Adventures in Formalisation:

Financial Contracts, Modules, and Two-Level Type Theory

Danil Annenkov
DIKU, Department of Computer Science
University of Copenhagen, Denmark

April 16, 2018

PhD Thesis
This thesis has been submitted to the PhD School of the Faculty of Science, University of Copenhagen, Denmark

Abstract

Over the last few decades, software has become essential for the proper functioning of systems in the modern world. Formal verification techniques are slowly being adopted in various industrial application areas, and there is a big demand for research in the theory and practice of formal techniques to achieve a wider acceptance of tools for verification.

We present three projects concerned with applications of certified programming techniques and proof assistants in the area of programming language theory and mathematics.

The first project is about a certified compilation technique for a domain-specific programming language for financial contracts (the CL language). The code in CL is translated into a simple expression language well-suited for integration with software components implementing Monte Carlo simulation techniques (pricing engines). The compilation procedure is accompanied with formal proofs of correctness carried out in the Coq proof assistant. Moreover, we develop techniques for capturing the dynamic behaviour of contracts with the passage of time. These techniques potentially allow for efficient integration of contract specifications with high-performance pricing engines running on GPGPU hardware.

The second project presents a number of techniques that allow for formal reasoning with nested and mutually inductive structures built up from finite maps and sets (also called semantic objects), and at the same time allow for working with binding structures over sets of variables. The techniques, which build on the theory of nominal sets combined with the ability to work with multiple isomorphic representations of finite maps, make it possible to give a formal treatment, in Coq, of a higher-order module system, including the ability to eliminate entirely, at compile time, abstraction barriers introduced by the module system. The development is based on earlier work on static interpretation of modules and provides the foundation for a higher-order module language for Futhark, an optimising compiler targeting data-parallel architectures, such as GPGPUs.

The third project is related to homotopy type theory (HoTT), a new branch of mathematics based on a fascinating idea connecting type theory and homotopy theory. HoTT provides us with a new foundation for mathematics allowing for developing machine-checkable proofs in various areas of computer science and mathematics. Along with Vladimir Voevodsky's univalence axiom, HoTT offers a formal treatment of the informal mathematical principle: equivalent structures can be identified. However, in some cases, the notion of weak equality available in HoTT leads to the "infinite coherence" problem when defining internally certain structures (such as a type of n-restricted semi-simplicial types, inverse diagrams and so on). We explain the basic idea of *two-level type theory*, a version of Martin-Löf type theory with two equality types: the first acts as the usual equality of homotopy type theory, while the second allows us to reason about strict equality. In this system, we can formalise results of partially meta-theoretic nature. We develop and explore in details how two-level type theory can be implemented in a proof assistant, providing a prototype implementation in the proof assistant `Lean`. We demonstrate an application of two-level type theory by developing some results from the theory of inverse diagrams using our Lean implementation.

Resumé

Denne afhandling består af tre dele og omhandler teknikker til udvikling af certificeret software samt anvendelse af bevisassistenter indenfor områder som programmeringssprogsteori og matematik.

Den første del omhandler en certificeret oversættelsesteknik for et domainespecifikt programmeringssprog til finansielle kontrakter (sproget CL). Kode i CL oversættes til et simpelt udtrykssprog, som er velegnet til integration med softwarekomponenter, der implementerer Monte-Carlo simuleringsteknikker (prisberegningssoftware). Oversættelsesproceduren er akkompagneret af et formelt korrekthedsbevis, der er etableret ved brug af bevisassistenten Coq.

Den anden del omhandler en række teknikker, der tillader formel ræsonnement med nestede og gensidigt induktive strukturer bygget op af endelige afbildninger og mængder (også kaldet semantiske objekter). Teknikkerne, som bygger på teorien om nominelle mængder kombineret med muligheden for at arbejde med multible isomorfe repræsentationer af endelige afbildninger, gør det muligt at give en formel behandling, i Coq, af et højere-ordens modulsystem. Behandlingen understøtter muligheden for at eliminere alle modulkonstruktioner og abstraktionsbarrierer på oversættelsestidspunktet. Teknikken baserer sig på tidligere arbejde indenfor statisk fortolkning af moduler og giver et fundament for et højere-ordens modulsprog for Futhark, en optimerende oversætter målrettet data-parallelle arkitekturer som GPGPUer.

Den tredie del omhandler en implementation af to-niveau typeteori, en version af Martin-Löfs typeteori indeholdende to lighedstyper. Den første fungerer som det sædvanlige lighedsbegreb fra homotopy typeteori, mens den anden tillader ræssonnementer omkring stringent lighed. I dette system er det muligt at formalisere resultater af delvist metateoretisk natur. Det undersøges i detaljer hvordan to-niveau typeteori kan implementeres i en bevisassistent og der udvikles en prototypeimplementation i bevisassistenten Lean. Ydermere demonstreres anvendelsen af to-niveau typeteori ved udvikling af nogle resultater (i den udviklede Lean-implementation) indenfor teorien om inverse diagrammer.

Contents

Preface

This dissertation has been submitted to the PhD School of Science, Faculty of Science, University of Copenhagen, in partial fulfilment of the degree of PhD at Department of Computer Science (DIKU).

The main content of the dissertation consists of three chapters, an introduction, and a conclusion. Some results of Chapter 2 where presented at the Nordic Workshop on Programming Theory 2016 (NWPT'16) by the author. The full source code of the formalisation presented in Chapter 2 is available online: `https://github.com/annenkov/contracts`.

Chapter 3 presents the author's contribution to ongoing work on a module system development and formalisation in collaboration with Martin Elsman, Cosmin Oancea, and Troels Henriksen at the HIPERFIT Research Center, DIKU, University of Copenhagen. The source code of the implementation of nominal sets in Coq (using type classes instead of modules) and the proof of normalisation from Section 3.2 are available online: `https://github.com/annenkov/stlcnorm`.

Chapter 4 presents the author's contribution to the work submitted for a publication with Paolo Capriotti and Nicolai Kraus, University Of Nottingham (publication preprint [ACK17]). The results of this work were presented by the author at the workshop on Homotopy Type Theory/Univalent Foundations(co-located with FSCD 2017). The full source code of the formalisation presented in Chapter 4 is available online: `https://github.com/annenkov/two-level`.

Acknowledgments

I would like to express my sincere gratitude to the people who made my PhD studies an excellent and exciting experience.

To my supervisor, Martin Elsman, for introducing me to the programming language research, for guiding and supporting me in pursuing my ideas, and for making me a better researcher.

To Fritz Henglein, for his hospitality and for the HIPERFIT Research Center allowing me to work on exiting problems in the area of finance.

To Patrick Bahr, for his beautiful work on contract formalisation in Coq, which provided me with the inspiration to work in the area of certified programming.

To Omri Ross, for sharing his experience in the area of finance and exploring the applications of the contract language to real-world problems.

To Andrzej Filinski, for his Semantics and Types course.

To my colleagues at the DIKU APL section, for the excellent research atmosphere and for interesting discussions over lunch.

To Thorsen Altenkirch, for inviting me to the FP Lab at University of Nottingham. To Paolo Capriotti and Nicolai Kraus, for all your patience explaining me category theory and inspiring me to study mathematics and pursue research in homotopy type theory. And to all the members of the FP Lab for the pleasant atmosphere and pubs every Friday.

I am so grateful to my wife Anna and to my daughter Arina, for supporting my crazy idea of pursuing PhD studies in Denmark and for being with me during this wonderful adventure. I am so happy that I have you, girls!

Chapter 1

Introduction

Over the last few decades, software has become essential for the proper functioning of systems in the modern world. Some of these systems are safety-critical, such as embedded systems in avionics, or nuclear power plants. But not only in these areas do software correctness play such a prominent role. For example, in the financial sector, software systems are responsible for executing financial transactions and managing assets by means of *smart contracts* on distributed ledgers [Woo15]. Formal verification techniques are slowly being adopted in various industrial application areas, including the area of finance and financial algorithms [PI17]. There is a big demand for research in the theoretical foundations for and practical aspects of formal techniques aimed at achieving a wider acceptance of tools for verification.

In general, our every day life relies more and more on complex software systems. Moreover, the development of complex software systems is a very costly process and discovering errors at the deployment stage may cause a significant increase in the overall cost of a system. For the last decade, software verification techniques have become available for the wider use, due to advances both in theories and in tools for formal verification. There are various approaches to formal software verification. Here we will focus on a particular direction based on various flavors of type theory and tools implementing them. These tools are called interactive theorem provers, or proof assistants. A number of large-scale verification projects use proof assistants, and we will mention some of them.

The CompCert project [Ler06] is one of the large-scale verification efforts for real-world software. It is a verified compiler for a significant subset of the C programming language carried out in the Coq proof assistant. The C programming language is widely used for development of numerous applications including critical systems. CompCert is used as a part of a verified toolchain in a number of projects.

