concrete syntax of types. The main type language consists of function types and built-in types (base types). On top of that, we have universal quantification using universal types and type variables. Lastly, the type language allows for recursive types through *computations in types* using a type-level fixpoint operator, type-level abstractions, type-level applications and type variables. What is particular about the type-level fixpoint operator `ifix` is that it takes fixpoints only at kind $K \rightarrow *$. The canonical and less restrictive fixpoint `fix` would take fixpoints at any kind K, but `ifix` is sufficient for the purpose of PIR. Furthermore, it is easier to define typing rules for the `ifix` operator than for the `fix` operator. For a more detailed consideration of type-level fixpoint operators, see section 3.1 of Peyton Jones et al[15]. The implementation of types is now straightforward.

```
Inductive ty :=
  | Ty_Var : tyname -> ty
  | Ty_Fun : ty -> ty -> ty
  | Ty_IFix : ty -> ty -> ty
  | Ty_Forall : binderTyname -> kind -> ty -> ty
  | Ty_Builtin : @some typeIn -> ty
  | Ty_Lam : binderTyname -> kind -> ty -> ty
  | Ty_App : ty -> ty -> ty.
```

vdecl	$vd ::= x : T$	variable declaration
tvdecl	$tvd ::= X :: K$	type declaration
constr	$c ::= x\ (T)$	constructor
dtdecl	$dtd ::= \texttt{data}\ (tvd)\ (\overline{tvd'}) = \bar{c}\ \texttt{with}\ x$	datatype declaration

Figure 4.3: Concrete syntax of declarations

4.1.5 Declarations and datatypes

Before we can define the syntax of terms, we have to define the syntax of variable declarations, type declarations, datatype declarations and constructors. These are the essential constructs with which we can define *bindings*. Figure 4.3 defines a BNF grammar that represents the concrete syntax of declarations and constructors.

Variable and type variable declarations are rather straightforward: the former declares a variable and its type, while the latter declares a type variable and its kind. Constructors declare a constructor name and the type of the constructor. A datatype declaration is more complex. The left-hand side of the declaration defines the name of the datatype as a type variable declaration, and it defines a sequence of type variable declarations that represent the type variables that the datatype is parameterised with. The right-hand side of the declaration consists of the name of a pattern matching function, and a sequence of constructors for the datatype. As an example, we could write the `Either` datatype from Haskell in PIR as follows:

$$\texttt{data}\ (\text{Either} :: * \longrightarrow * \longrightarrow *)\ (A :: *, B :: *) = \text{Left}\ (A \longrightarrow \text{Either}\ A\ B), \text{Right}\ (B \longrightarrow \text{Either}\ A\ B)\ \texttt{with}\ \text{matchEither}$$

The Coq implementation now follows easily, with a minor difference with respect to Figure 4.3: constructors also store a natural number that represents the arity of a constructor. This is mostly an artifact of earlier versions of the certification engine.

```
Inductive vdecl := VarDecl : binderName -> ty -> vdecl.
Inductive tvdecl := TyVarDecl : binderTyname -> kind -> tvdecl.
Inductive constr := Constructor : vdecl -> nat -> constr.
Inductive dtdecl := Datatype : tvdecl -> list tvdecl -> binderName -> list constr -> dtdecl.
```

4.1.6 Terms

Finally, we can define terms, which are mutually defined with bindings. Figure 4.4 defines a BNF grammar that represents the concrete syntax of terms and bindings. We elaborate on distinct parts of the language before we give the Coq implementation.

Simply typed lambda calculus. The basis of the language consists of the usual elements of a simply typed lambda calculus: variables, lambda abstractions and function applications.

Universal types. Universal quantification over types is facilitated by type abstraction and type instantiation.

Recursive types. Iso-recursive types require folding and unfolding of recursive types. An `IWrap` term converts an unfolded recursive type to a folded recursive type, while an `Unwrap` term converts a folded recursive type to an unfolded recursive type.

Constants. We annotate constant values with their built-in types. A first ingredient that we need to implement constants is to embed built-in types as Coq Types by define a mapping from constructors in the universe of built-in types to Coq Types.

```
Definition uniType (x : DefaultUni) : Type :=
  match x with
    | DefaultUniInteger  => Z
    (* hidden *)
  end.
```

terms	$t, u ::= x$	variable
	$\mid \lambda x : T.t$	lambda abstraction
	$\mid t\ u$	function application
	$\mid \Lambda X :: K.t$	type abstraction
	$\mid t\ \{T\}$	type instantiation
	$\mid \texttt{IWrap}\ T\ U\ t$	wrap/fold
	$\mid \texttt{Unwrap}\ t$	unwrap/unfold
	$\mid \texttt{Constant}\ \mathbb{U}_i\ c$	constant
	$\mid \mathbb{F}_i$	built-in function
	$\mid \texttt{Error}\ T$	error
	$\mid \texttt{let}\ (rec)\ \overline{b}\ \texttt{in}\ t$	let
bindings	$b ::= vd\ (str) = t$	term binding
	$\mid tvd = T$	type binding
	$\mid dtd$	datatype binding
recursivity	$rec ::= \texttt{NonRec}$	not recursive
	$\mid \texttt{Rec}$	recursive
strictness	$str ::= \texttt{NonStrict}$	not strict
	$\mid \texttt{Strict}$	strict

Figure 4.4: Concrete syntax of terms

In a similar fashion to how we defined `typeIn` to represent a built-in type in the default universe, we also define a datatype that represents a value of a built-in type.

```
Inductive valueOf (u : DefaultUni) := ValueOf : uniType u -> valueOf u.
```

We can now write *some* Boolean `true` value as `Some (ValueOf DefaultUniBool true)`. Similarly to the machinery for built-in types, this machinery for constants is mostly an artifact of the compiler implementation in Haskell.

Built-in functions. For built-in functions, we define a collection of possible built-in functions, denoted by $\mathbb{F}$, similarly to how we defined a universe $\mathbb{U}$ for built-in types. We denote some i^{th} built-in function in $\mathbb{F}$ by $\mathbb{F}_i$. Again similarly, there exists a default collection of built-in functions. These defaults include, amongst others, simple operations such as integer arithmetic operations, but more complex operations such as hashing as well. We use only this default collection throughout the rest of the thesis, although parameterisation over sets of built-in functions would not be very hard to implement.

```
Inductive DefaultFun :=
  | AddInteger
  (* hidden *)
  | SHA2
  (* hidden *).
```

Errors. Errors are annotated with their type, because without it we would not always be able to find the most principal type when performing type inference or type checking. We could type an error without an annotation as $\forall a :: K.a$, but this is not necessarily the most general type. As such, the type annotation signals exactly what type the error should have.

`let`-bindings. The addition of `let`-bindings to Plutus Core is the main feature of PIR. A `let`-binding can be either (mutually) recursive or non-recursive. Term bindings can be either strict or non-strict. The recursivity and strictness flags are implemented as empty constructors of inductive datatypes.

```
Inductive Recursivity := NonRec | Rec.
Inductive Strictness := NonStrict | Strict.
```

kind contexts	$\Delta ::= \emptyset$	empty
	$\mid\ \Delta, X :: K$	type variable binding
type contexts	$\Gamma ::= \emptyset$	empty
	$\mid\ \Gamma, x : T$	term variable binding

Figure 4.5: Concrete syntax of kind and type contexts

A `let`-binding can contain any number of bindings. A binding itself can be a term binding, a type binding or a datatype binding. It is important to note that type bindings are opaque – i.e., they do not behave like a type alias in Haskell would. Type bindings serve the same purpose as term bindings: term bindings bind term-level sub-computations, and type bindings bind type-level sub-computations.

Final implementation. Terms and bindings are implemented as mutually defined inductive datatypes, seeing as terms can appear inside term bindings. The implementation follows easily from Figure 4.4.

```
Inductive term :=
  | Let     : Recursivity -> list binding -> term -> term
  | Var     : name -> term
  | TyAbs   : binderTyname -> kind -> term -> term
  | LamAbs  : binderName -> ty -> term -> term
  | Apply   : term -> term -> term
  | Constant : @some valueOf -> term
  | Builtin : DefaultFun -> term
  | TyInst  : term -> ty -> term
  | Error   : ty -> term
  | IWrap   : ty -> ty -> term -> term
  | Unwrap  : term -> term
with binding :=
  | TermBind : Strictness -> vdecl -> term -> binding
  | TypeBind : tvdecl -> ty -> binding
  | DatatypeBind : dtdecl -> binding.
```

4.2 Static semantics

In this section, we define kind and type rules that represent kind and type checking algorithms. We design the type rules such that they always check for a normalised type, which simplifies the process of reasoning about typing judgements in Coq. To achieve this, we define and implement concepts such as capture-avoiding type substitution and type normalisation.

4.2.1 Kind and type contexts

Figure 4.5 defines the concrete syntax of kind and type contexts, which are crucial components of kind and type rules. We implement both types of contexts using partial maps, which are defined in the Software Foundation books[29, 30]. Total maps are implemented as functions from `strings` to elements of a given type `A`. Instantiating the type `A` with `option A` gives us a partial map, since a lookup of an element that is not in the map can then return `None`.

```
Definition total_map (A : Type) := string -> A.
Definition partial_map (A : Type) := total_map (option A).
```

An empty *total* map should return a default value, or it would not be total. The main operations defined on total maps are lookups and updates. A lookup is just function application since total maps are modelled as functions. An update is simple to define.

$$\frac{\Delta(X) = K}{\Delta \vdash X :: K}\ \textbf{K-Var} \qquad \frac{\Delta \vdash T_1 :: * \quad \Delta \vdash T_2 :: *}{\Delta \vdash T_1 \rightarrow T_2 :: *}\ \textbf{K-Fun}$$

$$\frac{\Delta \vdash T :: K \quad \Delta \vdash F :: (K \rightarrow *) \rightarrow (K \rightarrow *)}{\Delta \vdash \texttt{ifix}\ F\ T :: *}\ \textbf{K-IFix} \qquad \frac{\Delta, X :: K \vdash T :: *}{\Delta \vdash (\forall X :: K.T) :: *}\ \textbf{K-Forall}$$

$$\frac{\mathbb{U}_i \text{ has kind } K}{\Delta \vdash \mathbb{U}_i :: K}\ \textbf{K-Builtin} \qquad \frac{\Delta, X :: K_1 \vdash T :: K_2}{\Delta \vdash (\lambda X :: K_1.T) :: K_1 \rightarrow K_2}\ \textbf{K-Lam}$$

$$\frac{\Delta \vdash T_1 :: K_1 \rightarrow K_2 \quad \Delta \vdash T_2 :: K_1}{\Delta \vdash T_1\ T_2 :: K_2}\ \textbf{K-App}$$

Figure 4.6: Kinding of types

```
Definition t_empty {A : Type} (v : A) : total_map A := (fun _ => v).
Definition t_update {A} (m : total_map A) (x : string) (v : A) :=
  fun x' => if x =? x' then v else m x'.
Notation "'_' '!->' v" := (t_empty v) (* hidden *).
Notation "x '!->' v ';' m" := (t_update m x v) (* hidden *).
```

An empty *partial* map will always return `None`. Lookups and updates defined on partial maps follow from their total map equivalents.

```
Definition empty {A : Type} : partial_map A := t_empty None.
Definition update {A : Type} (m : partial_map A) (x : string) (v : A) := (x !-> Some v; m).
Notation "x '|->' v ';' m" := (update m x v) (* hidden *).
Notation "x '|->' v" := (update empty x v) (* hidden *).
```

Modelling maps like functions allows us to reason about maps using the same machinery that we can use to reason about functions. For example, we can prove that a partial map where a variable is shadowed is propositionally equal to a partial map where the shadowed variable is removed.

```
Theorem update_shadow : forall (A : Type) (m : partial_map A) x v1 v2,
    (x |-> v2 ; x |-> v1 ; m) = (x |-> v2 ; m).
Proof. (* hidden *). Qed.
```

Finally, we implement multi-updates on partial maps because, amongst other reasons, `let`-bindings can update a context with any number of bindings. The multi-update is right-to-left: we apply the first update in the list `xts` last and vice versa for the last update in `xts`.

```
Fixpoint mupdate {X:Type} (m : partial_map X) (xts : list (string * X)) :=
  match xts with
  | nil => m
  | ((x, v) :: xts') => x |-> v ; (mupdate m xts')
  end.
```

What remains is to instantiate partial maps to obtain kind and type contexts.

```
Definition Delta : Type := partial_map Kind.
Definition Gamma : Type := partial_map Ty.
```

4.2.2 Kind rules

Figure 4.6 defines the set of inference rules that describe a kind checking algorithm. The rules are mostly standard, seeing as the type language is a small superset of a simply typed lambda calculus. We elaborate on a few interesting rules before we implement the kind rules.

Built-in types. We assume for the **K-Builtin** rule that we know the kind of every built-in type in the universe $\mathbb{U}$. For the default universe in the compiler, all built-in types have a base kind. However, we would like to facilitate the implementation of a modular approach in the future, so we implement a function that "looks up" the kind of a built-in type.

```
Definition lookupBuiltinKind (u : DefaultUni) : Kind :=
  match u with
  | DefaultUniInteger  => Kind_Base
  (* hidden *)
  end.
```

Fixpoint type. As mentioned before in subsection 4.1.4, the type-level fixpoint operator takes fixpoints only at kind $K \longrightarrow *$. This is reflected in the **K-IFix** rule: F should have kind $(K \longrightarrow *) \longrightarrow (K \longrightarrow *)$. The fixpoint of F is a type function with kind $K \longrightarrow *$, which can then be applied to an argument T of kind K to produce a final result of base kind. Altogether, we denote the application of a fixpoint of F to T as `ifix` F T.

Final implementation. The implementation of kinding is a simple translation of the inference rules to an inductive datatype.

```
Reserved Notation "Delta '|-*' T ':' K" (* hidden *).
Inductive has_kind : Delta -> Ty -> Kind -> Prop := (* hidden *)
where "Delta '|-*' T ':' K" := (has_kind Delta T K).
```

4.2.3 Type substitution

Because of the design of the type language, types can contain unfinished computations that can be *normalised* – i.e., types can contain beta redexes. To facilitate beta reduction, we implement a naive substitution function on types.

```
Fixpoint substituteT (X : tyname) (U T : Ty) : Ty := (* hidden *).
```

We also implement multi-substitutions on types since we need these in chapter 5. The order in which we substitute types in `ss` is left-to-right: we apply the substitution at the head of the list first.

```
Fixpoint msubstT (ss : list (tyname * Ty)) (T : Ty) : Ty := (* ... *).
```

A problem with `substituteT` is that it is not scope-preserving. Substitution can result in accidental *capturing* if the type `U` contains free variables that are bound in `T`. This is not a problem if we only substitute closed types, though this is not true for our static semantics. In the type rules that we define later, types that contain beta redexes can also have an arbitrary number of free type variables. For this reason, we set out to define a capture-avoiding type substitution.

Capture-avoiding type substitution is equivalent to type substitution except for the cases where we move a type substitution below a binder. As an example, assume that we compute $[U/X]\ (\forall Y :: K.T)$. Naturally, if $X = Y$, then type substitution has no effect. However, if they are not equal, then two cases can occur: either Y occurs freely in U or it does not. In the former case, we must rename Y and all occurrences of Y in T to a fresh type variable, and continue substitution in the renamed version of T. In the latter case, type substitution proceeds as normal and we compute $\forall Y :: K.\, [U/X]\, T$.

We need four ingredients to implement capture-avoiding type substitution: (i) a function that collects free variables in a type, (ii) a function that generates a fresh type variable, (iii) a function that performs renaming and (iv) a well-foundedness measure. Ingredient (i) is straightforward. We reduce the problem of generating a fresh type variable to generating a type variable that is not equal to X and does not occur freely in U or T, and we implement a simple solution to this reduced problem. We define renaming as a specialised case of type substitution. The last ingredient is needed because capture-avoiding substitution is not structurally recursive on the type in which we perform substitution. In the case where we avoid capturing, we continue substitution on a renamed version of the original term, which prevents Coq from determining whether the function terminates. We use the *size* of a type, defined as the number of constructors that the type contains, as a well-foundedness measure.

```
Fixpoint ftv (T : Ty) : list tyname := (* hidden *).
Definition fresh (X : tyname) (U T : Ty) : tyname := (* hidden *).
Definition rename (X Y : tyname) (T : Ty) := substituteT X (Ty_Var Y) T.
Fixpoint size (T : Ty) : nat := (* hidden *).
```

The proof of well-foundedness follows from the fact that renaming preserves the size of a type.

```
Lemma rename_preserves_size : forall T X Y,
    size T = size (rename X Y T).
Proof. (* hidden *). Qed.
```

$$\frac{}{(\lambda X :: K.T_1)\ T_2 \equiv [T_2/X]\ T_1}\ \textbf{Q-Beta} \qquad \frac{}{T \equiv T}\ \textbf{Q-Refl}$$

$$\frac{T \equiv S}{S \equiv T}\ \textbf{Q-Symm} \qquad \frac{S \equiv U \quad U \equiv T}{S \equiv T}\ \textbf{Q-Trans}$$

$$\frac{S_1 \equiv S_2 \quad T_1 \equiv T_2}{S_1 \to T_1 \equiv S_2 \to T_2}\ \textbf{Q-Fun} \qquad \frac{S \equiv T}{\forall X :: K.S \equiv \forall X :: K.T}\ \textbf{Q-Forall}$$

$$\frac{S \equiv T}{\lambda X :: K.S \equiv \lambda X :: K.T}\ \textbf{Q-Lam} \qquad \frac{S_1 \equiv S_2 \quad T_1 \equiv T_2}{S_1\ T_1 \equiv S_2\ T_2}\ \textbf{Q-App}$$

$$\frac{F_1 \equiv F_2 \quad T_1 \equiv T_2}{\texttt{ifix}\ F_1\ T_1 \equiv \texttt{ifix}\ F_2\ T_2}\ \textbf{Q-IFix}$$

Figure 4.7: Type equality

It remains to define the final capture-avoiding type substitution function and prove its well-foundedness.

```
Equations? substituteTCA (X : tyname) (U T : Ty) : Ty by wf (size T) :=
  (* hidden *)
  substituteTCA X U (Ty_Forall Y K T) =>
    if X =? Y
      then Ty_Forall Y K T
      else if existsb (eqb Y) (ftv U)
        then let Y' := fresh X U T in
             let T' := rename Y Y' T in
             Ty_Forall Y' K (substituteTCA X U T')
        else Ty_Forall Y K T ;
  (* hidden *).
Proof. (* proof using rename_preserves_size *). Qed.
```

Finally, note that the use of De Bruijn indices as variables would have prevented this problem entirely. However, since we chose to stick close to the compiler implementation, we have to deal with problems of this type.