Another example of this kind is the JSCert project. JSCert is a specification of ECMAScript 5 (JavaScript) in Coq. JavaScript is widely used in web development to write complicated applications running in browsers. Many modern web applications have large code bases in JavaScript. It is also a well known fact that JavaScript is a language with many pitfalls, which is why "The JSCert project aims to *really* understand JavaScript". Moreover, the project features an interpreter in OCaml obtained using Coq's code extraction facilities.

Apart from the areas related to software verification and programming lan-

guage semantics mechanisation, in mathematics, one often wants proofs to be verified by some automatic procedure in order to ensure correctness. Mathematical proofs can be very complicated and may require the consideration of a large number of cases. Examples of large-scale developments in this area include proofs of the four-color theorem [Gon08] and the Feit–Thompson theorem [GAA+13].

Moreover, in such abstract areas of mathematics as homotopy theory, for a long time, it has been almost impossible to use proof assistants to carry out proofs. But with recent development of homotopy type theory [Uni13] (HoTT), it has become possible to carry out proofs in homotopy theory [LS13] and many other areas of mathematics in the language of type theory. This, in turn, allows for developing formalisations in proof assistants. HoTT offers a new foundation of mathematics, where types (or spaces, in the homotopical interpretation) become the basic objects for developing mathematics. The *Unimath* project [VAG+] takes this approach and aims at implementing a large body of mathematics in the Coq proof assistant. Moreover, from the dependently-typed programming perspective, HoTT offers a generic programming technique, allowing to change between different isomorphic representations of the same abstract data structure.

1.1 Type Theory and the Curry-Howard Correspondence

Type theory originates from Bertrand Russel's approach to avoid paradoxes in set theory. Since that time, type theory has been developed by many scientists including Alonzo Church, Haskell Curry, William Howard, Stephen Kleene, Kurt Gödel, Nicolaas de Bruijn, Per Martin-Löf, and others. In the form of *dependent* type theory, it became a foundation for various proof assistants.

The central notion in type theory is a *typing judgment*. That is, a term a has a type A:

$$a : A$$

Notice the similarity with the set-theoretic proposition $a \in A$. The important difference is that in type theory, each term comes with the type and internally in type theory we cannot ask if some term has type A or B. Essentially this is how programmers in statically typed programming languages think about programs and data types. Functional programming languages have especially strong connection to type theory, since theoretical foundations for such languages are variations of the lambda-calculi.

An important step in the development of type theory was the discovery of the connection between logic and type theory, which is now known as the Curry-Howard correspondence [Cur34, How80] (for the details of the discovery and the development of the Curry-Howard correspondence, see [Wad15]). This correspondence is also known in the literature as propositions-as-types, formulae-as-types, and the Curry-Howard isomorphism.

Roughly speaking, the idea of this correspondence is that type theory corresponds to *intuitionistic* logic, propositions correspond to types, and proofs correspond to terms (or programs). The summary of the correspondence is given in Table 1.1.

Logic	**Type Theory**
Proposition A	Type A
Proof of a proposition A	Term (program) $a : A$
Proof normalisation	Program execution
True	Unit
False	Empty type
Conjunction $A \wedge B$	Product type $A \times B$
Disjunction $A \vee B$	Coproduct type $A + B$
Implication $A \Rightarrow B$	Function type $A \to B$
Universal quantification $\forall x \in A, B(x)$	Dependent product $\Pi(x : A).B(x)$
Existential quantification $\exists x \in A, B(x)$	Dependent sum $\Sigma(x : A).B(x)$

Table 1.1: Propositions-as-types.

This correspondence was extended further: Joachim Lambek showed the correspondence between the lambda-calculus and Cartesian closed categories [Lam80].

The work of Per Martin-Löf [ML84] was another important step in the development of type theory. A number of modern proof assistants implement some variation of Martin-Löf type theory. Recent research in type theory have lead to the discovery of another deep connection: the connection between type theory and homotopy theory. Homotopy type theory [Uni13] establishes a correspondence between types and spaces (more precisely, ∞-groupoids), terms and points, identity types and paths in a space. Moreover, homotopy type theory refines the correspondence outlined in Table 1.1: propositions correspond not to any types, but to certain types that have at most one inhabitant. These types in the context of homotopy type theory are called *hProps*.

According to the Curry-Howard correspondence, finding a proof of some theorem is the same as finding a term of the given type, i.e. *inhabiting* the type. Following this idea, the process of proving a theorem corresponds to the process of writing a program that is accepted be the type-checker. This proving-as-programming idea have lead to software tools supporting this paradigm, namely, proof assistants.

1.2 Proof Assistants and Certified Programming

Proof assistants, or interactive theorem provers are tools that allow for stating and proving theorems by interacting with users. That is, users write proofs in a specialised language and the tool verifies correctness of these proofs. Proof assistants often offer some degree of proof automation by implementing decision and semi-decision procedures, or interacting with automated theorem provers (SAT and SMT solvers). Some proof assistants allow for writing user-defined automation scripts, or write extensions using a plug-in system. In the

present work we will focus on application of proof assistant based on dependent type theory (Coq [BC10], Agda [Nor07], Lean [dMKA+15]), although there are other tools based on different foundations, such as variations of set theory and higher-order logic (Mizar [GKN10], Isabelle/HOL [NWP02]), and meta-logical frameworks (Twelf [PS99]).

That is, in this thesis we will explore and discuss how dependent type theory (and its implementation in a proof assistant) can be used in particular cases. We will explore how to use the expressivity of dependent types to encode invariants of structures used in formalisations in such a way that it simplifies development of the formalisation.

With connection to proof assistant technology, by *certified programming* following [Chl13] we mean a process that produces a program along with a witness of its correctness with respect to the specification. The benefit of using proof assistants based on type theory is that programs and proofs live in the same realm. That is, one can write (functional) programs and reason about their properties using the same language. Moreover, some proof assistants allow for the extraction of computational parts of an implementation into some (usually functional) programming language.

Next, we briefly describe some proof assistants implementing dependent type theory, outlining their main features.

1.2.1 Coq

The theoretical foundation of the Coq proof assistant is the calculus of constructions [CH88] extended with inductive definitions leading to the calculus of inductive constructions [CP90, BC10]. The type theory of Coq distinguishes between two kinds of types: `Prop` and `Set`. The type of propositions `Prop` is used to encode properties that will be erased during program extraction, while `Set` is used for programs containing computational content. Moreover, `Prop` is impredicative, which means that statements quantifying over `Prop` still belong to `Prop`. Impredicativity of `Prop` makes working with logical connectives more convenient, but to maintain consistency, the elimination principle for propositions only allows the result of elimination to be in `Prop`.

Coq also features a hierarchy of type universes $\texttt{Type}_i$ for $i \in \mathbb{N}, i \geq 1$. Types `Prop` and `Set` belong to $\texttt{Type}_1$, and $\texttt{Type}_i$:$\texttt{Type}_{i+1}$. Most of the time users do not have to be explicit about universe levels; universe constraints are handled by the system automatically. Recent versions of Coq support universe polymorphism.

Coq features the following languages:

- the *Gallina* language, which is essentially a dependently typed programming language (also includes the language of commands, called The Vernacular);
- the *Ltac* language, allowing for writing *tactics* for proof automation.

The tactic language is often used to build complicated proof terms and to implement certain proof search strategies. While Gallina programs are always terminating, since consistency of the underlying logic depends on this property, tactics written in the Ltac language may fail to terminate without affecting the consistency. The standard library of Coq offers various useful primitive tactics,

along with proof searching procedures `auto` and `eauto`, which use a user-defined database of lemmas ("hints") when trying to solve a goal. The library also contains several decision procedures, such as `omega` for the Presburger arithmetic and `tauto` for the intuitionistic propositional calculus.

Coq supports type classes, which are useful for operation overloading and for proof automation through the resolution mechanism.

It is possible to obtain an implementation in OCaml, Haskell or Scheme from the Coq formalisation through the code extraction mechanism, provided that the development follows certain criteria.

1.2.2 Lean

We give a brief outline of features of the Lean proof assistant version 2 [dMKA+15], since this version has been used in this thesis.[1] Lean 2 has two different modes:

- The "strict" mode, based on a similar theoretical foundation as Coq: the calculus of inductive constructions with impredicative `Prop` and definitional proof irrelevance;
- the "HoTT" mode, supporting homotopy type theory (without impredicative or proof irrelevant universes), including some higher-inductive types.

Lean features the powerful elaboration mechanism allowing to infer universe levels, implicit arguments (including type class instances), supports notation overloading, and so on. Type classes allow for some proofs to be automated by the elaboration mechanism. One of the main motivations behind Lean is to bridge the gap between automated and interactive theorem provers by pushing automatic inference as far as possible.

Proofs in Lean can be written in *term* mode, which is basically syntactic sugar for proof terms in Lean's functional language to make proofs more readable. Alternatively, one can use tactics, similarly to Coq. Although, unlike in Coq, one can switch to the "tactic" mode in any place of the definition by using `begin ... end` and filling-in the proof using tactics.