4.2.4 Type normalisation

It is necessary to define normalisation of types because types can contain computations, and we would like our type rules to only type check with respect to normalised types. Moreover, it is easier to determine if types are equal if they are normal, and normalisation is necessary in the context of our approach to iso-recursive types. Figure 4.7 defines a notion of type equality, which is a first step on the road to a definition of type normalisation. Type equality will serve as a sanity check for the definition of type normalisation that we will formalise below. The definition of type equality can be split into three categories of rules: (i) congruence cases such as **Q-Fun** and **Q-Forall**, (ii) the beta reduction rule **Q-Beta**, and (iii) properties that hold in general for equalities: reflexivity, symmetry and transitivity. The implementation of type equality is a simple translation of the inference rules to an inductive datatype that represents a relation between types. Note that we use the capture-avoiding type substitution function.

```
Reserved Notation "T1 '=b' T2" (* hidden *).
Inductive EqT : Ty -> Ty -> Prop :=
  | Q_Beta : forall X K T1 T2 T1', substituteTCA X T2 T1 = T1' ->
      Ty_App (Ty_Lam X K T1) T2 =b T1'
  (* other rules hidden *)
where "T1 '=b' T2" := (EqT T1 T2).
```

Logically, the type normalisation relation should relate types to *normal* types. In summary, a type is normal if it contains no beta redexes. A normal type is not only beta redex free if it contains no function applications, since a type variable in a function position can also block beta reduction. We call a type a *neutral* type if it consists of a type variable applied to any number of normal types. Figure 4.8 describes the mutually defined sets of normal and neutral types, which we denote by *normal* and *neutral* in that order. We implement these sets as predicates on types using mutually defined inductive datatypes.

```
Inductive normal_Ty : Ty -> Prop := (* hidden *)
with neutral_Ty : Ty -> Prop := (* hidden *).
```

$$\frac{T \in normal}{\lambda X :: K.T \in normal}\ \textbf{NO-Lam} \qquad \frac{T \in neutral}{T \in normal}\ \textbf{NO-Neutral}$$

$$\frac{T_1 \in normal \quad T_2 \in normal}{T_1 \to T_2 \in normal}\ \textbf{NO-Fun} \qquad \frac{T \in normal}{\forall X :: K.T \in normal}\ \textbf{NO-Forall}$$

$$\frac{F \in normal \quad T \in normal}{\texttt{ifix}\ F\ T \in normal}\ \textbf{NO-IFix} \qquad \frac{}{\mathbb{U}_i \in normal}\ \textbf{NO-Builtin}$$

$$\frac{}{X \in neutral}\ \textbf{NE-Var} \qquad \frac{T_1 \in neutral \quad T_2 \in normal}{T_1\ T_2 \in neutral}\ \textbf{NE-App}$$

Figure 4.8: Normal and neutral types

$$\frac{T_1 \rightsquigarrow \lambda X :: K.T_1^n \quad T_2 \rightsquigarrow T_2^n \quad [T_2^n/X]\, T_1^n \rightsquigarrow T^n}{T_1\ T_2 \rightsquigarrow T^n}\ \textbf{N-Beta} \qquad \frac{T_1 \rightsquigarrow T_1^n \quad T_1^n \in neutral \quad T_2 \rightsquigarrow T_2^n}{T_1\ T_2 \rightsquigarrow T_1^n\ T_2^n}\ \textbf{N-App}$$

$$\frac{T_1 \rightsquigarrow T_1^n \quad T_2 \rightsquigarrow T_2^n}{T_1 \to T_2 \rightsquigarrow T_1^n \to T_2^n}\ \textbf{N-Fun} \qquad \frac{T \rightsquigarrow T^n}{\forall X :: K.T \rightsquigarrow \forall X :: K.T^n}\ \textbf{N-Forall}$$

$$\frac{T \rightsquigarrow T^n}{\lambda X :: K.T \rightsquigarrow \lambda X :: K.T^n}\ \textbf{N-Lam} \qquad \frac{}{X \rightsquigarrow X}\ \textbf{N-Var}$$

$$\frac{F \rightsquigarrow F^n \quad T \rightsquigarrow T^n}{\texttt{ifix}\ F\ T \rightsquigarrow \texttt{ifix}\ F^n\ T^n}\ \textbf{N-IFix} \qquad \frac{}{\mathbb{U}_i \rightsquigarrow \mathbb{U}_i}\ \textbf{N-Builtin}$$

Figure 4.9: Type normalisation

With a notion of normal and neutral types defined, we can now define a normalisation algorithm. Figure 4.9 defines a set of inference rules that describes a type normalisation algorithm. We denote by $T \rightsquigarrow T^n$ that a type T normalises to T^n, where the superscript n is just notation for normality and does not represent, for example, a variable. Just by visual inspection, we can see that the normalisation algorithm is similar to the definition of type equality.

Though we would like to implement type normalisation as a function, it is a rather hard problem to convince Coq that the function terminates. Proving well-foundedness of the function is equivalent to proving that the type language is strongly normalising, which requires a non-trivial proof using advanced proof techniques such as logical relations. Instead, we simply bypass this problem by implementing normalisation as a relation between types.

```
Inductive normalise : Ty -> Ty -> Prop := (* hidden *).
```

A first sanity check for the normalisation relation is to prove that it produces normal types. We can prove this using a one-liner.

```
Lemma normalise_to_normal : forall T Tn, normalise T Tn -> normal_Ty Tn.
Proof. induction 1; eauto. Qed.
```

We also prove that the normalisation relation is deterministic, and we prove that the normalisation relation is stable – i.e., normalisation of normal and neutral types returns exactly the input type.

```
Lemma normalisation__deterministic : forall T Tn T'n,
    normalise T Tn -> normalise T T'n -> Tn = T'n.
Proof. (* hidden *). Qed.

Theorem normalisation__stable :
  (forall T, normal_Ty T -> (forall Tn, normalise T Tn -> T = Tn)) /\
  (forall T, neutral_Ty T -> (forall Tn, normalise T Tn -> T = Tn)).
Proof. (* hidden *). Qed.

Lemma normalisation__stable' :
  (forall Tn, normal_Ty Tn -> normalise Tn Tn) /\ (forall Tn, neutral_Ty Tn -> normalise Tn Tn).
Proof. (* hidden *). Qed.
```

Lastly, we prove that the normalisation relation is sound with respect to our definition of type equivalence. Whether the normalisation relation is complete with respect to type equivalence remains unanswered.

Throughout this figure, when d or c is an argument, then:

$$
\begin{aligned}
d &= \mathtt{data}\ (X :: K)\ (\overline{Y :: J}) = \overline{c}\ \mathtt{with}\ x \\
c &= y\ (T) \\
T &= \overline{S} \rightarrow U \text{ and } U \text{ is the return type}
\end{aligned}
$$

Auxiliary functions

$$
\begin{aligned}
\mathrm{branchTy}(c, R) &= \overline{S \rightarrow} R \\
\mathrm{dataTy}(d) &= \lambda(\overline{Y :: J}).\forall R :: *.(\overline{\mathrm{branchTy}(c, \underline{R})}) \rightarrow R \\
\mathrm{constrLastTy}(d) &= X\ \overline{Y} \\
\mathrm{constrTy}(d, c) &= \forall(\overline{Y :: J}).\mathrm{branchTy}(c, \mathrm{constrLastTy}(d)) \\
\mathrm{matchTy}(d) &= \forall(\overline{Y :: J}).\mathrm{constrLastTy}(d) \rightarrow (\mathrm{dataTy}(d)\ \overline{Y})
\end{aligned}
$$

Binder functions

$$
\begin{aligned}
\mathrm{constrBind}(d, c) &= y : \mathrm{constrTy}(d, c) \\
\mathrm{constrBinds}(d) &= \overline{\mathrm{constrBind}(\underline{d}, c)} \\
\mathrm{matchBind}(d) &= x : \mathrm{matchTy}(d) \\
\mathrm{binds}_\Gamma(x : T = t) &= x : T \\
\mathrm{binds}_\Gamma(d) &= \mathrm{constrBinds}(d), \mathrm{matchBind}(d) \\
\mathrm{binds}_\Delta(X :: K = T) &= X :: K \\
\mathrm{binds}_\Delta(d) &= X :: K
\end{aligned}
$$

Figure 4.10: Auxiliary definitions

```
Theorem normalisation__sound : forall T Tn,
   normalise T Tn -> T =b Tn.
Proof. (* hidden *). Qed.

Lemma normalisation__complete : forall S T Sn,
   S =b T -> normalise S Sn -> normalise T Sn.
Proof. Abort.
```

4.2.5 Auxiliary definitions

We need a number of auxiliary definitions that work on bindings, declarations and constructors before we can finally define the type rules. These definitions are necessary because we must collect all term variable and type variable bindings that term bindings, type bindings and datatype bindings introduce. For example, each constructor of a datatype introduces a new term variable binding. Figure 4.10 describes the auxiliary definitions. A note on terminology: if we say that U is the result type of T, then we mean to say that $T = \overline{S} \rightarrow U$ and that U is not an arrow type. In other words, fully applying a function of type T to a list of arguments with subsequent types $\overline{S}$ will yield a non-arrow result type U.

```
Definition binds_Delta (b : Binding) : list (tyname * Kind) := (* hidden *).
Definition binds_Gamma (b : Binding) : list (name * Ty) := (* hidden *).
```

4.2.6 Well-formedness of constructors and bindings

As a last component before we finally define type rules, we define *well-formedness* of constructors and bindings by defining a set of inference rules that represent a well-formedness checking algorithm. Well-formedness rules are similar to type rules, but differ in the sense that they do not check for a type. Instead, we only check if the constructors and bindings are well-formed with respect to their type annotations. Figure 4.11 defines the set of inference rules that describe the well-formedness checking algorithm. We delay the implementation of well-formedness because it is mutually defined with the type rules.

$$\frac{c = x\ (\overline{S \longrightarrow U}) \quad U \text{ is result type} \quad \overline{\Delta \vdash S :: *}}{\Delta \vdash_{\text{ok\_c}} c : U}\ \textbf{W-Con} \qquad \frac{\Delta \vdash T :: * \quad T \rightsquigarrow T^n \quad \Delta; \Gamma \vdash t : T^n}{\Delta; \Gamma \vdash_{\text{ok\_b}} x : T\ (str) = t}\ \textbf{W-Term}$$

$$\frac{\Delta \vdash T :: K}{\Delta; \Gamma \vdash_{\text{ok\_b}} X :: K = T}\ \textbf{W-Type} \qquad \frac{d = \texttt{data}\ (X :: K)\ (\overline{Y :: J}) = \overline{c}\ \texttt{with}\ x \quad \Delta' = \Delta, \overline{Y :: J} \quad \overline{\Delta' \vdash_{\text{ok\_c}} c : \text{constrLastTy}(\underline{d})}}{\Delta; \Gamma \vdash_{\text{ok\_b}} d}\ \textbf{W-Data}$$

$$\frac{\overline{b} = b, \overline{b'} \quad \Delta; \Gamma \vdash_{\text{ok\_b}} b \quad \text{binds}_\Gamma(b) \rightsquigarrow \text{binds}_\Gamma(b)^n \quad (\Delta, \text{binds}_\Delta(b)); (\Gamma, \text{binds}_\Gamma(b)^n) \vdash_{\text{oks\_nr}} \overline{b'}}{\Delta; \Gamma \vdash_{\text{oks\_nr}} \overline{b}}\ \textbf{W-BindingsNonRec}$$

$$\frac{\overline{b} = b, \overline{b'} \quad \Delta; \Gamma \vdash_{\text{ok\_b}} b \quad \Delta; \Gamma \vdash_{\text{oks\_r}} \overline{b'}}{\Delta; \Gamma \vdash_{\text{oks\_r}} \overline{b}}\ \textbf{W-BindingsRec}$$

Figure 4.11: Well-formedness of constructors and bindings

Well-formedness of constructors requires that the *return* type of the constructor is a given type U, and it requires that the type *arguments* of the constructor all have a base kind. The well-formedness of term bindings follows from well-typedness of the bound term. As such, well-formedness is mutually defined with the typing judgements that we define later. Well-formedness of type bindings follows from well-kindedness of the bound type. Well-formedness of datatype bindings follows from well-formedness of its constructors with respect to the expected return type of constructors: the datatype name applied to the type parameters of the datatype. For example, a constructor of the `Either` datatype with type parameters A and B should always return something of type `Either` A B. Lastly, we also defined well-formedness of *sequences* of bindings. In the case of non-recursive bindings, the term and type variable bindings of the binding at the *head* of the sequence come into scope when we check well-formedness of the *tail* of the sequence. In the case for recursive bindings, the term and type variable bindings of each binding in the sequence should already be in scope, or else the bindings would not be (mutually) recursive, so we do not bring anything new into scope when checking well-formedness of the tail of the sequence. We omit rules for empty sequences as they are trivially well-formed.

4.2.7 Type rules

Figure 4.12 defines the set of inference rules that describe a type checking algorithm. We ensure that we can only type check for normal types. In addition, we take care to only update type contexts with normalised types. We elaborate on the more interesting rules now.

Variables. We remarked just before this paragraph that we update type contexts only with normalised types. However, we do not want to carry around witnesses of the invariant that states that type contexts only contain normalised types. Because of this, we normalise a looked-up type in the **T-Var** rule as a precaution. As such, we simplify proof engineering, and it is not a problem since type normalisation is stable.

Recursive types. We can unfold a type `ifix` F T to $(F\ (\lambda X :: K.\texttt{ifix}\ F\ X))\ T$, given that T has kind K. The reverse is true for the folding of types.

Built-in functions. In the **T-Builtin** rule, we assume that we can determine whether a built-in function has some type T. It makes sense that we can always determine this, since a built-in function with unknown type is not very useful. Therefore, we can require that any collection of built-in functions must be accompanied with a function that "looks up" the corresponding type of a built-in function. We can easily implement such a function for the default built-in function collection.

```
Definition lookupBuiltinTy (f : DefaultFun) : Ty := (* hidden *).
```

Constants. We defined machinery such as the `some`, `typeIn` and `valueOf` datatypes to represent *some* built-in types and *some* values of built-in types. In particular, this machinery prevents us from writing values that are not of the built-in type that a constant is annotated with. For example, a constant such as `Constant (Some (ValueOf DefaultUniBool true))` can only be of the type `Ty_Builtin (Some (TypeIn DefaultUniBool))`. So, if we consider the **T-Constant** rule, the premise "c has type $\mathbb{U}_i$" is trivially true in the Coq implementation.

$$\frac{\Gamma(x) = T \qquad T \rightsquigarrow T^n}{\Delta;\Gamma \vdash x : T^n}\ \textbf{T-Var} \qquad \frac{\Delta \vdash T_1 :: * \qquad T_1 \rightsquigarrow T_1^n \qquad \Delta;(\Gamma, x : T_1^n) \vdash t : T_2^n}{\Delta;\Gamma \vdash (\lambda x : T_1.t) : T_1^n \rightarrow T_2^n}\ \textbf{T-LamAbs}$$

$$\frac{\Delta;\Gamma \vdash t_1 : T_1^n \rightarrow T_2^n \qquad \Delta;\Gamma \vdash t_2 : T_1^n}{\Delta;\Gamma \vdash t_1\ t_2 : T_2^n}\ \textbf{T-Apply} \qquad \frac{(\Delta, X :: K);\Gamma \vdash t : T^n}{\Delta;\Gamma \vdash (\Lambda X :: K.t) : (\forall X :: K.T^n)}\ \textbf{T-TyAbs}$$

$$\frac{\Delta;\Gamma \vdash t_1 : (\forall X :: K_2.T_1^n) \qquad \Delta \vdash T_2 :: K_2 \qquad T_2 \rightsquigarrow T_2^n \qquad \left[T_2^n/X\right] T_1^n \rightsquigarrow T^n}{\Delta;\Gamma \vdash t_1\ \{T_2\} : T^n}\ \textbf{T-TyInst}$$

$$\frac{\begin{array}{ccc} \Delta \vdash T :: K & \Delta \vdash F :: (K \rightarrow *) \rightarrow (K \rightarrow *) & (F^n\ (\lambda X :: K.\mathtt{ifix}\ F^n\ X))\ T^n \rightsquigarrow T_0^n \\ T \rightsquigarrow T^n & F \rightsquigarrow F^n & \Delta;\Gamma \vdash M : T_0^n \end{array}}{\Delta;\Gamma \vdash \mathtt{IWrap}\ F\ T\ M : \mathtt{ifix}\ F^n\ T^n}\ \textbf{T-IWrap}$$

$$\frac{\Delta;\Gamma \vdash M : \mathtt{ifix}\ F^n\ T^n \qquad \Delta \vdash T^n :: K \qquad (F^n\ (\lambda X :: K.\mathtt{ifix}\ F^n\ X))\ T^n \rightsquigarrow T_0^n}{\Delta;\Gamma \vdash \mathtt{Unwrap}\ M : T_0^n}\ \textbf{T-Unwrap}$$

$$\frac{c \text{ has type } \mathbb{U}_i}{\Delta;\Gamma \vdash \mathtt{Constant}\ \mathbb{U}_i\ c : \mathbb{U}_i}\ \textbf{T-Constant} \qquad \frac{\begin{array}{c} \mathbb{F}_i \text{ has type } T \\ T \rightsquigarrow T^n \end{array}}{\Delta;\Gamma \vdash \mathbb{F}_i : T^n}\ \textbf{T-Builtin} \qquad \frac{\begin{array}{c} \Delta \vdash T :: * \\ T \rightsquigarrow T^n \end{array}}{\Delta;\Gamma \vdash \mathtt{Error}\ S : T^n}\ \textbf{T-Error}$$

$$\frac{\begin{array}{cccc} & \overline{\text{binds}_\Gamma(b) \rightsquigarrow \text{binds}_\Gamma(b)^n} & \Delta;\Gamma \vdash_{\text{oks\_nr}} \overline{b} & \\ \Delta' = \Delta, \overline{\text{binds}_\Delta(b)} & \Gamma' = \Gamma, \overline{\text{binds}_\Gamma(b)^n} & \Delta';\Gamma' \vdash t : T^n & \Delta \vdash T^n :: * \end{array}}{\Delta;\Gamma \vdash \mathtt{let}\ \mathtt{(NonRec)}\ \overline{b}\ \mathtt{in}\ t : T^n}\ \textbf{T-Let}$$

$$\frac{\begin{array}{cccc} & \overline{\text{binds}_\Gamma(b) \rightsquigarrow \text{binds}_\Gamma(b)^n} & \Delta';\Gamma' \vdash_{\text{oks\_r}} \overline{b} & \\ \Delta' = \Delta, \overline{\text{binds}_\Delta(b)} & \Gamma' = \Gamma, \overline{\text{binds}_\Gamma(b)^n} & \Delta';\Gamma' \vdash t : T^n & \Delta \vdash T^n :: * \end{array}}{\Delta;\Gamma \vdash \mathtt{let}\ \mathtt{(Rec)}\ \overline{b}\ \mathtt{in}\ t : T^n}\ \textbf{T-LetRec}$$

Figure 4.12: Typing of terms

Error. Recall that errors are annotated with a type for the sake of principal types. Errors can interrupt evaluation, after which they are propagated through parent constructors. It should be the case that the resulting error value (after propagation) has the same type as the original term. For example, a term that computes something of an integer type should not have a boolean type after evaluating to an error. The current type annotations in terms are not sufficient to manipulate the type annotation of an error such that it has the correct type after error propagation. For now, we remark that we ignore the type annotation such that an error can have any normalised type. We suspect that this issue can be solved by augmenting the type annotations of terms.

`let`-bindings. To type check non-recursive `let`-bindings, we first check whether the binding sequence is well-formed without updating the kind and type contexts. The **W-BindsNonRec** rule makes sure to bring term and type variable bindings into scope. We then make sure that the body of the `let`-binding is well-typed. To do this, we must update the kind context with the type variable bindings of the binding sequence, and we must update the type context with the normalised term variable bindings of the binding sequence. Type checking recursive `let`-bindings is similar, but only differs in the sense that we check well-formedness of the bindings sequence with the updated kind and type contexts. This is necessary because the bindings in the sequence are possibly mutually recursive, so all term and type variable bindings should be in scope before we check well-formedness.

Lastly, we explain the last premise of both the **T-Let** and **T-LetRec** rules. In Peyton Jones et al.[15], the type rules explicitly prevent escaping of the inner type. In other words, the term and type variable bindings of the binding sequence are not in scope when type checking the entire `let`-binding. However, the compiler implementation allows escaping on one condition: the `let`-binding that is being type checked is a top-level `let`-binding. This is allowed since compiling the `let`-binding to Plutus Core would result in a well-behaved term. This is problematic for multiple reasons. First, consider the compilation scheme from Peyton Jones et al.[15] that compiles a `let`-binding with a single type binding:

$$\mathbb{C}_{\text{type}}(\texttt{let (NonRec) } (X :: K = T) \texttt{ in } t) = (\Lambda X :: K.t)\ \{T\} \tag{4.1}$$

If the `let`-binding before compilation has an escaped type, then the type of the compiled term has changed! In addition, escaping makes proofs, such as the fact that "substitution preserves typing" and that "evaluation preserves typing", harder to complete. As such, we opted to disallow the escaping of types altogether. This means that our static semantics implementation is only sound with respect to the compiler implementation of the static semantics. Changes to the compiler or a different type rule may resolve these issues, but we leave it to future work to flesh out the solution.

Final implementation. Finally, we introduce the implementation of the type rules and the well-formedness rules. The rules in Figure 4.11 and 4.12 are translated to mutually defined inductive datatypes.

```
Reserved Notation "Delta ',,' Gamma '|-+' t ':' T" (* hidden *).
Reserved Notation "Delta '|-ok_c' c ':' T" (* hidden *).
Reserved Notation "Delta ',,' Gamma '|-oks_nr' bs" (* hidden *).
Reserved Notation "Delta ',,' Gamma '|-oks_r' bs" (* hidden *).
Reserved Notation "Delta ',,' Gamma '|-ok_b' b" (* hidden *).

Inductive has_type : Delta -> Gamma -> Term -> Ty -> Prop := (* hidden *)
with constructor_well_formed : Delta -> constructor -> Ty -> Prop := (* hidden *)
with bindings_well_formed_nonrec : Delta -> Gamma -> list Binding -> Prop := (* hidden *)
with bindings_well_formed_rec : Delta -> Gamma -> list Binding -> Prop := (* hidden *)
with binding_well_formed : Delta -> Gamma -> Binding -> Prop := (* hidden *)

where "Delta ',,' Gamma '|-+' t ':' T" := (has_type Delta Gamma t T)
and  "Delta '|-ok_c' c ':' T" := (constructor_well_formed Delta c T)
and "Delta ',,' Gamma '|-oks_nr' bs" := (bindings_well_formed_nonrec Delta Gamma bs)
and "Delta ',,' Gamma '|-oks_r' bs" := (bindings_well_formed_rec Delta Gamma bs)
and "Delta ',,' Gamma '|-ok_b' b" := (binding_well_formed Delta Gamma b).
```

We implemented type normalisation with the goal of designing the type checking algorithm such that it only checks for normal types. We can prove that the type checking algorithm only checks for normal types with a short proof.

```
Lemma has_type__normal : forall Delta Gamma flag t T,
    Delta ,, Gamma ;; flag |-+ t : T ->
    normal_Ty T.
Proof. (* proof with normalise_to_normal *). Qed.
```

$$\frac{}{\lambda x : T.t \in \mathit{value}}\ \textbf{V-LamAbs} \qquad \frac{}{\Lambda X :: K.t \in \mathit{value}}\ \textbf{V-TyAbs}$$

$$\frac{v \in \mathit{value} \qquad \neg \mathit{isError}(v)}{\texttt{IWrap}\ F\ T\ v \in \mathit{value}}\ \textbf{V-IWrap} \qquad \frac{}{\texttt{Constant}\ \mathbb{U}_i\ c \in \mathit{value}}\ \textbf{V-Constant}$$

$$\frac{}{\texttt{Error}\ T \in \mathit{value}}\ \textbf{V-Error} \qquad \frac{\langle 0, nv\rangle \in \mathit{neutral}}{nv \in \mathit{value}}\ \textbf{V-Neutral}$$

$$\frac{n < \mathit{arity}(\mathbb{F}_i)}{\langle n, \mathbb{F}_i\rangle \in \mathit{neutral}}\ \textbf{NV-Builtin} \qquad \frac{v \in \mathit{value} \qquad \neg \mathit{isError}(v) \qquad \langle \texttt{succ}\ n, nv\rangle \in \mathit{neutral}}{\langle n, nv\ v\rangle \in \mathit{neutral}}\ \textbf{NV-Apply}$$

$$\frac{\langle \texttt{succ}\ n, nv\rangle \in \mathit{neutral}}{\langle n, nv\ \{T\}\rangle \in \mathit{neutral}}\ \textbf{NV-TyInst}$$

Figure 4.13: Values and neutral terms

4.3 Dynamic Semantics

In this section, we define a big-step operational semantics for the term-level language in a Call-By-Value style using substitution. In the process, we define and implement additional concepts such as values, term substitution and type annotation substitution. It is important to note that the term-level language is not strongly normalising, which is a consequence of iso-recursive types[7]. For the semantics of default built-in functions, we closely follow the meaning that the Plutus Tx compiler assigns to these functions.

4.3.1 Values

We define the possible result values that can be produced by evaluating a term. Values can be suspended computations, such as lambda and type abstractions, terms that must further be type-unfolded, such as `IWrap` terms, or base terms such as constants and errors. In addition, similarly to how normal types include neutral types, values include neutral *terms*. We need neutral terms to properly evaluate built-in functions, because we need to fully apply a built-in function to compute its result. A neutral term is therefore a built-in function that is partially applied to a number of (non-error) values and types.

Figure 4.13 describes the mutually defined sets of values and neutral terms, which we denote by *value* and *neutral* respectively. Most of the rules are straightforward, except for rules that involve built-in functions. To determine whether a term is neutral, we use a peano natural number (with a zero-element 0 and a successor function `succ`) to count the arguments that a built-in function is applied to. We use the function *arity* to compute the arity of a built-in function. The previous two components allow us to define neutral terms and the **V-Neutral** rule. We can implement values and neutral terms as predicates using mutually defined inductive datatypes, together with a function that computes the arity of a default built-in function, and a predicate that defines whether a term is an `Error`.

```
Definition arity (df : DefaultFun) : nat := (* hidden *).
Inductive is_error : Term -> Prop := (* hidden *)

Inductive value : Term -> Prop := (* hidden *)
with neutral_value : nat -> Term -> Prop := (* hidden *).
```

4.3.2 Term and type annotation substitution

Our operational semantics uses two types of substitutions: substitution on terms and substitution on type annotations. We need term substitution primarily to define the semantics of function application. We need type annotation substitution primarily to define the semantics of type instantiation. Note that we make no effort to avoid accidental capturing like we did for our implementation of capture-avoiding type substitution since we will only use these substitution functions to substitute closed terms and types.

```
Fixpoint substitute (x : name) (s : Term) (t : Term) {struct t} : Term :=
  (* implementation hidden *)
with substitute_binding (x : name) (s : Term) (b : Binding) {struct b} : Binding :=
  (* implementation hidden *).
```

$$\frac{}{\lambda x : T.t \Downarrow^{0} \lambda x : T.t}\ \textbf{E-LamAbs} \qquad \frac{t_1 \Downarrow^{j_1} \lambda x : T.t_0 \qquad \begin{array}{c} t_2 \Downarrow^{j_2} v_2 \\ \neg isError(v_2) \end{array} \qquad [v_2/x]\ t_0 \Downarrow^{j_0} v_0}{t_1\ t_2 \Downarrow^{j} v_0 \text{ where } j = j_1 + j_2 + 1 + j_0}\ \textbf{E-Apply}$$

$$\frac{}{\Lambda X :: K.t \Downarrow^{0} \Lambda X :: K.t}\ \textbf{E-TyAbs} \qquad \frac{t_1 \Downarrow^{j_1} \Lambda X :: K.t_0 \qquad [T_2/X]\ t_0 \Downarrow^{j_0} v_0}{t_1\ \{T_2\} \Downarrow^{j} v_0 \text{ where } j = j_1 + 1 + j_0}\ \textbf{E-TyInst}$$

$$\frac{t_0 \Downarrow^{j_0} v_0 \qquad \neg isError(v_0)}{\texttt{IWrap}\ F\ T\ t_0 \Downarrow^{j_0} \texttt{IWrap}\ F\ T\ v_0}\ \textbf{E-IWrap} \qquad \frac{t_0 \Downarrow^{j_0} \texttt{IWrap}\ F\ T\ v_0}{\texttt{Unwrap}\ t_0 \Downarrow^{j} v_0 \text{ where } j = j_0 + 1}\ \textbf{E-Unwrap}$$

$$\frac{}{\texttt{Constant}\ \mathbb{U}_i\ c \Downarrow^{0} \texttt{Constant}\ \mathbb{U}_i\ c}\ \textbf{E-Constant}$$

Figure 4.14: Big-step operational semantics (basic rules)

For type annotation substitution, we reuse the `substituteT` function to perform the actual substitution of a type in some type annotation. Since we aim to use the type annotation substitution function only to substitute closed types, we do not have to use the capture-avoiding type substitution function.

```
Fixpoint substituteA (X : tyname) (U : Ty) (t : Term) {struct t} : Term :=
  (* implementation hidden *)
with substituteA_binding (X : tyname) (U : Ty) (b : Binding) {struct b} : Binding :=
  (* implementation hidden *).
```

In anticipation of chapter 5, we also implement multi-substitutions on terms and type annotations. The order in which we substitute terms and types that are in `ss` is left-to-right: we apply the substitution at the head of the list first.

```
Fixpoint msubst_term (ss : list (name * Term)) (t : Term) : Term :=
  (* implementation hidden *).
Fixpoint msubstA_term (ss : list (tyname * Ty).) (t : Term) : Term :=
  (* implementation hidden *).
```

In this subsection, we have omitted some details. In order to implement the two types of substitutions, we have also defined, amongst others, term substitution on sequences of bindings and type annotation substitution on constructors. Moreover, we have implemented multi-substitution functions for most of the substitution functions that we have implemented but not necessarily listed here. We will not explicitly use these additional substitution functions throughout the remainder of this thesis, but it is good to be aware of their existence.

4.3.3 Evaluation relation

We define the big-step operational semantics as an evaluation relation. We make sure that the operational semantics also counts the number of evaluation steps that are used. This is done in anticipation of chapter 5, where we employ step-indexing to formulate and prove dynamic semantics preservation. We denote by $t \Downarrow^{j} v$ that a term t evaluates in j steps to a value v. Naturally, we construct the operational semantics such that v is a value.

We will discuss the evaluation relation in four parts, which we define by giving inference rules in four separate figures. In all cases, we can put the inference rules into two categories: rules that describe how values evaluate to themselves, and (beta) reduction/computation rules. In sequence, we discuss (i) the basic evaluation rules, (ii) evaluation rules for `let`-bindings, (iii) evaluation rules for built-in functions, (iv) evaluation rules for errors, and (v) the final implementation of the evaluation relation. Recall that we do not give evaluation rules for datatype bindings, we ignore strictness flags for term bindings, we do not yet model laziness where it is sometimes needed, and the semantics of some built-in functions are not yet correctly modelled.

(i) Basic rules

Figure 4.14 describes the "basic" rules of the evaluation relation. That is, the figure describes the rules that do not involve `let`-bindings, built-in functions and errors. We do however make sure that sub-computations of computation rules do not result in errors, as an error would interrupt normal evaluation. We will give a short description of the rules.

The **E-LamAbs** and **E-TyAbs** rules describe that suspended computations evaluate to themselves. The **E-Apply** and **E-TyInst** rules describe how function and type instantiation/application can continue a suspended computation. Note that we use the same substitution notation for both term and type annotation substitution. We presume that it is clear from the context which type of substitution is performed. The **E-IWrap** rule makes sure that the wrapped term

$$\frac{\texttt{let (NonRec) } \overline{b} \texttt{ in } t \Downarrow^{j}_{nr} v}{\texttt{let (NonRec) } \overline{b} \texttt{ in } t \Downarrow^{j} v}\ \textbf{E-Let} \qquad \frac{\overline{b} \vdash \texttt{let (Rec) } \overline{b} \texttt{ in } t \Downarrow^{j}_{r} v}{\texttt{let (Rec) } \overline{b} \texttt{ in } t \Downarrow^{j} v}\ \textbf{E-LetRec}$$

$$\frac{t_0 \Downarrow^{j_0} v_0}{\texttt{let (NonRec) } \varepsilon \texttt{ in } t_0 \Downarrow^{j}_{nr} v_0 \text{ where } j = j_0 + 1}\ \textbf{E-Let-Nil}$$

$$\frac{t_1 \Downarrow^{j_1} v_1 \quad \neg isError(v_1) \quad [v_1/x]\,(\texttt{let (NonRec) } \overline{b} \texttt{ in } t_0) \Downarrow^{j_2}_{nr} v_2}{\texttt{let (NonRec) } ((x : T\ (str) = t_1), \overline{b}) \texttt{ in } t_0 \Downarrow^{j}_{nr} v_2 \text{ where } j = j_1 + 1 + j_2}\ \textbf{E-Let-TermBind}$$

$$\frac{[T/X]\,(\texttt{let (NonRec) } \overline{b} \texttt{ in } t_0) \Downarrow^{j_1}_{nr} v_1}{\texttt{let (NonRec) } ((X :: K = T), \overline{b}) \texttt{ in } t_0 \Downarrow^{j}_{nr} v_1 \text{ where } j = j_1 + 1}\ \textbf{E-Let-TypeBind}$$

$$\frac{t_0 \Downarrow^{j_0} v_0}{\overline{b_0} \vdash \texttt{let (Rec) } \varepsilon \texttt{ in } t_0 \Downarrow^{j}_{r} v_0 \text{ where } j = j_0 + 1}\ \textbf{E-LetRec-Nil}$$

$$\frac{\overline{b_0} \vdash \left[(\texttt{let (Rec) } \overline{b_0} \texttt{ in } t_1)/x\right] (\texttt{let (Rec) } \overline{b} \texttt{ in } t_0) \Downarrow^{j_1}_{r} v_1}{\overline{b_0} \vdash \texttt{let (Rec) } ((x : T\ (str) = t_1), \overline{b}) \texttt{ in } t_0 \Downarrow^{j}_{r} v_1 \text{ where } j = j_1 + 1}\ \textbf{E-LetRec-TermBind}$$

Figure 4.15: Big-step operational semantics (`let`-bindings)

is a value, such that the **E-Unwrap** rule can simply cancel out the inner `IWrap` constructor. Constants also evaluate to themselves, as is described by the **E-Constant** rule.

(ii) `let`-bindings

Figure 4.15 describes the evaluation rules for `let`-bindings. The operational semantics of `let`-bindings must be defined separately for non-recursive and recursive `let`-bindings. In particular, we also use two separate evaluation relations: one for the semantics of non-recursive `let`-bindings, and one for the semantics of recursive `let`-bindings. As we will see below, this is actually only necessary for the semantics of recursive `let`-bindings, since we must carry around an environment of bindings that are in scope. In contrast, non-recursive `let`-bindings do not require such an environment. We consider it useful nonetheless to present the semantics of both types of `let`-bindings in a similar style.

We denote by $\texttt{let (NonRec) } \overline{b} \texttt{ in } t_0 \Downarrow^{j}_{nr} v$ that the non-recursive `let`-binding evaluates to a value v in j steps. We denote by $\overline{b_0} \vdash \texttt{let (Rec) } \overline{b} \texttt{ in } t_0 \Downarrow^{j}_{r} v$ that the recursive `let`-binding evaluates to a value v in j steps given that the sequence of bindings $\overline{b_0}$ is in scope. The **E-Let** and **E-LetRec** rules make sure to embed the evaluation relations for `let`-bindings in the main evaluation relation. The **E-Let-Nil** and **E-LetRec-Nil** rules evaluate the inner term t_0 to a value v_0 in case the binding sequence is empty, which we denote by ε.