1.2.3 Agda

Agda is a dependently typed programming language implementing a predicative extension of Martin-Löf type theory [ML84]. It does not have a Prop-Set distinction as Coq.

Agda has a hierarchy of universes and supports universe polymorphism, but one has to be explicit about universe levels.

In comparison with Coq, Agda has more experimental features (like inductive-recursive and inductive-inductive definitions), and possibility to turn off some checks (like termination or strict positivity of inductive definitions), making a system possibly inconsistent, but more suitable for experimentation.

Working in Adga resembles a programming activity in a functional language. Agda supports powerful dependent pattern-matching constructs allowing one to write proofs as programs directly. Similarly to type classes in Coq and Lean, Agda supports instance arguments that can be inferred using the instance resolution mechanism. There is no build-in support for tactics in Agda,

[1] The Lean 2 repository can be found at `https://github.com/leanprover/lean2`

although there are some developments, supporting mechanisms similar to Coq's `auto` [KS15].

1.3 Thesis

The thesis considers three applications of proof assistant technology.

The semantics of domain-specific languages is an important and, at the same time, a realistic target for the application of formalisation and verification techniques. Financial contract specifications are often used in a specialised form in software components performing simulations to estimate the possible price of a contract. These software components are called pricing engines, which are optimised for performing simulations efficiently. The first question we consider is the following:

- Is it possible to develop a certified implementation of a compilation technique for the domain-specific financial contract specification language in the style of [BBE15], allowing for efficient interaction with the pricing engine?

Module systems provide a powerful abstraction mechanism allowing for writing generic highly parameterised code. For some application domains it is important to have static guarantees that module abstractions introduce no overhead. Formalisation of module systems is hard, and it has turned out to be essential to develop a number of techniques allowing for development of a module system formalisation in the Coq proof assistant. Thus, the second question is the following:

- Is it possible to develop a formalisation in the Coq proof assistant of a higher-order module system for the data-parallel array language Futhark [HSE+17] in the style of [Els99], aiming to keep it as close as possible to a pen-and-paper formalisation and to the implementation in the Futhark compiler?

The third project is related to the internalisation of partially meta-theoretical results in two-level type theory (homotopy type theory extended with strict equality). Homotopy type theory is young and a developing field with a number of open problems. For instance, the notion of weak equality available in HoTT could lead to the "infinite coherence" problem when defining internally certain structures (such as a type of n-restricted semi-simplicial types, inverse diagrams, and so on). Two-level type theory allows for approaching this problem. The third question we consider is the following:

- How can one leverage existing proof assistants to implement two-level type theory, and is it possible to use such an implementation for the development of formalisations of partially meta-theoretical results?

1.4 Contributions

We present three projects concerned with the applications of certified programming techniques and proof assistants in the area of programming language theory and in the area of mathematics.

The contributions on certified compilation of financial contracts are as follows:

- We present an extension of a domain-specific language for financial contracts developed by authors of [BBE15]. The extension features contract templates, or *instruments*. We focus on the extension allowing for parameterisation of contracts with respect to temporal parameters.
- We develop a payoff intermediate language inspired by traditional payoff languages and well-suited for the integration with Monte Carlo simulation techniques.
- We use the Coq proof assistant to develop a certified compilation procedure of contract templates into a parameterised payoff intermediate language.
- We further parameterise the compiled payoff expressions with the notion of "current time" allowing for capturing the evolution of contracts with the passage of time.
- We develop the proof of an extended soundness theorem in the Coq proof assistant. The theorem establishes a correspondence between the time-parameterised compilation scheme and the contract reduction semantics.
- We argue how the parametric payoff code allows for better performance due to avoiding recompilation with the change of parameters.

The contributions on the formalisation of a higher-order module system for the Futhark language are as follows:

- We develop a formalisation of the static interpretation of a module system in the style of [Els99] in the Coq proof assistant. This is one of the first developments in this style in Coq.
- For implementing the core concept of semantic objects we develop a technique that allows for using isomorphic representations of components of semantic objects with low proof obligation overhead. We use this technique to overcome limitations of the conservative strict positivity check in Coq.
- To deal with binding in the context of semantic objects, we apply nominal techniques. We develop a small library defining nominal sets in a generalised setting allowing for sets of variables to be bound at once. We use the developed library to define α-equivalence of semantic objects.

The contributions on formalisation of two-level type theory are as follows:

- We develop a technique allowing for two-level type theory to be implemented in existing proof assistants.
- We implement two-level type theory in the Lean proof assistant using the developed technique.
- We demonstrate how the fibrant fragment of two-level type theory can be used to develop proofs in a similar way as in homotopy type theory.

- As an application of the implemented type theory, we internalise some results on the theory of inverse diagrams in our Lean development.

1.5 Structure of the Dissertation

Following the outline of the thesis contributions, the main content of the thesis is split into three chapters.

- Chapter 2 describes our work on certified compilation of financial contracts, including a formal semantics, compilation soundness theorems, and a description of our Coq formalisation.
- Chapter 3 describes our formalisation of a higher-order module system in Coq, focusing on details of the implementation and the particular techniques applied.
- Chapter 4 discusses the motivation for two-level type theory and describes our approach to implementation, exemplified by a development in the Lean proof assistant.

Chapter 2

Certified Compilation of Financial Contracts

2.1 Background and Motivation

New technologies are emerging that have potential for seriously disrupting the financial sector. In particular, blockchain technologies, such as Bitcoins [Nak08] and the Ethereum Smart Contract peer-to-peer platform [Woo15], have entered the realm of the global financial market and it becomes essential to ask to which degree users can trust that the underlying implementations are really behaving according to the specified properties. Unfortunately, the answers are not clear and errors may result in irreversible high-impact events.

Contract description languages and payoff languages are used in large scale financial applications [Lex, Sim09], although formalisation of such languages in proof assistants and certified compilation schemes are less explored.

The work presented here builds on previous work on specifying financial contracts [AEH+06, AVR95, FSNB09, HKZ12, PES00] and in particular on a certified financial contract management engine and its associated domain-specific contract specification language [BBE15]. This framework allows for expressing a wide variety of financial contracts (a fundamental notion in financial software) and for reasoning about their functional properties (e.g., horizon and causality).

As in the previous work, the contract language that we consider is equipped with a denotational semantics, which is independent of stochastic aspects and depends only on an *external environment* $\texttt{Env} : \mathbb{N} \times \texttt{Label} \to \mathbb{R} \cup \mathbb{B}$, which maps observables (e.g., the price of a stock on a particular day) to values. We will refer to the contract language as described in [BBE15] and its extension developed in this chapter as the CL language. As the first contribution of this work, we present a certified compilation scheme that compiles a contract into a *payoff function*, which aggregates all cashflows in the contract, after discounting them according to some model. The result represents a single "snapshot" value of the contract. The payoff language is inspired by traditional payoff languages, and it is well suited for integration with Monte Carlo simulation techniques for pricing. It is essentially a small subset of a C-like expression language enriched with notation for looking up observables in the external environment. We show that compilation from CL to the payoff language preserves the cashflow

semantics.

The contract language described in [BBE15], deals with concrete contracts, such as a one year European call option on the AAPL (Apple) stock with strike price $100. The lack of genericity means that each time a new contract is created (even a very similar one), the contract management engine needs to compile the contract into the payoff language and further into a target language for embedding into the pricing engine. As our second contribution, we introduce the notion of a *financial instrument*, which allows for templating of contracts and which can be turned into a concrete contract by instantiating template variables with particular values. For example, a European call option instrument has template parameters such as maturity (the end date of the contract), strike, and the underlying asset that the option is based on. Compiling such a template once allows the engine to reuse compiled code, giving various parameter values as input to the pricing engine.

Moreover, an inherent property of contracts is that they evolve over time. This property is precisely captured by a contract reduction semantics. Each day, a contract becomes a new "smaller" contract, thus, for pricing purposes, contracts need to be recompiled at each time step, resulting in a dramatic compile time overhead. As our the third contribution we introduce a mechanism allowing for avoiding recompilation in relation with the contract evolution. A payoff expression can be parameterised over the *current time* so that evaluating the payoff code at time t gives us the same result (upto discounting) as first advancing the contract to time t, then compiling it to the payoff code, and then evaluating the result. Most of the payoff languages used in real-world applications require synchronization of the contract and the payoff code once a contract evolves [Lex08, Contract State and Pricing Synchronization]. But in some cases, as we mentioned earlier, it is important to capture the reduction semantics in the payoff language as well. Our result allows for using a single compilation procedure for both use cases: compiling a contract upfront and synchronizing at each time step.

The contract analysis and transformation code forms a core code base, which financial software crucially depends on. A certified programming approach using the Coq proof assistant allows us to prove various correctness results and to extract certified executable code.