The **E-Let-TermBind** and **E-Let-TypeBind** rules are similar to the **E-Apply** and **E-TyInst** rules from Figure 4.14. We make sure to substitute terms and types in both the tail of the binding sequence and the inner term, after which we continue to evaluate the result of this substitution.

The **E-LetRec-TermBind** rule wraps the bound term in a recursive `let`-binding containing the binding sequence $\overline{b_0}$, which we then substitute in both the tail of the binding sequence and the inner term. We wrap the bound term in the recursive `let`-binding because the bound term might be recursive with respect to any number of terms that are in scope in $\overline{b_0}$, so we must make sure that the $\overline{b_0}$ *stays* in scope. Then, we continue to evaluate the result of this substitution. Lastly, note that we do not give an **E-LetRec-TypeBind** rule because recursive type bindings are not supported by the Plutus Tx compiler.

(iii) Built-in functions.

Figure 4.16 describes the evaluation rules for built-in functions. The **E-NeutralBuiltin**, **E-NeutralApply** and **E-NeutralTyInst** rules describe how neutral terms evaluate to themselves as they are suspended computations. The **E-NeutralApplyFull** and **E-NeutralTyInstFull** rules describe how fully applied built-in functions evaluate to some computed value. The implementation of $isFullyApplied()$ is similar to the implementation of neutral terms: a fully applied built-in function is a neutral term applied to a non-error value or instantiated with a type, such that they are together no longer neutral. The $compute()$ function computes the result of a fully applied built-in function, and we implement it by a simple pattern match on the argument term.

$$\frac{}{\mathbb{F}_i \Downarrow^0 \mathbb{F}_i}\ \textbf{E-NeutralBuiltin} \qquad \frac{\langle 0, nv\ v\rangle \in neutral}{nv\ v \Downarrow^0 nv\ v}\ \textbf{E-NeutralApply} \qquad \frac{\langle 0, nv\ \{T\}\rangle \in neutral}{nv\ \{T\} \Downarrow^0 nv\ \{T\}}\ \textbf{E-NeutralTyInst}$$

$$\frac{\neg(\langle 0, t_1\ t_2\rangle \in neutral) \quad \begin{array}{c} t_1 \Downarrow^{j_1} nv_1 \\ \langle 0, nv_1\rangle \in neutral \end{array} \quad \begin{array}{c} t_2 \Downarrow^{j_2} v_2 \\ \neg isError(v_2) \end{array} \quad nv_1\ v_2 \Downarrow^{j_0} v_0}{t_1\ t_2 \Downarrow^{j} v_0 \text{ where } j = j_1 + j_2 + 1 + j_0}\ \textbf{E-NeutralApplyPartial}$$

$$\frac{\neg(\langle 0, t_1\ \{T\}\rangle \in neutral) \quad \begin{array}{c} t_1 \Downarrow^{j_1} nv_1 \\ \langle 0, nv_1\rangle \in neutral \end{array} \quad nv_1\ \{T\} \Downarrow^{j_0} v_0}{t_1\ \{T\} \Downarrow^{j} v_0 \text{ where } j = j_1 + 1 + j_0}\ \textbf{E-NeutralTyInstPartial}$$

$$\frac{isFullyApplied(nv_1\ v_2) \quad compute(nv_1\ v_2) = v}{nv_1\ v_2 \Downarrow^1 v}\ \textbf{E-NeutralApplyFull}$$

$$\frac{isFullyApplied(nv_1\ \{T\}) \quad compute(nv_1\ \{T\}) = v}{nv_1\ \{T\} \Downarrow^1 v}\ \textbf{E-NeutralTyInstFull}$$

Figure 4.16: Big-step operational semantics (built-in functions)

```
Inductive fully_applied_neutral : nat -> Term -> Prop := (* hidden *).
Definition compute_defaultfun (t : Term) : option Term :=
  match t with
  | (Apply
      (Apply
        (Builtin AddInteger)
        (Constant (@Some _ DefaultUniInteger (ValueOf _ x)))
      )
      (Constant (@Some _ DefaultUniInteger (ValueOf _ y)))
    ) => Datatypes.Some (Constant (@Some valueOf DefaultUniInteger)
            (ValueOf DefaultUniInteger (x + y)))
  (* other pattern matches hidden *)
  end.
```

We remark that we can not sensibly compute the result of every fully applied built-in function as of yet. For example, the `VerifySignature` built-in function in the default collection of built-in functions has a non-trivial Haskell implementation that we currently can not model in Coq, so we let it return `true` in all cases. It would be better to parameterise our static and dynamic semantics over all possible collections of built-in functions. We would then require of such a parameter that it also defines a sensible `compute_defaultfun` function. As a consequence, we probably have to require from this parameter certain other proofs and definitions that we need to make our own proofs and definitions go through. In essence, we would postulate the existence of a set of built-in functions with some desired properties. Due to time constraints, we leave this as a problem for future work.

The **E-NeutralApplyPartial** and **E-NeutralTyInstPartial** rules describe "computation" rules for partially applied and partially type-instantiated built-in functions. Most importantly, the pre-evaluation term must not already be a neutral term: in that case, the first three rules apply. Note the similarity between the two rules and the **E-Apply** and **E-TyInst** rules. In this case, however, we can not substitute to continue the suspended computation. Instead, we just try to evaluate the resulting application or type instantiation and we imagine that some form of substitution could have taken place. If the resulting application or type instantiation would be a neutral term, then the **E-NeutralApply** or **E-NeutralTyInst** rule would apply. If the resulting application or type instantiation would be a fully applied built-in function, then the **E-NeutralApplyFull** or **E-NeutralTyInstFull** rule would apply.

(iv) Errors.

Figure 4.17 describes the evaluation of errors and the propagation of errors in computation rules. Firstly, the **E-Error** rules describes how an error evaluates to itself. The other rules describe how a sub-computation of a computation rule that evaluates to an error interrupts the computation, after which the error is propagated upwards.

(v) Final implementation.

The implementation of the operational semantics is a straightforward translation of the rules in Figures 4.14, 4.15, 4.16 and 4.17 to mutually defined inductive datatypes.

$$\frac{}{\texttt{Error}\ T \Downarrow^{0} \texttt{Error}\ T}\ \textbf{E-Error} \qquad \frac{t_1 \Downarrow^{j_1} \texttt{Error}\ T}{t_1\ t_2 \Downarrow^{j} \texttt{Error}\ T \text{ where } j = j_1 + 1}\ \textbf{E-Error-Apply1}$$

$$\frac{t_2 \Downarrow^{j_2} \texttt{Error}\ T}{t_1\ t_2 \Downarrow^{j} \texttt{Error}\ T \text{ where } j = j_2 + 1}\ \textbf{E-Error-Apply2} \qquad \frac{t_1 \Downarrow^{j_1} \texttt{Error}\ T}{t_1\ \{T_2\} \Downarrow^{j} \texttt{Error}\ T \text{ where } j = j_1 + 1}\ \textbf{E-Error-TyInst}$$

$$\frac{t_0 \Downarrow^{j_0} \texttt{Error}\ T'}{\texttt{IWrap}\ F\ T\ t_0 \Downarrow^{j} \texttt{Error}\ T' \text{ where } j = j_0 + 1}\ \textbf{E-Error-IWrap} \qquad \frac{t_0 \Downarrow^{j_0} \texttt{Error}\ T}{\texttt{Unwrap}\ t_0 \Downarrow^{j} \texttt{Error}\ T \text{ where } j = j_0 + 1}\ \textbf{E-Error-Unwrap}$$

$$\frac{t_1 \Downarrow^{j_1} \texttt{Error}\ T'}{\texttt{let}\ (\mathsf{NonRec})\ ((x : T\ (str) = t_1), \overline{b})\ \texttt{in}\ t_0 \Downarrow^{j} \texttt{Error}\ T' \text{ where } j = j_1 + 1}\ \textbf{E-Error-Let-TermBind}$$

Figure 4.17: Big-step operational semantics (errors)

```
Reserved Notation "t '=[' j ']=>' v" (* hidden *).
Reserved Notation "t '=[' j ']=>nr' v" (* hidden *).
Reserved Notation "t '=[' j ']=>r' v 'WITH' bs0" (* hidden *).
Inductive eval : Term -> Term -> nat -> Prop := (* hidden *)
with eval_bindings_nonrec : Term -> Term -> nat -> Prop := (* hidden *)
with eval_bindings_rec : list Binding -> Term -> Term -> nat -> Prop := (* hidden *)
where "t '=[' j ']=>' v" := (eval t v j)
and "t '=[' j ']=>nr' v" := (eval_bindings_nonrec t v j)
and "t '=[' j ']=>r' v 'WITH' bs0" := (eval_bindings_rec bs0 t v j).
```

As a sanity check for our evaluation relation, we prove that evaluation always results in a value, and we prove that (neutral) values evaluate to themselves in 0 steps.

```
Lemma eval_to_value :
    (forall t v k, t =[k]=> v -> value v) /\
    (forall t v k, t =[k]=>nr v -> value v) /\
    (forall bs0 t v k, t =[k]=>r v WITH bs0 -> value v).
Proof. (* hidden *). Qed.

Lemma eval_value :
    (forall v, value v -> v =[0]=> v) /\
    (forall n v, neutral_value n v -> v =[0]=> v).
Proof. (* hidden *). Qed.
```

Chapter 5

Definitions and methods for semantics preservation

With the syntax and semantics of PIR and Plutus Core defined in the previous chapter, we turn our attention to the concept of semantics preservation. Recall from Definition 2.2 in chapter 2 that we consider some translation relation R_i to be correct if $R_i(t_i, t_{i+1})$ implies $[\![t_i]\!]_i \sim_i [\![t_{i+1}]\!]_{i+1}$, where $[\![t_i]\!]_i$ is a semantics of an AST t_i and $\sim_i$ is a relation between such semantics that captures that two terms are the same. In this chapter, we present the definitions of static semantics preservation and dynamic semantics preservation, which are both examples of a relation $\sim_i$. The definition of static semantics preservation is simple, and we give a simple method of proving static semantics preservation. The definition of and proof-method for dynamic semantics preservation are more involved. In summary, we employ and extend *step-indexed syntactic logical relations* as presented by Ahmed[1] to define and prove dynamic semantics preservation.

Note on the use of previous work and our contributions. Our discussion of static semantics preservation is short, and does not depend on any specific previous works. Our methods and results pertaining to dynamic semantics preservation are derived from a paper and corresponding appendix by Ahmed[1]. We adapt the work by Ahmed to our setting, and thereby contribute with new results as well: we show that step-indexed syntactic logical relations also work in the context of higher-order kinds and we extend the proof technique to a larger lambda calculus with additional constructs such as `let`-bindings. Moreover, we present an implementation of the logical relation in Coq. Nevertheless, since we adapt and extend Ahmed's work, much of the remainder of this chapter will show significant similarities with the contents of their paper. For example, Figures 5.1 and 5.2 are adaptations of Figures that Ahmed present[1]. We hope that the reader does not find this surprising, and we hope that this paragraph gives sufficient credit to Ahmed.

Note on limitations. The logical relation does not take into account (partially applied) built-in functions as of yet. Furthermore, the logical relation can only be used to prove dynamic semantics preservation for translation relations that do not change contexts and/or types. We believe that this does not undermine our results since we expect that solutions to these limitations will consist mostly of technical overhead.

Note on proof completeness. The lemmas and theorems on which results in section 5.2 depend are not always fully finished, and therefore `Admitted`. However, we took care not to rely on lemmas and theorems that are unlikely to hold. We suspect that fully completing all lemmas and theorems will mostly consist of technical overhead. We hope that this paragraph offers sufficient transparency with regards to the completeness of the results.

5.1 Static semantics preservation

We can quickly think of the simplest notion of static semantics preservation: two related terms must be well-typed with respect to the same kind context, type context and type. In fact, we defined an almost identical notion (Definition 3.1) for a simply typed lambda calculus in chapter 3. We update the definition to work in the setting of the PIR and Plutus Core languages, and we will call it *strong* static semantics preservation:

Definition 5.1 (Strong static semantics preservation (SSSP)). *For all t_i and t_{i+1}, if $R_i(t_i, t_{i+1})$ and $\Delta; \Gamma \vdash t_i : T^n$, then $\Delta; \Gamma \vdash t_{i+1} : T^n$.*

We can imagine translation relations for which this definition of static semantics preservation does not hold. For example, we could simulate non-strict evaluation of certain sub-terms by the use of *thunking*, which adds a lambda abstraction that binds a variable of unit type on top of a sub-term. Of course, we must make sure to *force* thunks by applying a variable that refers to a thunk to a unit constant. As such, a sub-term before the transformation could have some type

T^n, while its thunked counterpart has some type $() \rightarrow T^n$, where $()$ denotes a unit type. In addition, not only does the type change, so does the type context, which can now contain additional term variable bindings that refer to unit types.

Because Definition 5.1 may not suffice as a definition for static semantics preservation, we instead create a more general definition. In this definition, we state that the kind contexts, type contexts and types before and after the translation must be pairwise related. To formulate this, we assume that there exist three additional binary relations R_i^Δ, R_i^Γ and R_i^T that relate, in sequence, kind contexts, type contexts and types with respect to some translation relation R_i. We can now define *weak* static semantics preservation:

Definition 5.2 (Weak static semantics preservation (WSSP))**.** *For all t_i and t_{i+1}, if $R_i(t_i, t_{i+1})$ and $\Delta_i; \Gamma_i \vdash t_i : T_i^n$ and $R_i^\Delta(\Delta_i, \Delta_{i+1})$ and $R_i^\Gamma(\Gamma_i, \Gamma_{i+1})$ and $R_i^T(T_i^n, T_{i+1}^n)$, then $\Delta_{i+1}; \Gamma_{i+1} \vdash t_{i+1} : T_{i+1}^n$.*

We can show that strong static semantics preservation is an instance of weak static semantics preservation by using the identity relation for our additional binary relations: weak static semantics preservation subsumes strong static semantics preservation. We achieve this level of generalisation by not assuming too much about the additional binary relations. As such, we hope to prevent that Definition 5.2 is still too strong, meaning that there exist translation relations in the compiler for which the definition of weak static semantics preservation does not hold. We do, however, find it unlikely: if no relations exist between the kind contexts, type contexts and types, then it might be hard or impossible to prove any sort of static semantics preservation.

Given the definitions of static semantics preservation, we can prove static semantics preservation of a translation relation by induction on the typing judgement of the pre-translation term. We expect that any non-trivial translation relation will also require carrying around additional information to prove the "interesting" cases of the proof, whereas congruence cases will most often be trivial to prove. In chapter 6, it will become clearer what we mean by the distinction between interesting cases and congruence cases.

5.2 Dynamic semantics preservation

In chapter 3, we presented a definition (Definition 3.2) of dynamic semantics preservation for a simply typed lambda calculus. We required that terms that are related by a translation relation should be semantically equivalent. In order to define semantic equivalence, we employed logical relations, which "specify relations on well-typed terms via structural induction on the syntax of types"[1]. We defined two terms to be semantically equivalent if the two terms are related by our logical relation. It is not unsurprising that his approach worked. One of the first works that successfully used the logical relations proof technique was a proof of strong normalisation for a simply typed lambda calculus by Tait[38], and our preliminary results are derived from a Coq implementation of this proof that is discussed in the Programming Language Foundations book[30].

Our approach in the preliminary study does not easily scale up to the Plutus Tx compiler. Both higher-order kinds and recursive types prove to be problematic. On one hand, higher-order kinds allow for computations in types that can be normalised, which means we can not define and implement a logical relation by straightforward structural induction on the syntax of types. On the other hand, the addition of recursive types allows for encoding arbitrary recursion, which in turn facilitates writing terms that do not terminate. Non-termination is a complicating factor in comparing the equivalence of two terms.

A previous work by Ahmed, called "Step-Indexed Syntactic Logical Relations for Recursive and Quantified types"[1], offers the solution. In the paper, Ahmed extends and improves on research by Appel and McAllester[2], who are the first to employ step-indexing. Step-indexed logical relations, which are logical relations that are indexed by the number of remaining steps for future execution, solve our two problems: the logical relation is now designed via well-foundedness using the step-index as a well-foundedness measure, and we only consider terminating terms because we upper-bound the number of steps that a term may make in the future. Moreover, the λ-calculi that Ahmed considers for their work have a large overlap with the PIR and Plutus Core languages. The fact that they consider recursive and quantified types, which are exactly the problematic language constructs for this thesis project, proves to be helpful. Ahmed's approach is also less complex than alternative methods of proving program equivalence that do not employ step-indexed logical relations[1].

In this section, we adapt and extend the work on step-indexed syntactic logical relations by Ahmed[1] to our setting. We will discuss how the logical relation and contextual equivalence are related. We will define and implement a relational model, with which we define and implement the full step-indexed logical relation. We discuss interesting results such as the Montonicity lemma, the Validity lemma and the compatibility lemmas. Ultimately, we discuss how we can use these interesting results and additional lemmas and theorems to prove the Fundamental Property, which states that the step-indexed logical relation is reflexive.

5.2.1 Contextual equivalence

A valid remark to make is that two logically related terms are not necessarily semantically equivalent. One definition of semantic equivalence that numerous works use is that of *contextual* equivalence: two terms are contextually equivalent if we can place them in the same program context and observe that they have the same behaviour. Ahmed shows that their step-indexed logical relation for recursive and universal types is sound and complete with respect to contextual equivalence, whereas they show that their step-indexed logical relation is only sound with respect to contextual equivalence if they include existential types[1]. Because of the overlap between their work and our languages, we believe it to be likely that we can also prove soundness and completeness of our adapted logical relation with respect to contextual equivalence. Yet, this is outside the scope of this thesis project, though it could be an interesting subject for future work.

5.2.2 Logical relation: computations and values

Figure 5.1 defines the first part of the relational model with which we define the full logical relation. In this figure, we define the relational interpretation of normal types as computations, which we denote by $\mathcal{RC}[\![T^n]\!]_\rho$, and the relational interpretation of normal types as values, which we denote by $\mathcal{RV}[\![T^n]\!]_\rho$. We define the relational interpretation of normal types as errors as well, which we denote by $\mathcal{RE}[\![T^n]\!]_\rho$, since we reuse $\mathcal{RE}[\![T^n]\!]_\rho$ in every definition of $\mathcal{RV}[\![T^n]\!]_\rho$. Note that sets are defined at *normal* types. Though we do not explicitly require that the types are normal in the relational interpretations, the definition of the full logical relation ensures that they are normal. Also, note that we use $t \Downarrow^* v$ to denote that a term t evaluates to a value v in *any* number of steps.

The relational interpretations are defined as sets on paper, though our implementation of relational interpretations does not use a concept of sets. Instead, we model relational interpretations as recursive functions that build up a logical proposition (Prop), such that the logical proposition is equivalent to set membership. For example, we could try to implement the relational interpretation $\mathcal{RC}[\![T^n]\!]_\rho$ of normal types as computations using some function RC, such that RC k T rho v v' holds if and only if $(k, v, v') \in \mathcal{RC}[\![T]\!]_\rho$.

```
Check RC : nat -> Ty -> tymapping -> Term -> Term -> Prop.
```

We use χ to denote sets of tuples (k, v, v), where k is a natural number step-index, and where v and v' are terms. We use Rel_{T_1,T_2} to denote a set of sets χ for which three properties hold with respect to types T_1 and T_2: (i) v and v' are values, (ii) v and v' are closed and well-typed with respect to the normalised versions of T_1 and T_2 respectively, and (iii) χ must be monotone – that is, "χ must be closed with respect to a decreasing step-index"[1]. We do not require that T_1 and T_2 are already normal: this would be too restrictive. In essence, Rel_{T_1,T_2} defines a relation on values (with respect to a step-index) for some types T_1 and T_2. The definition of the full logical relation ensures that T_1 and T_2 are closed types as well. We implement $\chi \in Rel_{T_1,T_2}$ as a logical proposition that is equivalent to set membership.

```
Definition Rel (T T' : Ty) (Chi : nat -> Term -> Term -> Prop) : Prop :=
  forall j v v',
   Chi j v v' -> 0 < j ->
     value v /\ value v' /\
     (exists Tn, normalise T Tn /\ (empty ,, empty |-+ v : Tn)) /\
     (exists Tn', normalise T' Tn' /\ (empty ,, empty |-+ v' : Tn')) /\
     forall i,
       i <= j ->
       Chi i v v'.
```

We require that the step-index is strictly larger than 0 in the implementation, which is a consequence of how we implement $\mathcal{RC}[\![T^n]\!]_\rho$, $\mathcal{RV}[\![T^n]\!]_\rho$ and $\mathcal{RE}[\![T^n]\!]_\rho$. For now, it is sufficient to know that this will not cause problems.

The relational interpretations are parameterised by ρ, which denotes a type mapping. We need such a mapping because types can contain free type variables. To be more precise, we define type mappings such that they can be used both as a *semantic* substitution and a *syntactic* substitution. As a semantic substitution, we want ρ to substitute a relation on values for a type variable. We use the semantic substitution primarily in the definition of $\mathcal{RV}[\![X]\!]_\rho$ in Figure 5.1. As a syntactic substitution, we want ρ to substitute concrete, closed types for type variables. We primarily use the syntactic substitution to close off types in Figure 5.1 before we use P to check that values are closed and well-typed.

We define type mappings as partial maps that map type variables to triplets of the form (χ, T_1, T_2). We define some additional notation: given that $\rho(X) = (\chi, T_1, T_2)$, then $\rho_{sem}(X) = \chi$. The definition of the full logical relation makes sure that $\chi \in Rel_{T_1,T_2}$. If $\rho = \{X_1 \mapsto (\chi, T_{1,1}, T_{1,2}), \ldots, X_n \mapsto (\chi, T_{n,1}, T_{n,2})\}$, then $\rho_{msyn1}(T) = \left[T_{1,1}/X_1, \ldots, T_{n,1}/X_n\right] T$, and $\rho_{msyn2}(T) = \left[T_{1,2}/X_1, \ldots, T_{n,2}/X_n\right] T$. As such, ρ_{msyn1} and ρ_{msyn2} perform syntactic multi-substitutions on types. We implement a type mapping as a list of tuples with auxiliary operations that enable semantic substitutions and syntactic multi-substitutions.

$$
\begin{aligned}
P(T_1, T_2, v, v') = {} & v \in value \land T_1 \rightsquigarrow T_1^n \land \emptyset; \emptyset \vdash v : T_1^n \land \\
& v' \in value \land T_2 \rightsquigarrow T_2^n \land \emptyset; \emptyset \vdash v' : T_2^n
\end{aligned}
$$

$$
\begin{aligned}
Rel_{T_1,T_2} & \overset{def}{=} \{\chi \mid \forall (j, v, v') \in \chi. \\
& \qquad\qquad P(T_1, T_2, v, v') \land \\
& \qquad\qquad \forall i \leq j.(i, v, v') \in \chi\} \\
\mathcal{RE}[\![T^n]\!]_\rho & = \{(k, v, v) \mid P(\rho_{msyn1}(T^n), \rho_{msyn2}(T^n), v, v') \land \\
& \qquad\qquad isError(v) \land isError(v')\} \\
\mathcal{RV}[\![X]\!]_\rho & = \{(k, v, v') \mid P(\rho_{msyn1}(X), \rho_{msyn2}(X), v, v') \land \\
& \qquad\qquad \neg isError(v) \land \neg isError(v') \land \\
& \qquad\qquad \rho_{sem}(X) = \chi \land (k, v, v') \in \chi\} \cup \mathcal{RE}[\![X]\!]_\rho \\
\mathcal{RV}[\![\lambda X :: K.T]\!]_\rho & = \emptyset \cup \mathcal{RE}[\![\lambda X :: K.T]\!]_\rho \\
\mathcal{RV}[\![T_1\ T_2]\!]_\rho & = \emptyset \cup \mathcal{RE}[\![T_1\ T_2]\!]_\rho \\
\mathcal{RV}[\![\mathbb{U}_i]\!]_\rho & = \{(k, v, v') \mid P(\rho_{msyn1}(\mathbb{U}_i), \rho_{msyn2}(\mathbb{U}_i), v, v') \land \\
& \qquad\qquad v = \texttt{Constant}\ \mathbb{U}_i\ c \land v' = \texttt{Constant}\ \mathbb{U}_i\ c' \land \\
& \qquad\qquad c = c'\} \cup \mathcal{RE}[\![\mathbb{U}_i]\!]_\rho \\
\mathcal{RV}[\![T_1^n \rightarrow T_2^n]\!]_\rho & = \{(k, v, v') \mid P(\rho_{msyn1}(T_1^n \rightarrow T_2^n), \rho_{msyn2}(T_1^n \rightarrow T_2^n), v, v') \land \\
& \qquad\qquad v = \lambda x : T_1.e \land v' = \lambda x : T_1'.e' \land \\
& \qquad\qquad \forall j < k, v_0, v_0'. \\
& \qquad\qquad\quad (j, v_0, v_0') \in \mathcal{RV}[\![T_1^n]\!]_\rho \implies \\
& \qquad\qquad\quad (j, [v_0/x]\ e, [v_0'/x]\ e') \in \mathcal{RC}[\![T_2^n]\!]_\rho\} \cup \mathcal{RE}[\![T_1^n \rightarrow T_2^n]\!]_\rho \\
\mathcal{RV}[\![\texttt{ifix}\ F^n\ T^n]\!]_\rho & = \{(k, v, v') \mid P(\rho_{msyn1}(\texttt{ifix}\ F^n\ T^n), \rho_{msyn2}(\texttt{ifix}\ F^n\ T^n), v, v') \land \\
& \qquad\qquad v = \texttt{IWrap}\ F\ T\ v_0 \land v' = \texttt{IWrap}\ F'\ T'\ v_0' \land \\
& \qquad\qquad \forall j < k. \\
& \qquad\qquad\quad \emptyset \vdash \rho_{msyn1}(T^n) :: K \implies \\
& \qquad\qquad\quad \emptyset \vdash \rho_{msyn2}(T^n) :: K \implies \\
& \qquad\qquad\quad (F^n\ (\lambda X :: K.\texttt{ifix}\ F^n\ X))\ T^n \rightsquigarrow T_0^n \implies \\
& \qquad\qquad\quad (j, v_0, v_0') \in \mathcal{RV}[\![T_0^n]\!]_\rho\} \cup \mathcal{RE}[\![\texttt{ifix}\ F^n\ T^n]\!]_\rho \\
\mathcal{RV}[\![\forall X :: K.T^n]\!]_\rho & = \{(k, v, v') \mid P(\rho_{msyn1}(\forall X :: K.T^n), \rho_{msyn2}(\forall X :: K.T^n), v, v') \land \\
& \qquad\qquad v = \Lambda X :: K.e \land v' = \Lambda X :: K.e' \land \\
& \qquad\qquad \forall T_1, T_2, \chi. \\
& \qquad\qquad\quad \emptyset \vdash T_1 :: K \implies \\
& \qquad\qquad\quad \emptyset \vdash T_2 :: K \implies \\
& \qquad\qquad\quad \chi \in Rel_{T_1,T_2} \implies \\
& \qquad\qquad\quad \forall j < k. \\
& \qquad\qquad\qquad (j, [T_1/X]\ e, [T_2/X]\ e') \in \mathcal{RC}[\![T^n]\!]_{\rho[X \mapsto (\chi, T_1, T_2)]}\} \cup \mathcal{RE}[\![\forall X :: K.T^n]\!]_\rho
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{RC}[\![T^n]\!]_\rho = \{(k, e, e') \mid & \forall j < k, e_f. \\
& \quad e \Downarrow^j e_f \implies \\
& \quad \exists e_f'.e' \Downarrow^* e_f' \land (k - j, e_f, e_f') \in \mathcal{RV}[\![T^n]\!]_\rho\}
\end{aligned}
$$

Figure 5.1: Logical relation (computations and values)

```
Definition tymapping := list (tyname * ((nat -> Term -> Term -> Prop) * Ty * Ty)).
Fixpoint sem (rho : tymapping) (a : tyname) : option (nat -> Term -> Term -> Prop) :=
  (* hidden *).
Fixpoint msyn1 (rho : tymapping) : list (tyname * Ty) := (* hidden *)
Fixpoint msyn2 (rho : tymapping) : list (tyname * Ty) := (* hidden *)
```

Given some tymapping rho, we use the msubstT function from section 4.2 to apply the actual syntactic multi-substitution on types.

```
Check msubstT (msyn1 rho) : Ty -> Ty.
```

We must make a concession when we implement the relational interpretations of computations and values. $\mathcal{RC}[\![T^n]\!]_\rho$ and $\mathcal{RV}[\![T^n]\!]_\rho$ are not structurally recursive on normal types, but they are well-founded by the <-relation on natural numbers. This means that we must prove well-foundedness of our implementation, instead of using Coq's default structural recursion checker. We would like to use the Equations package since it simplifies implementing well-founded recursive functions, but the package does not support mutual well-founded definitions. This seems to be problematic since $\mathcal{RC}[\![T^n]\!]_\rho$ and $\mathcal{RV}[\![T^n]\!]_\rho$ are mutually recursive, but we can solve this problem by combining the implementations of the two into a single function. We make sure to also implement $\mathcal{RE}[\![T^n]\!]_\rho$ in the same function. Interestingly, it is not problematic to combine the three relational interpretations in a single implementation. We implement $\mathcal{RC}[\![T^n]\!]_\rho$ by “inlining” the right-hand-sides of $\mathcal{RV}[\![T^n]\!]_\rho$ and $\mathcal{RE}[\![T^n]\!]_\rho$.

```
1  Equations? RC (k : nat) (T : Ty) (rho : tymapping) (e e' : Term) : Prop by wf k :=
2    RC k T rho e e' =>
3      forall j (Hlt_j : j < k) e_f,
4        e =[j]=> e_f ->
5        exists e'_f j', e' =[j']=> e'_f /\
6
7        (exists Tn, normalise (msubstT (msyn1 rho) T) Tn /\ (empty ,, empty |-+ e_f : Tn)) /\
8        (exists Tn', normalise (msubstT (msyn2 rho) T) Tn' /\ (empty ,, empty |-+ e'_f : Tn')) /\
9
10       (
11         (
12           ~ is_error e_f /\
13           ~ is_error e'_f /\
14           (
15             match T with
16             (* other matches hidden *)
17             | Ty_Fun T1n T2n =>
18                 exists x e_body e'_body T1 T1',
19                   LamAbs x T1 e_body = e_f /\
20                   LamAbs x T1' e'_body = e'_f /\
21                   forall i (Hlt_i : i < k - j) v_0 v'_0,
22                     value v_0 /\ value v'_0 /\ RC i T1n rho v_0 v'_0 ->
23                     RC i T2n rho <{ [v_0 / x] e_body }> <{ [v'_0 / x] e'_body }>
24             (* other matches hidden *)
25             end
26           )
27         ) \/ (
28           is_error e_f /\
29           is_error e'_f
30         )
31       ).
32 Proof. all: lia. Qed.
```

In the above implementation, lines 3-5 are specific to $\mathcal{RC}[\![T^n]\!]_\rho$: for any j that is strictly less than k, the fact that e evaluates to e_f in j steps must imply that there exists also some e'_f to which e' evaluates in *any* number of steps. We then require that e_f and e'_f must be related as values for the $k - j$ steps that remain for future execution steps. The largest part of the function, which consists roughly of lines 7-31, is specific to $\mathcal{RV}[\![T^n]\!]_\rho$ and $\mathcal{RE}[\![T^n]\!]_\rho$.

We require that e_f and e'_f are closed and well-typed with respect to a closed and normalised version of T in lines 7-8. This is accomplished in figure 5.1 using the function P. Since we close off T, we must also re-normalise it as substitution can introduce beta redexes.

After the closedness and well-typedness constraints, we incorporate $\mathcal{RE}[\![T^n]\!]_\rho$ in the function. Either e_f and e'_f are both errors, in which case we consider them trivially related, or they are both not errors and we must satisfy some additional type-specific constraints. In figure 5.1, we accomplish the same distinction between errors and non-errors by applying set union with $\mathcal{RE}[\![T^n]\!]_\rho$ in the definition of each $\mathcal{RV}[\![T^n]\!]_\rho$.

The last part of the function, which spans lines 15-25, consists of the `match`-expression. We introduce additional constraints in each `match`-case that are specific to the matched type. We have omitted most of the `match`-cases in the Coq proof script snippet above, though we will discuss what the type-specific parts of each $\mathcal{RV}[\![T^n]\!]_\rho$ mean. We did list the `match`-case for function types as it nicely illustrates how we implement the definitions from Figure 5.1.

Type variables (X). Values should be related with respect to the semantic substitution $\rho_{sem}(X)$.

Lambda abstraction and function application types ($\lambda X :: K.T^n$ or $T_1^n\ T_2^n$). Values can not be related at lambda abstraction and function application types since there is no $v \in \mathit{value}$ such that v has a lambda abstraction or function application type.

Built-in types ($\mathbb{U}_i$). Since only constants can have built-in types, values are related at a built-in type only if the values are syntactically equal constants.

Function types ($T_1^n \to T_2^n$). We do not handle built-in functions as of yet. Therefore, only lambda abstractions can have function types, so the values must be of the form $\lambda x : T_1.e$ and $\lambda x : T'_1.e$ for *some* types T_1 and T'_1. We do not need to know how T_1 and T'_1 are related to T_1^n. The main takeaway should be that the values are lambda abstractions. Now, since lambda abstractions are suspended computations, we can define two lambda abstractions to be related at the function type if they are *extensionally* related: applying them to related arguments leads to related results. As the terminology suggests, the continuations of suspended computations must be related in terms of the relational interpretation of normal types as computations. As lambda abstractions might be applied to arbitrary terms, we must also consider what it means if terms in the argument position must be further reduced before we can perform beta reduction. Because of this, we consider some moment in the future where we have $j < k$ steps remaining for future execution. Then, given two values that are related at the type T_1^n for j steps, the continued computations of the lambda abstractions must be related as computations at the type T_2^n for j steps.

Fixpoint types (`ifix` $F^n\ T^n$). Values that have a fixpoint type must have the form of `IWrap` $F\ T\ v_0$ and `IWrap` $F'\ T'\ v'_0$ for some types F, T, F' and T'. Again, we do not need to know how these type annotations relate to F^n and T^n: the main takeaway should be that the values are `IWrap` values. Now, recall that the terms v_0 and v'_0 are themselves values. We define the relational interpretation such that the `IWrap` values are related if the unwrapped values are related.

Universally quantified types ($\forall X :: K.T^n$). Values that are related at a universal type must be type abstractions of the form $\Lambda X :: K.e$ and $\Lambda X :: K.e'$. They are related in a somewhat similar way to how lambda abstractions are related at a function type. However, we can pick arbitrary K-kinded types T_1 and T_2 for type instantiation instead of related types. Since these types are not syntactically equal, we must delay the syntactic substitution of types by using a type mapping ρ. Moreover, we must pick a semantic substitution $\chi \in Rel_{T_1,T_2}$ in case we have to relate two values at the type variable X at some point. In summary, we define the relational interpretation such that instantiation with arbitrary types should lead to related results.

Picking arbitrary types for type instantiation allows us to prove a larger class of contextual equivalence statements than when we would have picked related types. More specifically, we can prove a larger class of *free theorems*[41]: theorems that follow directly from types without having to inspect their term inhabitants. One simple example of a free theorem is that any two terms of a type $\forall X :: *.X \to X$ must be contextually equivalent since the only term that has this type is the identity function. It can be argued that allowing for the picking of arbitrary types makes the logical relation unnecessarily expressive. We only need to prove contextual equivalence of terms that are related by a translation relation, and these translation relations are not likely to make profound changes in types and type annotations. Because of this, it might have been sufficient to assume that type abstractions are only ever instantiated with related types, or even equal types. Nevertheless, we choose not to too stray too far from Ahmed's work.