The rest of the chapter is structured as follows. We describe an extension of the original contract language [BBE15] with template expressions for temporal parameters in Section 2.2.1. Next, we describe the syntax and the semantics of the payoff intermediate language designed to capture aspects relevant for the contract pricing purposes in Section 2.3. Section 2.4 describes our compilation approach, including a novel technique to transfer the contract evolution behavior to payoff expressions. We also state and sketch proofs of soundness theorems with respect to the denotational and the reduction semantics of contracts. Formalisation of our contract compilation approach, including code extraction, along with the example of Haskell code generation, are described in Section 2.5.

$$
\begin{aligned}
c &::= \texttt{zero} \mid \texttt{transfer}(p_1, p_2, a) \mid \texttt{scale}(e, c) \mid \\
&\quad\ \texttt{translate}(t, c) \mid \texttt{ifWithin}(e, t, c_1, c_2) \mid \texttt{both}(c_1, c_2) \\
e &::= op(e_1,\ e_2,\ \ldots,\ e_n) \mid \texttt{obs}(l, i) \mid \texttt{acc}(\lambda v.\ e_1, n, e_2) \mid r \mid b \\
t &::= n \mid v \\
op &::= \texttt{add} \mid \texttt{sub} \mid \texttt{mult} \mid \texttt{lt} \mid \texttt{neg} \mid \texttt{cond} \mid \ \ldots
\end{aligned}
$$

Figure 2.1: Syntax of CL.

2.2 The Contract Language

2.2.1 Syntax and Semantics

We assume a countably infinite set of program variables, ranged over by v. Moreover, we use n, i, r, and b to range over natural numbers, integers, reals, and booleans. We use p to range over parties. The contract language (CL) that we consider follows the style of [BBE15] and is extended with template variables (see Figure 2.1).

Expressions (e) may contain *observables*, which are interpreted in an external environment. The `acc` construct allows for accumulating a value over a given number of days n.

A contract (c) may be empty(`zero`), a transfer of one unit of some asset a (`transfer`), a scaled contract (`scale`), a translation of a contract into the future (`translate`), the composition of two contracts (`both`), or a generalised conditional $\texttt{ifWithin}(cond, t, c_1, c_2)$, which checks the condition *cond* repeatedly during the period given by t and evaluates to c_1 if $cond = \texttt{true}$ or to c_2 if *cond* never evaluates to `true` during the period t.

The main difference between the original version of the contract language and the version presented here is the introduction of *template expressions* (t), which, for instance, allows us to write contract templates with the contract maturity as a parameter. This feature requires refined reasoning about the temporal properties of contracts, such as causality. Certain constructs in the original contract language, such as $\texttt{translate}(n, c)$ and $\texttt{ifWithin}(cond, n, c_1, c_2)$, are designed such that basic properties of the contract language, including the property of causality, are straightforward to reason about. In particular, the displacement numbers n in the above constructs are constant positive numbers. For templating, we refine the constructs to support template expressions in place of positive constants. One of the consequences of adding template variables is that the semantics of contracts now depends also on mappings of template variables in a *template environment* $\texttt{TEnv} : \texttt{Var} \to \mathbb{N}$, which is also the case for many temporal properties of contracts. For example, the type system for ensuring causality of contracts [BBE15] and the concept of symbolic contract horizon HOR are now parameterised by template environments. The modified version of HOR is given in Figure 2.2, where $\mathcal{T}[\![t]\!]_\delta$ represents the semantics of template expressions (see Figure 2.5)

On the other hand, some properties such as *simple* or *obvious* causality can be verified without information from a template environment. Although, this property might be too restrictive for some contracts which are causal, but not

$$\begin{aligned}
\mathrm{HOR}_\delta(\texttt{zero}) &= 0 & \mathrm{HOR}_\delta(\texttt{scale}(e,c)) &= \mathrm{HOR}_\delta(c)\\
\mathrm{HOR}_\delta(\texttt{transfer}(p,q,\texttt{a})) &= 1 & \mathrm{HOR}_\delta(\texttt{translate}(t,c)) &= \mathcal{T}[\![t]\!]_\delta \oplus \mathrm{HOR}_\delta(c)\\
\mathrm{HOR}_\delta(\texttt{let } x = e \texttt{ in } c) &= \mathrm{HOR}_\delta(c)\\
\mathrm{HOR}_\delta(\texttt{both}(c_1,c_2)) &= \max(\mathrm{HOR}_\delta(c_1), \mathrm{HOR}_\delta(c_2))\\
\mathrm{HOR}_\delta(\texttt{ifWithin}(e,t,c_1,c_2)) &= \mathcal{T}[\![t]\!]_\delta \oplus \max(\mathrm{HOR}_\delta(c_1), \mathrm{HOR}_\delta(c_2))
\end{aligned}$$

where $$a \oplus b = \begin{cases} 0 & \text{if } b = 0 \\ a + b & \text{otherwise} \end{cases}$$

Figure 2.2: Symbolic horizon.

obviously causal (see [BBE15, Section 3.2]).

Let us consider examples of the contracts written in English and expressed in CL.

Example 2.1: The definition of an European option contract:

> European options are contracts that give the owner the right, but not the obligation, to buy or sell the underlying security at a specific price, known as the strike price, on the option's expiration date (investopedia.com).

Let us take: the expiration date to be 90 days into the future and set the strike at 100 USD. We can implement the European option contract with these parameters in CL as follows:

```
translate(90,
  if(obs(AAPL,0) > 100.0,
    scale(obs(AAPL,0) - 100.0, transfer(you, me, USD)),
    zero))
```

Example 2.2: Three month FX swap for which the payment schedule has been settled:

```
scale(1.000.000,
      both(all[translate(22, transfer(me, you, EUR)),
               translate(52, transfer(me, you, EUR)),
               translate(83, transfer(me, you, EUR))],
           scale(7.21,
                 all[translate(22, transfer(you, me, DKK)),
                     translate(52, transfer(you, me, DKK)),
                     translate(83, transfer(you, me, DKK))])))
```

In the example, we have written $\texttt{all}[c_1, \cdots, c_n]$ as an abbreviation for the contract $\texttt{both}(c_1, \texttt{both}(\cdots, c_n))$. We use the `all` shortcut with the `translate` combinator to implement a schedule of payments.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash r : \texttt{Real}} \qquad \frac{}{\Gamma \vdash b : \texttt{Bool}} \qquad \frac{l \in \texttt{Label}_\tau}{\Gamma \vdash \texttt{obs}(l, t) : \tau}$$

$$\frac{\Gamma \vdash e_i : \tau_i \qquad \vdash op : \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash op(e_1, \ldots, e_n) : \tau}$$

$$\frac{\Gamma, x : \tau \vdash e_1 : \tau \qquad \Gamma \quad \vdash e_2 : \tau}{\Gamma \vdash \texttt{acc}(\lambda x.\ e_1, d, e_2) : \tau}$$

$\boxed{\Gamma \vdash c : \texttt{Contr}}$

$$\frac{}{\Gamma \vdash \texttt{zero} : \texttt{Contr}} \qquad \frac{}{\Gamma \vdash \texttt{transfer}(p, q, \texttt{a}) : \texttt{Contr}}$$

$$\frac{\Gamma \vdash c : \texttt{Contr}}{\Gamma \vdash \texttt{translate}(d, c) : \texttt{Contr}} \qquad \frac{\Gamma \vdash c_i : \texttt{Contr}}{\Gamma \vdash \texttt{both}(c_1, c_2) : \texttt{Contr}}$$

$$\frac{\Gamma \vdash e : \texttt{Real} \quad \Gamma \vdash c : \texttt{Contr}}{\Gamma \vdash \texttt{scale}(e, c) : \texttt{Contr}} \qquad \frac{\Gamma \vdash e : \tau \quad \Gamma, x : \tau \vdash c : \texttt{Contr}}{\Gamma \vdash \texttt{let}\ x = e\ \texttt{in}\ c : \texttt{Contr}}$$

$$\frac{\Gamma \vdash e : \texttt{Bool} \quad \Gamma \vdash c_i : \texttt{Contr}}{\Gamma \vdash \texttt{ifWithin}(e, d, c_1, c_2) : \texttt{Contr}}$$

Figure 2.3: Typing rules for contracts and expressions of CL.

Using CL, considered in the present work, we can abstract some parameters of the contact in Example 2.1 to template variables (T for expiration date, and S for strike)[1]:

```
translate(T,
  if(obs(AAPL,0) > S,
    scale(obs(AAPL,0) - S, transfer(you, me, USD)),
    zero))
```

Such a parameterisation plays well with a way how users could interact with a contract management system. Contract templates could be exposed to users as *instruments* that can be instantiated with concrete values from users' input.

Next, we discuss how adding template expressions to the contract language affects its semantics. We extend the denotational semantics from [BBE15] to accommodate the idea of template expressions. The semantics for the expression sublanguage stays unchanged, since these expressions do not contain template expressions. That is, the semantics for an expression $e \in \texttt{Exp}$ in Figure 2.1 is given by the partial function $\mathcal{E}\,[\![e]\!] : [\![\Gamma]\!] \times \texttt{Env} \rightharpoonup [\![\tau]\!]$. On the other hand, we modify the semantic function for contacts by adding a template

[1] In our implementation we focus on contract templates allowing for template expressions to represent temporal parameters, like maturity. Other parameters, e.g. strike, could be expressed as constant observable values.

$\boxed{\vdash op : \tau_1 \times \cdots \times \tau_n \to \tau}$

$$
\begin{array}{ll}
\vdash \oplus : \texttt{Real} \times \texttt{Real} \to \texttt{Real} & \text{for } \oplus \in \{+, -, \cdot, /, max, min\} \\
\vdash \oplus : \texttt{Real} \times \texttt{Real} \to \texttt{Bool} & \text{for } \oplus \in \{\leq, <, =, \geq, >\} \\
\vdash \oplus : \texttt{Bool} \times \texttt{Bool} \to \texttt{Bool} & \text{for } \oplus \in \{\wedge, \vee\} \\
\vdash \neg : \texttt{Bool} \to \texttt{Bool} & \\
\vdash \textbf{if} : \texttt{Bool} \times \tau \times \tau \to \tau & \text{for } \tau \in \{\texttt{Real}, \texttt{Bool}\}
\end{array}
$$

Figure 2.4: Typing of expression operators.

environment as an argument:

$$
\begin{aligned}
\mathcal{C}\,[\![c]\!] &: [\![\Gamma]\!] \times \texttt{Env} \times \texttt{TEnv} \rightharpoonup \texttt{Trace} \\
\texttt{Trace} &= \mathbb{N} \to \texttt{Trans} \\
\texttt{Trans} &= \texttt{Party} \times \texttt{Party} \times \texttt{Asset} \to \mathbb{R}
\end{aligned}
$$

As the original contract semantics, it depends on the external environment $\texttt{Env} : \mathbb{N} \times \texttt{Label} \to \mathbb{R} \cup \mathbb{B}$ and variable assignments that map each free variable of type τ to a value in $[\![\tau]\!]$. Where

$$
\begin{aligned}
[\![\texttt{Real}]\!] &= \mathbb{R} \\
[\![\texttt{Bool}]\!] &= \mathbb{B}
\end{aligned}
\tag{2.1}
$$

Given a typing environment Γ, the set of *variable assignments* in Γ, written $[\![\Gamma]\!]$, is the set of all partial mappings γ from variable names to $\mathbb{R} \cup \mathbb{B}$ such that $\gamma(x) \in [\![\tau]\!]$ iff $x : \tau \in \Gamma$. The typing rules also remain the same for expressions and for contracts. These rules are presented in Figure 2.3, and the typing of expression operators is given in Figure 2.4.

By the result on the semantics of contracts [BBE15, Proposition 3], for well-typed expressions and well-typed closed contracts semantic functions $\mathcal{E}\,[\![-]\!]$ and $\mathcal{C}\,[\![-]\!]$ are total. The semantics for expressions and contracts is given in Figure 2.5.

We define an *instantiation function* that takes a contract and a template environment containing values for template variables, and produces another contract that does not contain template variables by replacing all occurrences of template variables with corresponding values from the template environment.

Definition 2.2.1 (Instantiation function)**.**

$$
\begin{aligned}
\mathsf{inst} : \texttt{Cont}r \times \texttt{TEnv} &\to \texttt{Contr} \\
\mathsf{inst}(\texttt{zero}, \delta) &= \texttt{zero} \\
\mathsf{inst}(\texttt{let } e \texttt{ in } c, \delta) &= \mathsf{inst}(c, \delta) \\
\mathsf{inst}(\texttt{transfer}(p_1, p_2, \mathsf{a}), \delta) &= \texttt{transfer}(p_1, p_2, \mathsf{a}) \\
\mathsf{inst}(\texttt{scale}(e, c), \delta) &= \mathsf{inst}(c, \delta) \\
\mathsf{inst}(\texttt{translate}(t, c), \delta) &= \texttt{translate}(\mathcal{T}\,[\![t]\!]_\delta, \mathsf{inst}(c, \delta)) \\
\mathsf{inst}(\texttt{both}(c_1, c_2), \delta) &= \texttt{both}(\mathsf{inst}(c_1, \delta), \mathsf{inst}(c_2, \delta)) \\
\mathsf{inst}(\texttt{ifWithin}(e, t, c_1, c_2), \delta) &= \texttt{ifWithin}(e, \mathcal{T}\,[\![t]\!]_\delta, \mathsf{inst}(c_1, \delta), \mathsf{inst}(c_2, \delta))
\end{aligned}
$$

$\boxed{\mathcal{T}[\![t]\!] : \mathtt{TEnv} \to \mathbb{N}}$

$$\mathcal{T}[\![n]\!]_\delta = n \qquad \mathcal{T}[\![v]\!]_\delta = \delta(v)$$

$\boxed{\mathcal{E}[\![e]\!] : [\![\Gamma]\!] \times \mathtt{Env} \to [\![\tau]\!]}$

$$\begin{aligned}
\mathcal{E}[\![r]\!]_{\gamma,\rho} = r; \quad \mathcal{E}[\![b]\!]_{\gamma,\rho} = b; \quad \mathcal{E}[\![x]\!]_{\gamma,\rho} &= \gamma(x) \\
\mathcal{E}[\![\mathtt{obs}(l,t)]\!]_{\gamma,\rho} &= \rho(l,t) \\
\mathcal{E}[\![op(e_1,\ldots,e_n)]\!]_{\gamma,\rho} &= [\![op]\!](\mathcal{E}[\![e_1]\!]_{\gamma,\rho},\ldots,\mathcal{E}[\![e_n]\!]_{\gamma,\rho}) \\
\mathcal{E}[\![\mathtt{acc}(\lambda x.\ e_1,d,e_2)]\!]_{\gamma,\rho} &= \begin{cases} \mathcal{E}[\![e_2]\!]_{\gamma,\rho} & \text{if } d=0 \\ \mathcal{E}[\![e_1]\!]_{\gamma[x\mapsto v],\rho} & \text{if } d>0 \end{cases} \\
\text{where} \qquad v &= \mathcal{E}[\![\mathtt{acc}(e_1,d-1,e_2)]\!]_{\gamma,\rho/-1}
\end{aligned}$$

$\boxed{\mathcal{C}[\![c]\!] : [\![\Gamma]\!] \times \mathtt{Env} \times \mathtt{TEnv} \to \mathbb{N} \to \mathtt{Trans}}$

$$\begin{aligned}
\mathtt{Trans} &= \mathtt{Party} \times \mathtt{Party} \times \mathtt{Asset} \to \mathbb{R} \\
\mathcal{C}[\![\mathtt{zero}]\!]_{\gamma,\rho,\delta} &= \lambda n.\lambda t.0 \\
\mathcal{C}[\![\mathtt{scale}(e,c)]\!]_{\gamma,\rho,\delta} &= \lambda n.\lambda(p,q,a).\mathcal{E}[\![e]\!]_{\gamma,\rho,\delta} \cdot \mathcal{C}[\![c]\!]_{\gamma,\rho,\delta}(n)(p,q,a) \quad \text{where} \\
&\quad (-\cdot-) : \mathbb{R} \to \mathtt{Trace} \to \mathtt{Trace} \\
&\quad \text{denotes trace multiplication defined pointwise.} \\
\mathcal{C}[\![\mathtt{both}(c_1,c_2)]\!]_{\gamma,\rho,\delta} &= \lambda n.\lambda t.\mathcal{C}[\![c_1]\!]_{\gamma,\rho,\delta}(n)(t) + \mathcal{C}[\![c_2]\!]_{\gamma,\rho,\delta}(n)(t) \quad \text{where} \\
&\quad (-+-) : \mathtt{Trace} \to \mathtt{Trace} \to \mathtt{Trace} \\
&\quad \text{denotes trace addition defined pointwise.} \\
\mathcal{C}[\![\mathtt{translate}(t,c)]\!]_{\gamma,\rho,\delta} &= \mathit{delay}(\mathcal{T}[\![T]\!]_\delta, \mathcal{C}[\![c]\!]_{\gamma,\rho,\delta}), \quad \text{where} \\
\mathit{delay}(d,f) &= \lambda n. \begin{cases} f(n-d) & \text{if } n \geq d \\ \lambda x.0 & \text{otherwise} \end{cases}
\end{aligned}$$

$$\mathcal{C}[\![\mathtt{transfer}(p,q,\mathtt{a})]\!]_{\gamma,\rho,\delta} = \begin{cases} \lambda n.\lambda t.0 & \text{if } p=q \\ \mathit{unit}_{a,p,q} & \text{otherwise, where} \end{cases}$$

$$\mathit{unit}_{a,p,q}(n)(p',q',b) = \begin{cases} 1 & \text{if } b=a, p=p', q=q', n=0 \\ -1 & \text{if } b=a, p=q', q=p', n=0 \\ 0 & \text{otherwise} \end{cases}$$

$$\mathcal{C}[\![\mathtt{let}\ x = e\ \mathtt{in}\ c]\!]_{\gamma,\rho,\delta} = \mathcal{C}[\![c]\!]_{\gamma[x\mapsto v],\rho}, \text{ where } v = \mathcal{E}[\![e]\!]_{\gamma,\rho,\delta}$$

$$\mathcal{C}[\![\mathtt{ifWithin}(e,t,c_1,c_2)]\!]_{\gamma,\rho,\delta} = \mathit{iter}(\mathcal{T}[\![t]\!]_\delta, \rho), \text{ where}$$

$$\mathit{iter}(i,\rho') = \begin{cases} \mathcal{C}[\![c_1]\!]_{\gamma,\rho'} & \text{if } \mathcal{E}[\![e]\!]_{\gamma,\rho'} = \mathit{true} \\ \mathcal{C}[\![c_2]\!]_{\gamma,\rho'} & \text{if } \mathcal{E}[\![e]\!]_{\gamma,\rho'} = \mathit{false} \wedge i = 0 \\ \mathit{delay}(1, \mathit{iter}(i-1, \rho'/1)) & \text{if } \mathcal{E}[\![e]\!]_{\gamma,\rho'} = \mathit{false} \wedge i > 0 \end{cases}$$

Figure 2.5: Denotational semantics of expressions and contracts of CL.