In our implementation, we can still differentiate between computations (`RC`) and values (`RV`). We define `RV` such that it is a special case of `RC`.

```
Definition RV (k : nat) (T : Ty) (rho : tymapping) (v v' : Term) : Prop :=
  value v /\ value v' /\ RC k T rho v v'.
```

$$
\begin{aligned}
\mathcal{RD}\llbracket \emptyset \rrbracket &= \{\emptyset\} \\
\mathcal{RD}\llbracket \Delta, X :: K \rrbracket &= \{\rho\,[X \mapsto (\chi, T_1, T_2)] \mid \rho \in \mathcal{RD}\llbracket \Delta \rrbracket \wedge \emptyset \vdash T_1 :: K \wedge \emptyset \vdash T_2 :: K \wedge \chi \in Rel_{T_1,T_2}\} \\
\mathcal{RG}\llbracket \emptyset \rrbracket_\rho &= \{(k, \emptyset, \emptyset)\} \\
\mathcal{RG}\llbracket \Gamma, x : T^n \rrbracket_\rho &= \{(k, \gamma\,[x \mapsto v], \gamma'\,[x \mapsto v']) \mid (k, \gamma, \gamma') \in \mathcal{RG}\llbracket \Gamma \rrbracket_\rho \wedge T^n \in normal \wedge (k, v, v') \in \mathcal{RV}\llbracket T^n \rrbracket_\rho\}
\end{aligned}
$$

$$
\begin{aligned}
\Delta; \Gamma \vdash e \leq e' : T^n \overset{def}{=}\ & \Delta; \Gamma \vdash e : T^n \wedge \Delta; \Gamma \vdash e' : T^n \wedge \\
& (\forall k \geq 0. \forall \rho, \gamma, \gamma'. \\
& \qquad \rho \in \mathcal{RD}\llbracket \Delta \rrbracket \wedge (k, \gamma, \gamma') \in \mathcal{RG}\llbracket \Gamma \rrbracket_\rho \implies \\
& \qquad (k, \gamma(\rho_{msyn1}(e)), \gamma'(\rho_{msyn2}(e'))) \in \mathcal{RC}\llbracket T^n \rrbracket_\rho) \\
\Delta; \Gamma \vdash e \sim e' : T^n \overset{def}{=}\ & \Delta; \Gamma \vdash e \leq e' : T^n \wedge \Delta; \Gamma \vdash e' \leq e : T^n
\end{aligned}
$$

Figure 5.2: Logical relation (contexts, approximation and equivalence)

We can move from RC to RV given that the first of the related expressions evaluates to a value in $j < k$ steps. We can move immediately from RV to RC.

```
Lemma RC_to_RV : forall k T rho e e',
    RC k T rho e e' ->
    forall j (Hlt_j : j < k) e_f,
      e =[j]=> e_f ->
      exists e'_f j', e' =[j']=> e'_f /\
        RV (k - j) T rho e_f e'_f.
Proof. (* hidden *). Qed.

Lemma RV_to_RC : forall k T rho v v',
    RV k T rho v v' ->
    RC k T rho v v'.
Proof. (* hidden *). Qed.
```

There is one peculiar consequence of this RV definition. Consider some proof context that includes a hypothesis (`H : RV k T rho v v'`). Since values evaluate to themselves in 0 steps, we can only ever hope to extract information from H if $0 < k$. For example, given such a hypothesis H we sometimes need to extract the information that v and v' are closed and well-typed. We do not run into such problems in our implementation: by construction of the dynamic semantics and the full logical relation, we only need to extract such information if $0 < k$.

5.2.3 Logical relation: contexts, approximation and equivalence

In the previous subsection we defined and implemented the relational interpretations of normal types as computations and values. In this subsection, we define and implement the relational interpretation of kind contexts as type mappings, which we denote by $\mathcal{RD}\llbracket \Delta \rrbracket$, and the relational interpretation of type contexts as term substitutions, which we denote by $\mathcal{RG}\llbracket \Gamma \rrbracket_\rho$. Lastly, we define and implement the full logical relation. Figure 5.2 defines these last relational interpretations and the full logical relation.

The relational interpretation $\mathcal{RD}\llbracket \Delta \rrbracket$ of kind contexts is a set of type mappings ρ. The definition is such that the domains of ρ and Δ match if $\rho \in \mathcal{RD}\llbracket \Delta \rrbracket$. Furthermore, the definition ensures that the syntactic substitutions have the correct kind and that the semantic substitutions follow from their corresponding syntactic substitutions.

```
Definition kass := list (name * Kind).
Inductive RD : kass -> tymapping -> Prop :=
  | RD_nil :
      RD nil nil
  | RD_cons : forall ck rho T1 T2 Chi X K,
      empty |-* T1 : K ->
      empty |-* T2 : K ->
      Rel T1 T2 Chi ->
      RD ck rho ->
      RD ((X, K) :: ck) ((X, (Chi, T1, T2)) :: rho).
```

We do not relate kind contexts to type mappings in our implementation. Instead, we relate *kind assignments* to type mappings. A kind assignment `ck` can still be used to represent a kind context by using `mupdate`. Additionally, we can use the `msubstA_term` function from section 4.3 to apply syntactic multi-substitutions on type annotations.

```
Check mupdate empty ck : Delta.
Check msubstA_term (msyn1 rho) : Term -> Term.
Check msubstA_term (msyn2 rho) : Term -> Term.
```

The relational interpretation $\mathcal{RG}[\![\Gamma]\!]_\rho$ of type contexts is a set of triplets (k, γ, γ'). Here, k is a step-index and γ and γ' are term substitutions. Again, the definition is such that the domains of γ, γ' and Γ match if $(k, \gamma, \gamma') \in \mathcal{RG}[\![\Gamma]\!]_\rho$. The definition also ensures that the substituents have the correct type, and that γ and γ' map variables to values that are related for k steps at normal types.

```
Definition env := list (name * Term).
Definition tass := list (name * Ty).
Inductive RG (rho : tymapping) (k : nat) : tass -> env -> env -> Prop :=
  | RG_nil :
      RG rho k nil nil nil
  | RG_cons : forall x T v1 v2 c e1 e2,
      RV k T rho v1 v2 ->
      normal_Ty T ->
      RG rho k c e1 e2 ->
      RG rho k ((x, T) :: c) ((x, v1) :: e1) ((x, v2) :: e2).
```

In our implementation, we again do not directly relate type contexts to term substitutions. Instead, we relate *type assignments* to two *term environments*. We can use a type assignment `ct` to represent a type context by using `mupdate`, and we can use the term environments `e1` and `e2` to represent multi-substitutions on terms using the `msubst_term` function from section 4.3:

```
Check mupdate empty ct : Gamma.
Check msubst_term e1 : Term -> Term.
Check msubst_term e2 : Term -> Term.
```

We say that a term e logically approximates a term e' with respect to Δ, Γ and T^n if the following two properties hold: (i) both e and e' must be well-typed with respect to Δ, Γ and T^n, and (ii) we require that the *closed instantiations* of e and e', if such closed instantiations exist, are related as computations at the type T^n for some k steps. We employed the same approach of reasoning about closed instantiations of terms instead of reasoning about closed terms directly in chapter 3. We achieve closed instantiations by applying the syntactic type annotation substitutions and term substitutions, which we obtain from the relational interpretations of kind and type contexts, to e and e'. This should yield closed terms, since (i) the domains of the substitutions match the domains of the contexts, since (ii) the free (type) variables of e and e' should be subsets of the context domains, and since (iii) we only substitute closed types and terms. Note also that we only substitute values that are related for k steps at normal types.

We denote logical approximation as $\Delta; \Gamma \vdash e \leq e' : T^n$. Two terms are logically equivalent if they both logically approximate each other, which we denote by $\Delta; \Gamma \vdash e \sim e' : T^n$.

```
Definition LR_logically_approximate (Delta : partial_map Kind) (Gamma : partial_map Ty)
  (e e' : Term) (T : Ty) :=
    (Delta ,, Gamma |-+ e : T) /\
    (Delta ,, Gamma |-+ e' : T) /\
    forall k rho env env' ct ck,
      Delta = mupdate empty ck ->
      Gamma = mupdate empty ct ->
      RD ck rho ->
      RG rho k ct env env' ->
      RC k T rho
        (msubst_term env (msubstA_term (msyn1 rho) e))
        (msubst_term env' (msubstA_term (msyn2 rho) e')).

Definition LR_logically_equivalent (Delta : partial_map Kind) (Gamma : partial_map Ty)
  (e e' : Term) (T : Ty) :=
    LR_logically_approximate Delta Gamma e e' T /\
    LR_logically_approximate Delta Gamma e' e T.
```

5.2.4 Monotonicity and Validity

Monotonicity and Validity are important lemmas that we prove about our implementation. Monotonicity states that RC, RV and RG are *monotone* – that is, RC, RV and RG are "closed with respect to a decreasing step-index"[1]. As a consequence, we can freely replace a step-index with a smaller step-index without invalidating the corresponding relation.

```
Lemma RC_monotone : forall k rho T i e e' ck,
    RD ck rho -> RC k T rho e e' -> i <= k -> RC i T rho e e'.
Proof. (* hidden *). Qed.
```

The fact that RV is monotone follows easily from the fact that RC is monotone, since RV is implemented as a special case of RC.

```
Lemma RV_monotone : forall k rho T i v v' ck,
    RD ck rho -> RV k T rho v v' -> i <= k -> RV i T rho v v'.
Proof. (* proof using RC_monotone hidden *). Qed.
```

The fact that RG is monotone follows easily from the fact that RV is monotone, since we relate values by RV in the definition of RG.

```
Lemma RG_monotone : forall c ck rho i k env env',
    RD ck rho -> RG rho k c env env' -> i <= k -> RG rho i c env env'.
Proof. (* proof using RV_monotone hidden *). Qed.
```

With the Monotonicity lemmas proven, we can prove the Validity lemma. This lemma states that all types are *valid* – that is, the relational interpretation of values at any type T^n is in $Rel_{\rho_{msyn1}(T^n),\rho_{msyn2}(T^n)}$. More formally, we prove that $\mathcal{RV}[\![T^n]\!]_\rho \in Rel_{\rho_{msyn1}(T^n),\rho_{msyn2}(T^n)}$.

Recall from the definition of Rel_{T_1,T_2} that three properties must hold for some set $\chi = (k, v, v')$ such that $\chi \in Rel_{T_1,T_2}$: (i) v and v' are values, (ii) v and v' are closed and well-typed with respect to the normalised versions of T_1 and T_2 respectively, and (iii) χ must be monotone. We can extract (i) and (ii) directly from $\mathcal{RV}[\![T^n]\!]_\rho$, and we have (iii) by the Monotonicity lemmas.

```
Lemma validity : forall ck rho T,
    RD ck rho ->
    Rel (msubstT (msyn1 rho) T) (msubstT (msyn2 rho) T) (fun k e e' => RV k T rho e e').
Proof. (* proof using RV_monotone hidden *). Qed.
```

5.2.5 Fundamental Property

The last and most vital part of defining a logical relation is to prove the Fundamental Property, which states that the logical relation is reflexive. If the logical relation is not reflexive, then we can not prove that a term is logically related to itself, and therefore not contextually equivalent to itself, which is nonsensical. We will prove that a well-typed term e logically approximates itself.

Theorem 5.3 (Fundamental Property (i)). *Given that* $\Delta; \Gamma \vdash e : T^n$, *then* $\Delta; \Gamma \vdash e \leq e : T^n$.

Proof. We prove the Fundamental Property by induction on the typing judgement for e.

```
Lemma LR_reflexivity : forall Delta Gamma e T,
    Delta ,, Gamma |-+ e : T ->
    LR_logically_approximate Delta Gamma e e T.
Proof. (* hidden *). Qed.
```

To make the proof go through, we depend on a range of lemmas and theorems. These lemmas include, amongst others: lemmas that state that our various kinds of substitution preserve typing and kinding, lemmas that state that the type-level language has the Preservation and Strong Normalisation properties, lemmas that state that our big-step operational semantics is deterministic, lemmas that prove congruence for multi-substitutions, and many more. Most importantly, we define a set of compatibility lemmas. Each compatibility lemma generally proves a single case of the reflexivity proof. For example, we prove a compatibility lemma for the proof case that corresponds to the **T-Apply** rule from Figure 4.12.

```
Lemma compatibility_Apply : forall Delta Gamma e1 e2 e1' e2' T1n T2n,
    LR_logically_approximate Delta Gamma e1 e1' (Ty_Fun T1n T2n) ->
    LR_logically_approximate Delta Gamma e2 e2' T1n ->
    LR_logically_approximate Delta Gamma (Apply e1 e2) (Apply e1' e2') T2n.
Proof. (* hidden *). Admitted.
```

The meaning of this compatibility lemma can be summarised: applying approximating functions to approximating arguments results in an approximation. If we define a compatibility lemma for roughly each type rule, then the proofs for most cases of the `LR_reflexivity` lemma follow almost directly from the corresponding compatibility lemmas. By proving the Fundamental Property, we have also fully defined the logical relation.

□

From Theorem 5.3, it follows directly that any well-typed term e is logically equivalent to itself.

Corollary 5.4 (Fundamental Property (ii))**.** *Given that* $\Delta; \Gamma \vdash e : T^n$*, then* $\Delta; \Gamma \vdash e \sim e : T^n$*.*

5.2.6 From logical approximation to semantics preservation

Finally, we can update our definition of dynamic semantics preservation. We consider a translation relation to be dynamic semantics preserving if the pre-translation term logically approximates the post-translation term. This new definition does not seem to be in line with our earlier statement that logically related terms are not necessarily contextually equivalent. However, since we have not yet proven any soundness and completeness results, we will use the logical relation in our definition for now. A proof of dynamic semantics preservation typically proceeds by induction on the type derivation.

Definition 5.5 (Dynamic semantics preservation)**.** *For all* t_i *and* t_{i+1}*, if* $R_i(t_i, t_{i+1})$ *and* $\Delta_i; \Gamma_i \vdash t_i : T_i^n$*, then* $\Delta_i; \Gamma_i \vdash t_i \leq t_{i+1} : T_i^n$*.*

If future work were to prove that the logical relation is sound and/or complete with respect to contextual equivalence, that work should update the definition of dynamic semantics preservation to incorporate contextual equivalence instead of logical approximation. As such, instead of being used in definition directly, the logical relation would primarily be a means to prove contextual equivalence.

Lastly, we can gather from the new definition that we can not yet prove theorems that are based on Definition 5.5 for translation relations that change contexts and/or types. However, we feel that a solution to this limitation will mostly consist of technical overhead. It is probable that this solution will make a distinction between strong and weak dynamic semantics preservation, which is similar to how we made a distinction between strong and weak static semantics preservation.

Chapter 6

Proofs of semantics preservation

In this chapter, we will prove strong static semantics preservation and dynamic semantics preservation for an example translation relation: the desugaring of non-recursive `let`-bindings. This translation relation characterises a real compiler pass in the Plutus Tx compiler. We will first describe the translation relation, after which we prove the two types of semantics preservation.

Note on proof completeness. The results in section 6.3 depend on the same lemmas and theorems that the results from section 5.2 depend on. As such, we hope that this paragraph and the **Note on proof completeness** paragraph at the start of chapter 5 offer sufficient transparency with regards to the completeness of the results.

6.1 The translation relation

Figure 6.1 defines the translation relation R_{CNR} that describes the desugaring of non-recursive `let`-bindings. The compiler pass that this translation relation characterises compiles non-recursive binding sequences consisting of just strict term bindings to explicit applications of lambda abstractions. The **CNR-Binding** rule defines how a strict term binding that binds a term t_b can be translated to a function f_b. This function is defined such that, given an argument t, we add the compiled term binding on top of t. The **CNR-Bindings-Nil** and **CNR-Bindings-Cons** rules define how non-recursive sequences of strict term bindings can be translated to sequences of functions that add compiled term bindings on top of an argument term. In the **CNR-Let** rule, we compose the functions in the sequence $\overline{f_b}$ and apply them to the body of the `let`-binding. Furthermore, the **CNR-Cong** rule defines that any term, binding or binding sequence can also be related as a congruence. For example, Figure 6.2 shows the conclusion of an example desugaring derivation where one `let`-binding is desugared, whereas the other is not.

We implement the translation relation as an inductive datatype. Bindings are related to functions on terms through `CNR_Binding`, while lists of bindings are related to lists of functions on terms through `CNR_Bindings`. Terms are related to terms through `CNR_Term`, where we have the choice of either relating the terms as a desugaring or as a congruence. We use right folds over lists to compose the functions on terms that result from desugaring, and we use the `Cong` relation to express congruence with respect to `CNR_Term`.

```
Inductive CNR_Term : Term -> Term -> Type :=
  | CNR_Let : forall {bs t_body t_body' f_bs},
     CNR_Term t_body t_body' ->
     CNR_Bindings bs f_bs ->
     CNR_Term (Let NonRec bs t_body) (fold_right apply t_body' f_bs )
  | CNR_Cong : forall {t t'}, Cong CNR_Term t t' -> CNR_Term t t'
with CNR_Bindings : list Binding -> list (Term -> Term) -> Type :=
  | CNR_Nil : CNR_Bindings nil nil
  | CNR_Cons : forall {b bs f_b f_bs},
     CNR_Binding      b   f_b ->
     CNR_Bindings     bs  f_bs ->
     CNR_Bindings (b :: bs) (f_b :: f_bs )
with CNR_Binding : Binding -> (Term -> Term) -> Type :=
  | CNR_Desugar : forall {v t_bound t_bound' ty},
     CNR_Term t_bound t_bound' ->
     CNR_Binding
       (TermBind Strict (VarDecl v ty) t_bound)
       (fun t_bs => Apply (LamAbs v ty t_bs) t_bound').
```

$$\frac{\vdash t_0 \triangleright t_0' \qquad \vdash \overline{b} \triangleright \overline{f_b}}{\vdash \texttt{let (NonRec) } \overline{b} \texttt{ in } t_0 \triangleright (\overline{f_b} \circ \mathrm{id})\ t_0'}\ \textbf{CNR-Let (i)} \qquad \frac{\mathrm{Cong}\ t\ t'}{\vdash t \triangleright t'}\ \textbf{CNR-Cong (ii)}$$

$$\frac{}{\vdash \varepsilon \triangleright \varepsilon}\ \textbf{CNR-Bindings-Nil (iii)} \qquad \frac{\vdash b \triangleright f_b \qquad \vdash \overline{b'} \triangleright \overline{f_{b'}}}{\vdash b, \overline{b'} \triangleright f_b, \overline{f_{b'}}}\ \textbf{CNR-Bindings-Cons (iv)}$$

$$\frac{\vdash t_b \triangleright t_b' \qquad f_b(t) \overset{def}{=} ((\lambda x : T.t)\ t_b')}{\vdash (x : T\ (\texttt{Strict}) = t_b) \triangleright f_b}\ \textbf{CNR-Binding (v)}$$