$\boxed{\mathcal{TC}(c)}$

$$\frac{}{\mathcal{TC}(\texttt{zero})} \qquad \frac{\mathcal{TC}(c)}{\mathcal{TC}(\texttt{let}\ e\ \texttt{in}\ c)} \qquad \frac{}{\mathcal{TC}(\texttt{transfer}(p_1, p_2, \texttt{a}))}$$

$$\frac{\mathcal{TC}(c)}{\mathcal{TC}(\texttt{scale}(e, c))} \qquad \frac{n \text{ is a numeral} \qquad \mathcal{TC}(c)}{\texttt{translate}(n, c)} \qquad \frac{\mathcal{TC}(c_1) \qquad \mathcal{TC}(c_2)}{\mathcal{TC}(\texttt{both}(c_1, c_2))}$$

$$\frac{n \text{ is a numeral} \qquad \mathcal{TC}(c_1) \qquad \mathcal{TC}(c_2)}{\mathcal{TC}(\texttt{ifWithin}(e, n, c_1, c_2))}$$

Figure 2.6: Template-closed contracts.

We define an inductive predicate that holds only for contract expressions without template variables (Figure 2.6). We call such contracts *template-closed*.

Lemma 2.2.1. *It is straightforward to establish the following fact: for any contract c and template environment δ, application of the instantiation function gives a template-closed contract:*

$$\mathcal{TC}(\mathsf{inst}(c, \delta))$$

Proof. By induction on the structure of c. ■

Lemma 2.2.2 (Instantiation soundness)**.** *For any contract c, template environments δ and δ', external environment ρ, and any value environment γ, the contract c and $\mathsf{inst}(c, \delta)$ are semantically equivalent. That is, $\mathcal{C}[\![c]\!]_{\gamma,\rho,\delta} = \mathcal{C}[\![\mathsf{inst}(c, \delta)]\!]_{\gamma,\rho,\delta'}$.*

Proof. By induction on the structure of c. The case of $\texttt{ifWithin}(e, t, c_1, c_2)$ requires inner induction on $n = \mathcal{T}[\![t]\!]_\delta$.

Notice also, that the semantic function on the right hand side takes arbitrary template environments δ', since after instantiation, the template environment does not affect the result. ■

The reduction semantics of the contract language presented in [BBE15] remains the same, although, we make additional assumption that the contract expressions is closed wrt. template variables. We provide the reduction semantics in Figure 2.7 for completeness of the presentation.

The functions $\underline{\mathsf{translate}}(n, c)$, $\underline{\mathsf{both}}(c_1, c_2)$, and $\underline{\mathsf{scale}}(e, c)$ represent corresponding smart constructors, which perform some simplifications before constructing a corresponding contract. Expressions built using smart constructors are semantically equivalent to the corresponding expressions that use ordinary constructors. The $\mathsf{sp_E}$ function denotes contract specialisation (see [BBE15, Section 4.1]).

Example 2.3: Let us consider a simple example of the contract reduction. We take the following contract containing two transfers: one transfer is scheduled

$$\boxed{c \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}$$

$$\frac{c \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}{\texttt{translate}(0, c) \overset{T}{\Longrightarrow}_{\gamma,\rho} c'} \qquad \frac{}{\texttt{zero} \overset{T_0}{\Longrightarrow}_{\gamma,\rho} \texttt{zero}} \qquad \frac{}{\texttt{transfer}(p, q, \texttt{a}) \overset{T_{p,q,a}}{\Longrightarrow}_{\gamma,\rho} \texttt{zero}}$$

$$\frac{d > 0}{\texttt{translate}(d, c) \overset{T_0}{\Longrightarrow}_{\rho} \underline{\texttt{translate}}(d-1, c)} \qquad \frac{c \overset{T}{\Longrightarrow}_{\gamma,\rho} c' \quad r = \mathsf{sp_E}(e, \gamma, \rho) \quad r \in \mathbb{R}}{\texttt{scale}(e, c) \overset{r*T}{\Longrightarrow}_{\gamma,\rho} \underline{\texttt{scale}}(r, c')}$$

$$\frac{c_i \overset{T_i}{\Longrightarrow}_{\gamma,\rho} c_i'}{\texttt{both}(c_1, c_2) \overset{T_1+T_2}{\Longrightarrow}_{\gamma,\rho} \underline{\texttt{both}}(c_1', c_2')} \qquad \frac{c \overset{T_0}{\Longrightarrow}_{\gamma,\rho} c' \quad e' = \mathsf{sp_E}(e, \gamma, \rho)}{\texttt{scale}(e, c) \overset{T_0}{\Longrightarrow}_{\gamma,\rho} \underline{\texttt{scale}}(\mathsf{translExp}(-1, e'), c')}$$

$$\frac{\begin{array}{c}\mathsf{sp_E}(e, \gamma, \rho) = e' \\ c \overset{T}{\Longrightarrow}_{\gamma',\rho} c'\end{array} \qquad \gamma' = \begin{cases}\gamma[x \mapsto e'] & \text{if } e' \in \mathbb{R} \cup \mathbb{B} \\ \gamma & \text{otherwise}\end{cases}}{\texttt{let } x = e \texttt{ in } c \overset{T}{\Longrightarrow}_{\gamma,\rho} \underline{\textbf{let}}\ x = \mathsf{translExp}(-1, e')\ \underline{\textbf{in}}\ c}$$

$$\frac{\mathsf{sp_E}(e, \gamma, \rho) = \mathit{false} \quad c_2 \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}{\texttt{ifWithin}(e, 0, c_1, c_2) \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}$$

$$\frac{\mathsf{sp_E}(e, \gamma, \rho) = \mathit{true} \quad c_2 \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}{\texttt{ifWithin}(e, d, c_1, c_2) \overset{T}{\Longrightarrow}_{\gamma,\rho} c'}$$

$$\frac{\mathsf{sp_E}(e, \gamma, \rho) = \mathit{false} \quad d > 0}{\texttt{ifWithin}(e, d, c_1, c_2) \quad \overset{T_0}{\Longrightarrow}_{\gamma,\rho} \quad \texttt{ifWithin}(e, d-1, c_1, c_2)}$$

where $\quad T_0 = \lambda t.0 \qquad r * T = \lambda t.r \cdot T(t)$

$$T_1 + T_2 = \lambda t.T_1(t) + T_2(t)$$

$$T_{p,q,a} = \lambda(p', q', a').\begin{cases}1 & \text{if } (p', q', a') = (p, q, a) \\ -1 & \text{if } (p', q', a') = (q, p, a) \\ 0 & \text{otherwise}\end{cases}$$

Figure 2.7: Contract reduction semantics assuming $\mathcal{TC}(c)$.

on the current day (no translation into the future), and another transfer is scheduled on the following day.

```
c :≡ both(transfer(you,me),
          translate(1,transfer(you,me)))
```

We get the following derivation tree for a one-step reduction of c:

$$\frac{}{\texttt{transfer}(you, me, \texttt{USD})} \overset{T_{you,me}}{\Longrightarrow} \texttt{zero}$$

$$\frac{\dfrac{1 > 0}{\texttt{translate}(1, \texttt{transfer}(you, me, \texttt{USD}))} \overset{T_0}{\Longrightarrow} \texttt{transfer}(you, me, \texttt{USD})}{c \overset{T_{you,me}}{\Longrightarrow} \texttt{transfer}(you, me, \texttt{USD})}$$

There is an implicit simplification in the result of reduction due to the usage of smart constructors:

$$\underline{\texttt{both}}(\texttt{zero}, \underline{\texttt{translate}}(0, \texttt{transfer}(you, me, \texttt{USD}))) = \texttt{transfer}(you, me, \texttt{USD})$$

2.2.2 Traces as a Vector Space

In Section 2.4 of the paper on CL [BBE15] it was mentioned that the set `Trans` of transfers between parties forms a vector space. In this section we will make this precise and go a bit further, discussing the set of traces `Trace` and the *delay* operation on traces.