Figure 6.1: Desugaring of non-recursive `let`-bindings (R_{CNR})

$$\frac{\vdots}{\vdash ((\texttt{let (NonRec) } \overline{b_1} \texttt{ in } t_1)\ (\texttt{let (NonRec) } \overline{b_2} \texttt{ in } t_2)) \triangleright ((\texttt{let (NonRec) } \overline{b_1'} \texttt{ in } t_1')\ ((\overline{f_{b_2}} \circ \mathrm{id})\ t_2'))}$$

Figure 6.2: Conclusion of an example derivation for R_{CNR}

6.2 A proof of static semantics preservation

We prove the following Theorem:

Theorem 6.1 (R_{CNR} is strongly static semantics preserving). *For all t_1 and t_2, if $R_{CNR}(t_1, t_2)$ and $\Delta; \Gamma \vdash t_1 : T^n$, then $\Delta; \Gamma \vdash t_2 : T^n$.*

Proof. The type rules as we implemented them in section 4.2 are mutually defined, which means we also need a mutual induction principle if we want to prove Theorem 6.1 by induction on the typing derivation. We use the `Scheme` command to generate the mutual induction principle for us.

```
Scheme has_type__ind := Minimality for has_type Sort Prop
  with constructor_well_formed__ind := Minimality for constructor_well_formed Sort Prop
  with (* hidden *).
```

In order for us to use the generated induction principle for our proof, we need to define a proposition for each of the inductive datatypes that make up the implementation of the type rules. This is reflected in the type of the induction principle.

```
Check has_type__ind
       : forall
         (P : Delta -> Gamma -> Term -> Ty -> Prop)
         (P0 : Delta -> constructor -> Ty -> Prop)
         (P1 P2 : Delta -> Gamma -> list Binding -> Prop)
         (P3 : Delta -> Gamma -> Binding -> Prop),
           (* hidden *).
```

For example, we define the proposition for the main `has_type` inductive datatype as the main result that we aim to prove. This proposition does not have to include an assumption that `t1` is well-typed since well-typedness follows from the fact that we apply the induction principle to a typing derivation. The same is true for the other propositions.

```
Definition P_has_type Delta Gamma t1 T : Prop :=
  forall t2,
    CNR_Term t1 t2 ->
    Delta ,, Gamma |-+ t2 : T.
```

The proposition for the well-formedness of a single binding (`binding_well_formed`) is more interesting. In this proposition, we must make a distinction between congruence proof cases and *interesting* proof cases.

```
Definition P_binding_well_formed Delta Gamma b1 : Prop :=
  ( forall b2,
      Delta ,, Gamma |-ok_b b1 ->
      Congruence.Cong_Binding CNR_Term b1 b2 ->
      Delta ,, Gamma |-ok_b b2 /\
      binds_Delta b2 = binds_Delta b1 /\ binds_Gamma b2 = binds_Gamma b1
  ) /\ (
    forall f_b2 t T bs1Gn,
      Delta ,, Gamma |-ok_b b1 ->
      CNR_Binding b1 f_b2 ->
      map_normalise (binds_Gamma b1) bs1Gn ->
      mupdate Delta (binds_Delta b1) ,, mupdate Gamma bs1Gn |-+ t : T ->
      Delta ,, Gamma |-+ (f_b2 t) : T
  ).
```

In the case of a congruence relation, we just aim to prove that the post-translation binding is still well-formed with respect to the same kind and type contexts. Moreover, we need to prove that the term and type variable bindings produced by the pre- and post-translation bindings are equal. If they would not be equal, then we would not be able to prove that R_{CNR} has the *strong* static semantics preservation property. In particular, recall that we update the kind and type contexts with these term and type variable bindings, and different term and type variable bindings would lead to different kind and type contexts, which goes against the definition of strong static semantics preservation.

Before we discuss the interesting disjunct of the proposition that we listed above, note that we did not introduce the `map_normalise` relation before. The relation models a `List.map` function for type indices and it does exactly what its name suggests: it maps the normalisation relation over all its "arguments" to obtain a list of term variable bindings that now contain normalised types.

In the case of a desugaring relation, we aim to prove that the application of the post-translation function is well-typed with respect to the same kind and type contexts. In pursuit of this proof, we consider some well-typed term to which we apply the function, such that we can prove the well-typedness of the result. For non-recursive binding sequences, we aim to prove a similar proposition in the case of desugaring.

```
Definition P_bindings_well_formed_nonrec Delta Gamma bs1 : Prop :=
  ( (* hidden *) ) /\ (
    forall f_bs2 t T bs1Gn,
      Delta ,, Gamma |-oks_nr bs1 ->
      CNR_Bindings bs1 f_bs2 ->
      map_normalise (flatten (map binds_Gamma bs1)) bs1Gn ->
      (mupdate Delta (flatten (map binds_Delta bs1))) ,, (mupdate Gamma bs1Gn) |-+ t : T ->
      Delta ,, Gamma |-+ (fold_right apply t f_bs2) : T
  ).
```

Note that we have not seen the `flatten` function before: it makes sure to collect and reorder the term and type variable bindings of all bindings in the binding sequence such that we can update the contexts with them.

After defining propositions for the `constructor_well_formed` and `bindings_well_formed_rec` datatypes, we can apply the mutual induction principle and prove Theorem 6.1. Most of the proof cases only require `inversion` and some proof automation using `eauto` to go through. More interesting cases, such as the proof case corresponding to the **W-Term** rule from Figure 4.12, require some more work. We do not list the full proof here.

```
Theorem CNR_Term__SSP : forall Delta Gamma t1 T,
    Delta ,, Gamma |-+ t1 : T ->
    P_has_type Delta Gamma t1 T.
Proof with ( (* hidden *) ).
  apply has_type__ind with
    (P := P_has_type) (P0 := P_constructor_well_formed)
    (P1 := P_bindings_well_formed_nonrec) (P2 := P_bindings_well_formed_rec)
    (P3 := P_binding_well_formed).
  (* hidden *).
Qed.
```

□

6.3 A proof of dynamic semantics preservation

We prove the following Theorem:

Theorem 6.2 (R_{CNR} is dynamic semantics preserving)**.** *For all t_1 and t_2, if $R_{CNR}(t_1, t_2)$ and $\Delta;\Gamma \vdash t_1 : T^n$, then $\Delta;\Gamma \vdash t_1 \leq t_2 : T^n$.*

Proof. We prove Theorem 6.2 by mutual induction on the typing derivation. Again, we need to define a proposition for each of the inductive datatypes that make up the implementation of the type rules. First and foremost, the proposition for the `has_type` datatype is defined to be our main proof goal.

```
Definition P_has_type Delta Gamma e T : Prop :=
  forall t',
    CNR_Term t t' ->
    LR_logically_approximate Delta Gamma t t' T.
```

The interesting proposition that we define is for the `binding_well_formed` datatype, where we have to make a distinction between congruence proof cases and interesting proof cases.

```
1  Definition P_binding_well_formed Delta Gamma b : Prop :=
2    ( forall b',
3        Congruence.Cong_Binding CNR_Term b b' ->
4        forall Delta_t Gamma_t bsGn t t' T bs bs',
5          Delta_t = mupdate Delta (binds_Delta b) ->
6          map_normalise (binds_Gamma b) bsGn ->
7          Gamma_t = mupdate Gamma bsGn ->
8          LR_logically_approximate Delta_t Gamma_t (Let NonRec bs t) (Let NonRec bs' t') T ->
9          LR_logically_approximate Delta Gamma
10           (Let NonRec (b :: bs) t) (Let NonRec (b' :: bs') t') T
11   ) /\ (
12     forall fb',
13       CNR_Binding b fb' ->
14       forall Delta_t Gamma_t bsGn t t' T bs fbs',
15         Delta_t = mupdate Delta (binds_Delta b) ->
16         map_normalise (binds_Gamma b) bsGn ->
17         Gamma_t = mupdate Gamma bsGn ->
18         LR_logically_approximate Delta_t Gamma_t
19           (Let NonRec bs t) (fold_right apply t' fbs') T ->
20         LR_logically_approximate Delta Gamma
21           (Let NonRec (b :: bs) t) (fold_right apply t' (fb' :: fbs')) T
22   ).
```

In the case of congruence, we aim to prove that we can safely add related bindings b and b' to the binding sequences of two logically approximating, non-recursive `let`-bindings such that the new `let`-bindings are still logically approximating. We update kind and type contexts similarly to how we discussed updating them in section 6.2. In essence, we define this disjunct of the proposition as an inductive argument, where line 8 is the induction hypothesis.

In the case of desugaring, we aim to prove that we can safely desugar a binding b to a function f_b. Given some non-recursive `let`-binding that logically approximates a desugared, non-recursive `let`-binding, we can add b to the binding sequence of the `let`-binding and we can apply f_b to the desugared `let`-binding such that the results are still logically approximating. Again, we have defined this disjunct of the proposition as an inductive argument, where lines 18-19 form the induction hypothesis. The proposition for non-recursive binding sequences is similar in the case of desugaring.

```
Definition P_bindings_well_formed_nonrec Delta Gamma bs : Prop :=
  ( (* hidden *) ) /\ (
    forall fbs',
      CNR_Bindings bs fbs' ->
      forall Delta_t Gamma_t bsGn t t' T,
        Delta_t = mupdate Delta (flatten (List.map binds_Delta bs)) ->
        map_normalise (flatten (List.map binds_Gamma bs)) bsGn ->
        Gamma_t = mupdate Gamma bsGn ->
        LR_logically_approximate Delta_t Gamma_t t t' T ->
        LR_logically_approximate Delta Gamma (Let NonRec bs t) (fold_right apply t' fbs') T
  ).
```

Having defined propositions for the `constructor_well_formed` and `bindings_well_formed_rec` inductive datatypes, we can apply the mutual induction principle and prove Theorem 6.2.

```
Theorem CNR_Term__DSP : forall Delta Gamma e T,
    Delta ,, Gamma |-+ e : T ->
    P_has_type Delta Gamma e T.
Proof with ( (* hidden *) ).
  apply has_type__ind with
    (P := P_has_type)
    (P0 := P_constructor_well_formed) (P1 := P_bindings_well_formed_nonrec)
    (P2 := P_bindings_well_formed_rec) (P3 := P_binding_well_formed).
  (* hidden *).
Qed.
```

Conveniently, we can reuse the set of compatibility lemmas that we have implemented in section 5.2 to prove all the congruence cases! For example, consider the proof case that corresponds to the **T-Apply** rule from Figure 4.12, where we have to prove $\Delta;\Gamma \vdash (t_1\ t_2) \leq (t_1'\ t_2') : T_2^n$ assuming that we are proving a congruence case such that Cong $(t_1\ t_2)\ (t_1'\ t_2')$. After some proof scripting we find through inspection of Cong and the use of induction hypotheses that t_1 logically approximates t_1' and that t_2 logically approximates t_2'. This is exactly the information that we need to apply the `compatibility_Apply` lemma from section 5.2.

For the interesting cases of the proof, we must instead prove some new compatibility lemmas. These compatibility lemmas are, for the most part, similar to ones that have defined before when we proved the Fundamental Property in section 5.2. For example, consider the desugaring proof case for the **W-Term** rule from Figure 4.12. For the proof of the Fundamental Property, we proved a compatibility lemma `compatibility_TermBind`.

```
Lemma compatibility_TermBind : forall Delta Gamma stricty x Tb Tbn tb tb’ b b’ bs bs’ t t’ Tn,
    Delta |-* Tb : Kind_Base ->
    normalise Tb Tbn ->
    forall Delta_ih Gamma_ih bsGn,
      b = TermBind stricty (VarDecl x Tb) tb ->
      b’ = TermBind stricty (VarDecl x Tb) tb’ ->
      Delta_ih = mupdate Delta (binds_Delta b) ->
      map_normalise (binds_Gamma b) bsGn ->
      Gamma_ih = mupdate Gamma bsGn ->
      LR_logically_approximate Delta_ih Gamma_ih (Let NonRec bs t) (Let NonRec bs’ t’) Tn ->
      LR_logically_approximate Delta Gamma tb tb’ Tbn ->
      LR_logically_approximate Delta Gamma
        (Let NonRec (b :: bs) t) (Let NonRec (b’ :: bs’) t’) Tn.
Proof with (* hidden *). (* hidden *). Qed.
```

For our current proof, we must prove a new compatibility lemma for term bindings that are desugared. Since we have proved compatibility lemmas for term bindings, applications and lambda abstractions before, we can reuse parts of their proofs to finish the proof of the new compatibility lemma without problems.

```
Lemma compatibility_TermBind__desugar : forall Delta Gamma t t’ Tn b bs fbs’ tb tb’ x Tb Tbn,
    Delta |-* Tb : Kind_Base ->
    normalise Tb Tbn ->
    forall Delta_ih Gamma_ih bsGn,
      b = TermBind Strict (VarDecl x Tb) tb ->
      Delta_ih = mupdate Delta (binds_Delta b) ->
      map_normalise (binds_Gamma b) bsGn ->
      Gamma_ih = mupdate Gamma bsGn ->
      LR_logically_approximate Delta_ih Gamma_ih
        (Let NonRec bs t) (fold_right apply t’ fbs’) Tn ->
      LR_logically_approximate Delta Gamma tb tb’ Tbn ->
      LR_logically_approximate Delta Gamma
        (Let NonRec (b :: bs) t) (Apply (LamAbs x Tb (fold_right apply t’ fbs’)) tb’) Tn.
Proof with (* hidden *). (* hidden *). Qed.
```

The collection of old and new compatibility lemmas is then sufficient to prove Theorem 6.2.

□

Chapter 7

Conclusion

The field of formal compiler verification has seen extensive research contributions for good reasons: (i) compiler verification is a hard problem to solve since compilers are pieces of advanced software that carry out a range of non-trivial tasks, and (ii) compiler verification is crucial if we want to offer formal guarantees about the code that they produce. We argued in our introduction that compiler verification can be seen as even more vital in the context of safety-critical software such as smart contracts, since past experience has made clear how safety issues in smart contract systems can result in substantial (financial) damages. Moreover, we specifically argued that rigorous formal methods such as compiler verification can not be passed over directly when we consider security of smart contract systems.

As justified by these motivations, Krijnen et al. started the development of a certification engine, developed in the Coq Proof Assistant, for the Plutus Tx compiler[17]. The compiler, which is part of the Plutus Platform for the Cardano blockchain, compiles a subset of Haskell, using PIR as an intermediate language, to Plutus core, the latter of which is a slight superset of System F_{ω}^{μ}: System F extended with higher-order kinds and iso-recursive types. The certification engine itself falls in the category of certifying compilers, as it verifies individual runs of the Plutus Tx compiler and produces machine-checkable certificates of the compiled code's correctness.

While most of the ongoing development on the engine has concerned itself with recognising admissible behaviour of compiler passes through the use of translation relations, it remained an open problem to show that the translation relations, and per extension the compiler passes, are well-behaved. More formally, it remained an open problem to show that the semantics of terms are preserved across translation relations, which lead us to the main research question of this thesis:

Research question: *Can we formally verify that translation relations are semantics preserving?*

In chapter 3, we discussed the results of a preliminary study in which we set out to answer the research question in the context of a simply typed lambda calculus. We achieved positive results by identifying and working out three main goals on paper and in Coq: (i) constructing a formal static and dynamic semantics, (ii) defining static and dynamic semantics preservation and (iii) proving our previously defined concepts of semantics preservation for example translation relations. Most importantly, we used the logical relations proof technique, the implementation of which is based on the Normalisation chapter of the Programming Language Foundations book[30], to express and prove dynamic semantics preservation. The positive results lead us to believe that we could scale up the methods to the Plutus Tx compiler, seeing as the languages upon which the Plutus Tx compiler operates are based on a simply typed lambda calculus.

For the main study, we laid out the same goals that we set for ourselves in the preliminary study, but now in the context of the PIR and Plutus Core languages. In chapter 4, we formally defined the syntax, static semantics and dynamic semantics of the PIR and Plutus Core languages. The static semantics is designed to represent kind and type checking algorithms where we make sure that we only type check for normal types, which simplifies reasoning about typing judgements. The dynamic semantics is defined and implemented as a big-step operational semantics (evaluation relation) since such a dynamic semantics emphasises results of computations. As an added advantage, the big-step style allows us to incorporate the counting of evaluation steps into the semantics directly.

In chapter 5, we defined the concept of (strong/weak) static semantics preservation, and we defined dynamic semantics preservation by adapting and extending step-indexed logical relations[1]. We made a distinction between strong and weak static semantics preservation in order to accommodate for a wide variety of static semantics preservation theorems. Step-indexing offered a solution to problematic aspects of the PIR and Plutus Core languages that prevented us from easily scaling up the methods that we used in the preliminary study. On one hand, higher-order kinds allow for computations in types, which means we could not implement the logical relation by straightforward induction on the syntax of types. On the other hand, iso-recursive types allow for non-termination on the term-level. Step-indexing gives us both a well-foundedness measure that we can use to prove well-foundedness of the logical relation, and it gives us a way to compare terms in the presence of non-termination. After defining the logical relation, we proved the Fundamental Property, which states that the logical relation is reflexive. We prove reflexivity by defining and proving a set of compatibility lemmas that each apply to one case of the reflexivity proof.

Finally, we proved strong static semantics preservation and dynamic semantics preservation for a translation relation that characterises the desugaring of non-recursive `let`-bindings in chapter 6. Interestingly, we could reuse the compatibility lemmas that we defined in chapter 5 and parts of their proofs to prove dynamic semantics preservation. As such, the compatibility lemmas allow for a modular approach to dynamic semantics preservation proofs: we only have to prove new compatibility lemmas for non-congruence cases of the proof, and we can reuse compatibility lemmas that we have defined before.

As we accomplished the goals that we laid out, we believe that these positive results imply that we can answer the research question affirmatively:

Conclusion: *We can formally verify that translation relations are semantics preserving.*

The primary contribution of this thesis is therefore that we show that we can formally verify semantics preservation properties for translation relations. Moreover, we have given a practical implementation of step-indexed logical relations that is reusable for proofs of dynamic semantics preservation within the certification engine. We also suspect that the implementation can be adapted to work for similar Coq-implemented projects such as compilers or other verification tools.