First, we recall the definition of a vector space.

Definition 2.2.2 (Vector Space)**.** A *vector space* over a field $\mathbb{F}$ is a set V equipped with two operations:

- vector addition $- + - : V \times V \to V$
- scalar multiplication $- \cdot - : \mathbb{F} \times V \to V$

These operations satisfy the following axioms.

- associativity of vector addition: $\forall u, v, w \in V,\ (u + v) + w = u + (v + w)$;
- commutativity of vector addition: $\forall u, v \in V. u + v = v + u$;
- identity of vector addition: there exists a *zero vector* $\mathbf{0}$, s.t $\forall v \in V.\ v + \mathbf{0} = v$;
- inverse of vector addition : for every $v \in V$ there exists an *additive inverse* $-v$, s.t. $v + (-v) = \mathbf{0}$;
- compatibility of scalar multiplication with field multiplication: $\forall a, b \in \mathbb{F},\ v \in V.\ (ab) \cdot v = a \cdot (b \cdot v)$
- identity of scalar multiplication: $\forall v \in V.\ 1 \cdot u = u$
- distributivity of scalar multiplication over vector addition: $\forall a \in \mathbb{F}, v, u \in V.\ a \cdot (v + u) = a \cdot v + a \cdot u$;
- distributivity of scalar multiplication over field addition: $\forall a, b \in \mathbb{F},\ v \in V.\ (a + b) \cdot v = a \cdot v + b \cdot v$

Transfers are defined as functions to real numbers:

$$\texttt{Trans} = \texttt{Party} \times \texttt{Party} \times \texttt{Asset} \to \mathbb{R}$$

It is a well-known fact that functions to a field form a vectors space with operations given pointwise.

We shall spell out explicitly how transfers form a vector space. We use T to denote elements of `Trans`. Operations on elements of `Trans` are the following:

- $T_1 + T_2 = \lambda p.\ T_1(p) + T_2(p)$, where $p : \texttt{Party} \times \texttt{Party} \times \texttt{Asset}$
- $r \cdot T = \lambda p.\ r \cdot T(p)$

Let us now check that these two operations satisfy the axioms of a vector space.

- associativity and commutativity of transfer addition follow from the properties of addition on real numbers;
- the zero vector is just a constantly zero transfer $\mathbf{0} = \lambda p.\ 0$, and its property follows from addition with zero in $\mathbb{R}$: $(T + \mathbf{0})(p) = T(p) + 0 = T(p)$;
- the inverse is also just a negation from $\mathbb{R}$: $(-T) = \lambda p.(-T(p))$;
- identity of scalar multiplication follows from multiplication of real numbers;
- both distributivity laws follow from the fact that $\mathbb{R}$ is a field.

As one can see, we have never used any specific properties of `Trans` to define the operations, or in our reasoning about vector space axioms. The fact that `Trans` is a vector space is indeed an instance of a more general result as we could have used any function to $\mathbb{R}$ and show that it satisfies vector space axioms.

Now we consider the type of the semantics function for contracts. We define a *trace* to be a function from the set of natural numbers $\mathbb{N}$ to `Trans`

$$\texttt{Trace} = \mathbb{N} \to \texttt{Trans}$$

The semantics of well-types contracts is defined in terms of traces

$$\mathcal{C}\,[\![c]\!] : [\![\Gamma]\!] \times \texttt{Env} \times \texttt{TEnv} \to \texttt{Trace}$$

Traces are functions to `Trans` and we have shown that `Trans` is a vector space. We can show that `Trace` with operations defined pointwise also a vector space. The argument is essentially the same as for transfers, but with all the properties of real numbers replaced by the properties of `Trans`.

Remark 2.2.1. In addition to the usual vector space structure, transfers satisfy an additional anti-symmetry axiom. That is, for any parties p_1, p_2, and an asset a

$$T(p_1, p_2, a) = -T(p_2, p_1, a)$$

This property extends pointwise to traces as well. This means that CL is interpreted into a subspace of the vector space `Trace` given by all traces with anti-symmetry property.

Contract combinators, like `scale` and `both` are interpreted as operations of a vector space `Trace`: scalar multiplication and vector addition respectively. In addition to these operations, the `translate` combinator of the contract language could be interpreted in terms of vectors spaces. Let us first recall a definition of a linear map.

Theorem 2.2.1 (Linear map)**.** *For two vector spaces V and W over the same field $\mathbb{F}$ we call the function $f : V \to W$ a linear map, if it preserves vector addition and vector multiplication. That is, it satisfies the following conditions (for any $v \in V$, $w \in W$ and $a \in \mathbb{F}$)*

$$f(v + w) = f(v) + f(w)$$
$$f(a \cdot v) = a \cdot f(v)$$

Let as consider the delay operation on contracts (see 2.5):

$$\begin{aligned} &\mathit{delay}(-,-) : \mathbb{N} \times \texttt{Trace} \to \texttt{Trace} \\ &\mathit{delay}(d, f) = \lambda n. \begin{cases} f(n-d) & \text{if } n \geq d \\ \lambda x.0 & \text{otherwise} \end{cases} \end{aligned} \tag{2.2}$$

For some fixed $n \in \mathbb{N}$ we have $\mathit{delay}(n,-) : \texttt{Trace} \to \texttt{Trace}$. We know that `Trace` is a vector space, so we can ask if the *delay* function is a linear map.

Lemma 2.2.3. *The delay function defined by equation 2.2 is a linear map.*

Proof. First, we show that *delay* preserves vector addition. Fix some $n \in \mathbb{N}$, we write delay_n for $\mathit{delay}(n,-)$, and $\overline{T}$ for elements on `Trace`. By functional extensionality, for any $t \in \mathbb{N}$, we have to show

$$\mathit{delay}_n(\overline{T}_1 + \overline{T}_2)(t) = \mathit{delay}_n(\overline{T}_1)(t) + \mathit{delay}_n(\overline{T}_2)(t)$$

We proceed by cases of $t \geq n$.

- $t \geq n$:

$$\begin{aligned} \mathit{delay}_n(\overline{T}_1 + \overline{T}_2)(t) &= \text{by definition of } \mathit{delay} \text{ for } t \geq n \\ &= (\overline{T}_1 + \overline{T}_2)(t-n) \\ &= \text{by definition of pointwise addition of traces} \\ &= \overline{T}_1(t-n) + \overline{T}_2(t-n) \\ &= \text{by definition of } \mathit{delay} \text{ for } \overline{T}_1 \text{ and } \overline{T}_2 \\ &= \mathit{delay}_n(\overline{T}_1)(t) + \mathit{delay}_n(\overline{T}_2)(t) \end{aligned}$$

- otherwise:

$$\mathbf{0} = \mathbf{0} + \mathbf{0}$$

  where $\mathbf{0} = \lambda x.0$ is the constantly zero transfer. This holds by the identity of vector addition axiom.

Now we show that *delay* preserves scalar multiplication. Fix some $n \in \mathbb{N}$. By functional extensionality, for any $t \in \mathbb{N}$ we have to show

$$\mathit{delay}_n(r \cdot \overline{T})(t) = r \cdot \mathit{delay}_n(\overline{T})(t)$$

We proceed by cases of $t \geq n$.

- $t \geq n$:

$$\begin{aligned} \mathit{delay}_n(r \cdot \overline{T})(t) &= \text{by definition of } \mathit{delay} \text{ for } t \geq n \\ &= (r \cdot \overline{T})(t - n) \\ &= \text{by definition of pointwise scalar multiplication of traces} \\ &= r \cdot \overline{T}(t - n) \\ &= \text{by definition of } \mathit{delay} \text{ for } \overline{T} \\ &= r \cdot \mathit{delay}_n(\overline{T})(t) \end{aligned}$$

- otherwise:

$$\mathbf{0} = r \cdot \mathbf{0}$$

where $\mathbf{0} = \lambda x.0$. This is one of the properties of vectors, which can be derived from the basic vector space axioms.

■

Knowing that `Trace` is a vector space gives us the important set of properties, which are used in the proofs involving the contract semantics. We will see some examples in the proof of compilation soundness in Section 2.4. Moreover, a many contract equivalences are direct reflections of vector space axioms.

2.3 The Payoff Intermediate Language

2.3.1 Motivation

The contract language allows for capturing different aspects of financial contracts. We consider a particular use case for the contract language, where one wants to calculate an estimated price of a contract according to some stochastic model by performing simulations. Simulations is often implemented using Monte Carlo techniques, for instance, by evaluating a contract price at current time for randomly generated possible market scenarios and discounting the outcome according to some model. A software component that implements such a procedure is called a *pricing engine* and aims to be very efficient in performing large amount of calculations by exploiting the parallelism [ABB+16]. For this use case, one has to take the following aspects into account:

- Contracts should be represented as simple functions that take prices of assets involved in the contract (randomly generated by a pricing engine) and return one value corresponding to the overall outcome of the contract.
- The resulting value of the contract should be discounted according to a given discount function.