Secondary contributions. Throughout chapters 4, 5 and 6 we rely on and adapt a number of previous works, amongst which Krijnen et al.[17], Peyton Jones et al.[15], Chapman et al.[7], Ahmed[1] and the Software Foundation books[29, 30], but we also add to and extend some works with secondary contributions:

- We implement the formal static and dynamic semantics of the PIR and Plutus Core languages in Coq. Furthermore, we define a formal dynamic semantics of some PIR- and Plutus Core-specific language constructs for which a formal dynamic semantics did not previously exist. (Section 4)
- We implement step-indexed logical relations in Coq, and we extend step-indexed logical relations to work in the context of higher-order kinds. Moreover, we implement step-indexed logical relations for functional languages that are intended for practical use. (Section 5)

7.1 Discussion

Several aspects of the thesis warrant a discussion. Aspects such as *limitations of results* and *proof completeness* have already been discussed throughout the thesis, though we find it useful to give a comprehensive overview of our remarks here. Lastly, we spend some words on our *trusted codebase*.

7.1.1 Limitations of results

We aim to only discuss the limitations of our definitions and implementations here, and not the limitations that concern themselves with proofs and their completeness. The limitations of our results, proofs and their completeness aside, can be put into roughly two categories. On one hand, we have that our definitions and implementations of the static semantics differ slightly from the compiler implementation. On the other hand, our results with respect to the dynamic semantics and the step-indexed logical relation are not always complete. We refer to the first category as category (i), and the second category as (ii).

Escaping type bindings (i). We do not allow type bindings to escape `let`-bindings, whereas the compiler allows this if the `let`-binding is top-level. For the sake of proof simplicity, it is beneficial to disallow or allow escaping altogether, so we opted not to allow escaping at all.

Typing of errors (i). Errors have type annotations that signal the type of the error. If the error would not have a type annotation, then we can not produce a principal type for the error. A sensible catch-all type would be $\forall X :: *.X$, but this type is not necessarily the most general. Nonetheless, we ignore the type annotation in our implementation since we can not define a type-preserving dynamic semantics otherwise. More specifically, the type annotations on terms do not carry the necessary information to make sure that the error has the correct type annotation after error propagation.

Incomplete evaluation rules (ii). We did not give evaluation rules for datatype bindings.

Strictness and laziness (ii). In the evaluation of term bindings, we ignore strictness flags.

Semantics of built-in functions (ii). The semantics of some more complex built-in functions can not yet be modelled in Coq. For example, the `VerifySignature` built-in function has a complex Haskell implementation that we can not reproduce in Coq, so we let this built-in function return `true` by default.

Built-in functions in the logical relation (ii). (Partially applied) built-in functions are not handled as of yet in the logical relation. A crucial difference between lambda abstractions and built-in functions is that we can reason about the body of a lambda abstraction since we reason about closed instantiations of terms. Built-in functions have no body, so instead we must somehow express in the logical relation that fully applying a built-in function to related arguments leads to related results. We have not found a satisfying solution to this yet.

Strong and weak dynamic semantics preservation(ii). As of yet, the logical relation can only be used to prove dynamic semantics preservation for translation relations that are strongly static semantics preserving.

We do not think that the limitations undermine our results. The results that we do show are sufficiently substantiated, and we have no reason to believe that resolving the abovementioned issues should invalidate our previous results. In fact, we expect that solutions to these problems will consist mostly of technical overhead, though we expect that adding built-in functions to the logical relation and implementing weak dynamic semantics preservation will require some new insights. We did not resolve these issues due to time constraints. Given more time, we are confident that we could have resolved these issues in a satisfactory manner.

7.1.2 Proof completeness

Proofs of logical approximation that we discuss in chapters 5 and 6, which are the proofs of the Fundamental Property (reflexivity) and dynamic semantics preservation for R_{CNR}, depend on a range of different lemmas and theorems. Because of time constraints, we were not able to fully prove all the lemmas and theorems on which the logical approximation proofs depend: they are `Admitted`. We take care to admit only lemmas and theorems that should logically hold. For example, we know that our different types of substitution should preserve typing and kinding, we know that our type-level language should be strongly normalising, we know that our normalisation relation should preserve kinding, and so on. In spite of that, it could still be the case that, for example, our implementation of term substitution does not preserve typing, though this would probably signal an error in the implementation of term substitution, and not an error in our judgement. Again, given more time, we are confident that we could have completed the proofs in a satisfactory manner.

7.1.3 Trusted codebase

A detail that we have purposely not mentioned throughout the thesis is that, in order for some of our proofs to go through, we need the property that (type) binders are *unique*. That is, a term or type should bind a (type) variable only once. The compiler implementation makes sure that variables are unique through a compiler pass that performs renaming. How then, do we prove dynamic semantics preservation for the translation relation that corresponds with renaming? Maybe more generally, how can we make sure that we always reason about terms for which the uniqueness property holds? One solution would be to define a function that performs renaming, such that we can apply renaming to terms after which we know that the uniqueness property holds. Another solution would be to axiomatise the uniqueness property, which is arguably justified since we could always apply a renaming function when needed. An advantage over the first solution is that axiomatisation would lead to simpler proofs. One way or the other, both solutions may require us to add to our *trusted codebase*: code that we do not or can not verify, but instead trust to function correctly.

Adding to the trusted codebase is always a trade-off. The more we add to our trusted codebase, the less formal our guarantees are. Currently, the trusted codebase of this thesis consists of the axiomatisation of the uniqueness property, and in addition the `coqc` compiler for Coq. Nevertheless, we can not really prevent the fact that we we must trust the `coqc` compiler. Lastly, we could also view limitations of our results and proof completeness as some form of trusted codebase. After all, we trust that both do not greatly influence the validity of the results.

7.2 Future work

Solving the issues with respect to limitations of results, proof completeness and the trusted codebase are one item of future work that is necessary. Naturally, as the correctness of compiler passes depends on the correctness of the translation relations that characterise them, it is also a justified goal to prove semantics preservation for more translation relations. Lastly, one interesting piece of future work would be to prove soundness and completeness of our logical relation with respect to a definition of contextual equivalence. Such soundness and completeness results will strengthen the formal guarantees that proofs of dynamic semantics preservation give.

Appendix A

Preliminary study: `Notation` commands for abstract syntax

```
Declare Custom Entry stlc.
Declare Custom Entry stlc_ty.

Notation "<{ e }>" := e (e custom stlc at level 99).
Notation "<{{ e }}>" := e (e custom stlc_ty at level 99).
Notation "( x )" := x (in custom stlc, x at level 99).
Notation "( x )" := x (in custom stlc_ty, x at level 99).
Notation "x" := x (in custom stlc at level 0, x constr at level 0).
Notation "x" := x (in custom stlc_ty at level 0, x constr at level 0).
Notation "S -> T" := (ty_arrow S T) (in custom stlc_ty at level 50, right associativity).
Notation "x y" := (tm_app x y) (in custom stlc at level 1, left associativity).
Notation "\ T , t" := (tm_abs T t) (in custom stlc at level 90,
                                    T custom stlc_ty at level 99,
                                    t custom stlc at level 99,
                                    left associativity).
Coercion tm_var : ref >-> tm.
Notation "{ x }" := x (in custom stlc at level 1, x constr).
Notation "{ x }" := x (in custom stlc_ty at level 1, x constr).

Notation "'Bool'" := ty_bool (in custom stlc_ty at level 0).
Notation "'if' x 'then' y 'else' z" :=
  (tm_if x y z) (in custom stlc at level 89,
                 x custom stlc at level 99,
                 y custom stlc at level 99,
                 z custom stlc at level 99,
                 left associativity).
Notation "'true'" := true (at level 1).
Notation "'true'" := tm_true (in custom stlc at level 0).
Notation "'false'" := false (at level 1).
Notation "'false'" := tm_false (in custom stlc at level 0).

Notation "'let' t1 ':' T1 'in' t2" := (tm_let t1 T1 t2) (in custom stlc at level 0,
                                                          T1 custom stlc_ty at level 99).

Notation "'Unit'" := (ty_unit) (in custom stlc_ty at level 0).
Notation "'unit'" := tm_unit (in custom stlc at level 0).
```

Bibliography

Main Sources

[1] Amal J. Ahmed. “Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types”. In: *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings.* Ed. by Peter Sestoft. Vol. 3924. Lecture Notes in Computer Science. Springer, 2006, pp. 69–83. DOI: 10.1007/11693024_6. URL: https://doi.org/10.1007/11693024%5C_6.

[2] Andrew W. Appel and David A. McAllester. “An indexed model of recursive types for foundational proof-carrying code”. In: *ACM Trans. Program. Lang. Syst.* 23.5 (2001), pp. 657–683. DOI: 10.1145/504709.504712. URL: https://doi.org/10.1145/504709.504712.

[3] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. “A Survey of Attacks on Ethereum Smart Contracts (SoK)”. In: *Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings.* Ed. by Matteo Maffei and Mark Ryan. Vol. 10204. Lecture Notes in Computer Science. Springer, 2017, pp. 164–186. DOI: 10.1007/978-3-662-54455-6_8. URL: https://doi.org/10.1007/978-3-662-54455-6%5C_8.

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq’Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN: 978-3-642-05880-6. DOI: 10.1007/978-3-662-07964-5. URL: https://doi.org/10.1007/978-3-662-07964-5.

[7] James Chapman et al. “System F in Agda, for Fun and Profit”. In: *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings.* Ed. by Graham Hutton. Vol. 11825. Lecture Notes in Computer Science. Springer, 2019, pp. 255–297. DOI: 10.1007/978-3-030-33636-3_10. URL: https://doi.org/10.1007/978-3-030-33636-3%5C_10.

[8] Adam Chlipala. “A verified compiler for an impure functional language”. In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010.* Ed. by Manuel V. Hermenegildo and Jens Palsberg. ACM, 2010, pp. 93–106. DOI: 10.1145/1706299.1706312. URL: https://doi.org/10.1145/1706299.1706312.

[9] Christopher Colby et al. “A certifying compiler for Java”. In: *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000.* Ed. by Monica S. Lam. ACM, 2000, pp. 95–107. DOI: 10.1145/349299.349315. URL: https://doi.org/10.1145/349299.349315.

[11] Chris Cummins et al. “Compiler fuzzing through deep learning”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018.* Ed. by Frank Tip and Eric Bodden. ACM, 2018, pp. 95–105. DOI: 10.1145/3213846.3213848. URL: https://doi.org/10.1145/3213846.3213848.

[12] Maulik A. Dave. “Compiler verification: a bibliography”. In: *ACM SIGSOFT Softw. Eng. Notes* 28.6 (2003), p. 2. DOI: 10.1145/966221.966235. URL: https://doi.org/10.1145/966221.966235.

[13] Herman Geuvers. “Proof assistants: History, ideas and future”. In: *Sadhana* 34.1 (2009), pp. 3–25.

[14] Limin Jia et al. “AURA: a programming language for authorization and audit”. In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008.* Ed. by James Hook and Peter Thiemann. ACM, 2008, pp. 27–38. DOI: 10.1145/1411204.1411212. URL: https://doi.org/10.1145/1411204.1411212.

[15] Michael Peyton Jones et al. “Unraveling Recursion: Compiling an IR with Recursion to System F”. In: *Mathematics of Program Construction - 13th International Conference, MPC 2019, Porto, Portugal, October 7-9, 2019, Proceedings.* Ed. by Graham Hutton. Vol. 11825. Lecture Notes in Computer Science. Springer, 2019, pp. 414–443. DOI: 10.1007/978-3-030-33636-3_15. URL: https://doi.org/10.1007/978-3-030-33636-3%5C_15.

[16] Guy L. Steele Jr. "It's Time for a New Old Language". In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017.* Ed. by Vivek Sarkar and Lawrence Rauchwerger. ACM, 2017, p. 1. DOI: `10.1145/3018743.3018773`. URL: `https://doi.org/10.1145/3018743.3018773`.

[17] Jacco Krijnen et al. "Translation Certification for Smart Contracts". ICFP 2021, Workshop on Type-Driven Development. 2021.

[18] Ramana Kumar et al. "CakeML: a verified implementation of ML". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014.* Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 179–192. DOI: `10.1145/2535838.2535841`. URL: `https://doi.org/10.1145/2535838.2535841`.

[19] Xavier Leroy. "A Formally Verified Compiler Back-end". In: *J. Autom. Reason.* 43.4 (2009), pp. 363–446. DOI: `10.1007/s10817-009-9155-4`. URL: `https://doi.org/10.1007/s10817-009-9155-4`.

[20] Xavier Leroy. "Formal verification of a realistic compiler". In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: `10.1145/1538788.1538814`. URL: `https://doi.org/10.1145/1538788.1538814`.

[21] Xavier Leroy et al. "CompCert-a formally verified optimizing compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress.* 2016.

[22] Loi Luu et al. "Making Smart Contracts Smarter". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016.* Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 254–269. DOI: `10.1145/2976749.2978309`. URL: `https://doi.org/10.1145/2976749.2978309`.

[23] Muhammad Izhar Mehar et al. "Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack". In: *J. Cases Inf. Technol.* 21.1 (2019), pp. 19–32. DOI: `10.4018/JCIT.2019010102`. URL: `https://doi.org/10.4018/JCIT.2019010102`.

[24] George C. Necula. "Proof-Carrying Code". In: *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997.* Ed. by Peter Lee, Fritz Henglein, and Neil D. Jones. ACM Press, 1997, pp. 106–119. DOI: `10.1145/263699.263712`. URL: `https://doi.org/10.1145/263699.263712`.

[25] George C. Necula. "Translation validation for an optimizing compiler". In: *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000.* Ed. by Monica S. Lam. ACM, 2000, pp. 83–94. DOI: `10.1145/349299.349314`. URL: `https://doi.org/10.1145/349299.349314`.

[26] George C. Necula and Peter Lee. "The Design and Implementation of a Certifying Compiler". In: *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998.* Ed. by Jack W. Davidson, Keith D. Cooper, and A. Michael Berman. ACM, 1998, pp. 333–344. DOI: `10.1145/277650.277752`. URL: `https://doi.org/10.1145/277650.277752`.

[27] James Painter. "Correctness of a compiler for arithmetic expressions". In: *Proceedings of a Symposium in Applied Mathematics.* Vol. 19. 1967, pp. 33–41.

[28] Daejun Park, Yi Zhang, and Grigore Rosu. "End-to-End Formal Verification of Ethereum 2.0 Deposit Smart Contract". In: *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I.* Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 12224. Lecture Notes in Computer Science. Springer, 2020, pp. 151–164. DOI: `10.1007/978-3-030-53288-8\_8`. URL: `https://doi.org/10.1007/978-3-030-53288-8%5C_8`.

[29] Benjamin C. Pierce et al. *Logical Foundations.* Vol. 1. Software Foundations. Version 6.0. Electronic textbook, 2021. URL: `http://softwarefoundations.cis.upenn.edu`.

[30] Benjamin C. Pierce et al. *Programming Language Foundations.* Vol. 2. Software Foundations. Version 6.0. Electronic textbook, 2021. URL: `http://softwarefoundations.cis.upenn.edu`.

[31] Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation Validation". In: *Tools and Algorithms for Construction and Analysis of Systems, 4th International Conference, TACAS '98, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings.* Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Springer, 1998, pp. 151–166. DOI: `10.1007/BFb0054170`. URL: `https://doi.org/10.1007/BFb0054170`.

[33] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "Translation validation for a verified OS kernel". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013.* Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 471–482. DOI: `10.1145/2491956.2462183`. URL: `https://doi.org/10.1145/2491956.2462183`.

[34] Matthieu Sozeau. “Equations: A Dependent Pattern-Matching Compiler”. In: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings.* Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 419–434. DOI: `10.1007/978-3-642-14052-5\_29`. URL: `https://doi.org/10.1007/978-3-642-14052-5%5C_29`.

[35] Matthieu Sozeau and Cyprien Mangin. “Equations reloaded: high-level dependently-typed functional programming and proving in Coq”. In: *Proc. ACM Program. Lang.* 3.ICFP (2019), 86:1–86:29. DOI: `10.1145/3341690`. URL: `https://doi.org/10.1145/3341690`.

[36] Martin Strecker. “Formal Verification of a Java Compiler in Isabelle”. In: *Automated Deduction - CADE-18, 18th International Conference on Automated Deduction, Copenhagen, Denmark, July 27-30, 2002, Proceedings.* Ed. by Andrei Voronkov. Vol. 2392. Lecture Notes in Computer Science. Springer, 2002, pp. 63–77. DOI: `10.1007/3-540-45620-1\_5`. URL: `https://doi.org/10.1007/3-540-45620-1%5C_5`.

[37] Wouter Swierstra. “Heterogeneous binary random-access lists”. In: *J. Funct. Program.* 30 (2020), e10. DOI: `10.1017/S0956796820000064`. URL: `https://doi.org/10.1017/S0956796820000064`.

[38] William W. Tait. “Intensional Interpretations of Functionals of Finite Type I”. In: *J. Symb. Log.* 32.2 (1967), pp. 198–212. DOI: `10.2307/2271658`. URL: `https://doi.org/10.2307/2271658`.

[41] Philip Wadler. “Theorems for Free!” In: *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989.* Ed. by Joseph E. Stoy. ACM, 1989, pp. 347–359. DOI: `10.1145/99370.99404`. URL: `https://doi.org/10.1145/99370.99404`.

Online Sources

[4] Augur. *Serpent Compiler Vulnerability, REP & Solidity Migration.* Visited: 13th of October, 2021. 2017. URL: `https://medium.com/@AugurProject/serpent-compiler-vulnerability-rep-solidity-migration-5d91e4ae90dd`.

[6] Lorenz Bredenbach et al. *An In-Depth Look at the Parity Multisig Bug.* Visited: 13th of October, 2021. 2017. URL: `https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`.

[10] Martin Constantino-Bodin. *List of Coq PL Projects.* Visited: 13th of October, 2021. 2019. URL: `https://github.com/coq/coq/wiki/List%20of%20Coq%20PL%20Projects`.

[32] OpenZeppelin Security. *Serpent Compiler Audit.* Visited: 13th of October, 2021. 2017. URL: `https://blog.openzeppelin.com/serpent-compiler-audit-3095d1257929/`.

[39] Parity Technologies. *A Postmortem on the Parity Multi-Sig Library Self-Destruct.* Visited: 13th of October, 2021. 2017. URL: `https://www.parity.io/blog/a-postmortem-on-the-parity-multi-sig-library-self-destruct/`.

[40] Parity Technologies. *The Multi-sig Hack: A Postmortem.* Visited: 13th of October, 2021. 2017. URL: `https://www.parity.io/blog/the-multi-sig-hack-a-postmortem`.

List of Theorems

List of Figures