One way of achieving this would be to implement an interpreter for the contract language as part of a pricing engine. Although this approach is quite general, interpreting a contract in the process of pricing will cause significant performance overhead. Moreover, it will be harder to reason about correctness of the interpreter, since it could require non-trivial encoding in languages targeting

GPGPU devices. For that reason we take another approach: translating a contract from CL to an intermediate representation and, eventually, to a function in the pricing engine implementation language.

The main motivation behind the payoff language is to bridge the gap between CL and programming languages usually used to implement pricing engines. The payoff language should be relatively easy to compile to various target languages such as Haskell, Futhark [HEO14], or OpenCL. We would like to consider a language containing fewer domain-specific features and being closer to a subset of some general purpose language, making a mapping from the payoff language to a target language straightforward. Moreover, we would like to parameterise payoff expressions with template expressions, like in our extended CL. In addition to template expressions we want payoff expressions to capture the dynamic nature of contracts: reduction with the passage of time. This feature is usually not present in most of payoff languages, but it is important for efficient interaction with a pricing engine. Since our target languages include high-performance languages for GPGPU computing, and payoff functions are usually relatively small pieces of code, for efficient execution one often needs to inline these functions. If our payoff code was not parametric, it would require recompiling of big portions of the pricing engine code base when contracts evolve in time. Sometimes it is also necessary to estimate the sensitivity of a contract to the passage of time. In this case one wants the time parameter to be a part of the payoff expression.

2.3.2 Syntax and Semantics

The payoff intermediate language is an expression language ($il \in \texttt{ILExpr}$) with binary and unary operations, extended with conditionals and generalised conditionals `loopif`, behaving similarly to `ifWithin`. Template expressions ($t \in \texttt{TExprZ}$) in this language are extensions of the template expressions of the contract language with integer literals and addition. We will often refer to this language as *payoff expressions*.

$$
\begin{aligned}
il &::= \texttt{now} \mid \texttt{model}(l, t) \mid \texttt{if}(il_1, il_2, il_3) \mid \\
&\quad\ \texttt{loopif}(il, il, il, t) \mid \texttt{payoff}(t, p, p) \mid \\
&\quad\ \mathit{unop}(il) \mid \mathit{binop}(il_1, il_2) \mid t^e \\
\mathit{unop} &::= \texttt{neg} \mid \texttt{not} \\
\mathit{binop} &::= \texttt{add} \mid \texttt{mult} \mid \texttt{sub} \mid \texttt{lt} \mid \texttt{and} \mid \texttt{or} \mid \texttt{ltn} \mid \ldots \\
t^e &::= n \mid i \mid v \mid \texttt{tplus}(t_1^e, t_2^e)
\end{aligned}
$$

The semantics of payoff expressions (Figure 2.8) depends on environments $\rho \in \texttt{Env}$ and $\delta \in \texttt{TEnv}$ similarly to the semantics of the contract language. Payoff expressions can evaluate to a value of type $\mathbb{N}$, $\mathbb{R}$, or $\mathbb{B}$. We add $\mathbb{N}$ to the semantic domain, because we need to interpret the `now` construct, which represents the "current time" parameter and template expressions t^e. The semantics also depends on a *discount function* $d : \mathbb{N} \to \mathbb{R}$. The $t_0 \in \mathbb{N}$ parameter is used to add relative time shifts introduced by the semantics of `loopif`; t is a current time, which will be important later, when we introduce a mechanism to cut payoffs before a certain point in time.

The semantics for unary and binary operations is a straightforward mapping to corresponding arithmetic and logical operations, provided that the

$$\boxed{\mathcal{IL}\,[\![il]\!] : \texttt{Env} \times \texttt{TEnv} \times \mathbb{N} \times \mathbb{N} \times (\mathbb{N} \to \mathbb{R}) \times \texttt{Party} \times \texttt{Party} \rightharpoonup \mathbb{N} \cup \mathbb{R} \cup \mathbb{B}}$$

$$\begin{aligned}
\mathcal{IL}\,[\![t^e]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} &= \mathcal{T}\,[\![t^e]\!]_\delta + t_0 \\
\mathcal{IL}\,[\![unop(il)]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} &= [\![unop]\!]\,(\mathcal{IL}\,[\![il]\!]_{\rho,\delta,t_0,t,d,p_1,p_2}) \\
\mathcal{IL}\,[\![binop(il_0,il_1)]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} &= [\![binop]\!]\,(\mathcal{IL}\,[\![il_0]\!]_{\rho,\delta,t_0,t,d,p_1,p_2}\,,\ \mathcal{IL}\,[\![il_1]\!]_{\rho,\delta,t_0,t,d,p_1,p_2}) \\
\mathcal{IL}\,[\![\texttt{model}(l,t^e)]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} &= \rho(l, \mathcal{T}\,[\![t^e]\!]_\delta + t_0) \\
\mathcal{IL}\,[\![\texttt{now}]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} &= t \\
\mathcal{IL}\,[\![if(il_0,il_1,il_2)]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} &= \begin{cases} \mathcal{IL}\,[\![il_1]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} & \text{if } \mathcal{IL}\,[\![il_0]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} = true \\ \mathcal{IL}\,[\![il_2,]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} & \text{if } \mathcal{IL}\,[\![il_0]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} = false \end{cases} \\
\mathcal{IL}\,[\![\texttt{payoff}(t^e,p_1',p_2')]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} &= \begin{cases} d(\mathcal{T}\,[\![t^e]\!]_\delta) & \text{if } p_1' = p_1, p_2' = p_2 \\ -d(\mathcal{T}\,[\![t^e]\!]_\delta) & \text{if } p_1' = p_2, p_2' = p_1 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

$$\mathcal{IL}\,[\![\texttt{loopif}(il_0,il_1,il_2,t^e)]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} = iter(\mathcal{T}\,[\![t^e]\!]_\delta\,, t_0), where$$

$$iter\ n\ t_0 = \begin{cases} \mathcal{IL}\,[\![il_1]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} & \text{if } \mathcal{IL}\,[\![il_0]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} = true \\ \mathcal{IL}\,[\![il_2]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} & \text{if } \mathcal{IL}\,[\![il_0]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} = false \wedge n = 0 \\ iter(n-1)(t_0+1) & \text{if } \mathcal{IL}\,[\![il_0]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} = false \wedge i > 0 \end{cases}$$

Figure 2.8: Semantics of payoff expressions.

arguments have appropriate types. For example, $[\![\texttt{add}]\!]\,(v_1, v_2) = v_1 + v_2$, if $v_1, v_2 \in \mathbb{R}$.

The semantics for `loopif` is very similar to the semantics of `ifWithin`, although we do not "advance" external environments. Instead, we increment parameter t_0, which is added to the time shift when looking up a value in the semantics for `model`. The semantic function $\mathcal{IL}\,[\![-]\!]$ considers only payoffs between two parties p_1 and p_2, which are given as the last two parameters. More precisely, it considers payoffs from party p_1 to party p_2 as positive and as negative, if payoffs go in the opposite direction.

Another way of defining the semantics could be a *bilateral view* on payoffs. In this case only cashflows to or from one fixed party to any other party are considered. The semantics for the `payoff` then would be defined as follows:

$$\mathcal{IL}\,[\![\texttt{payoff}(t,p_1',p_2')]\!]_{\rho,\delta,t_0,t,d,p} = \begin{cases} d(\mathcal{T}\,[\![t]\!]_\delta) & \text{if } p_2' = p \\ -d(\mathcal{T}\,[\![t]\!]_\delta) & \text{if } p_1' = p \\ 0 & \text{otherwise} \end{cases}$$

2.4 Compiling Contracts to Payoffs

The contract language (Figure 2.1) consist of two levels, namely constructors to build contracts (c) and constructors to build expressions (e), which are used in the constructors for contracts (`scale`, `ifWithin`, etc.). We compile both levels into a single payoff language. The compilation functions $\tau_{\mathrm{e}}[\![-]\!] : \texttt{Expr}\times\texttt{TExprZ} \rightharpoonup \texttt{ILExpr}$ and $\tau_{\mathrm{c}}[\![-]\!] : \texttt{Contr}\times\texttt{TExprZ} \rightharpoonup \texttt{ILExpr}$ are recursively defined on the syntax of expressions and contracts, respectively, taking the starting time $t_0 \in \texttt{TExprZ}$ as a parameter.

$$
\begin{aligned}
\tau_{\mathrm{e}}[\![\texttt{cond}(b,\ e_0,\ e_1])]\!]_{t_0} &= \texttt{if}(\tau_{\mathrm{e}}[\![b]\!]_{t_0}, \tau_{\mathrm{e}}[\![e_0]\!]_{t_0}, \tau_{\mathrm{e}}[\![e_1]\!]_{t_0}) \\
\tau_{\mathrm{e}}[\![\texttt{obs}(l,i)]\!]_{t_0} &= \texttt{model}(l, \texttt{tplus}(t_0, i)) \\
\tau_{\mathrm{c}}[\![\texttt{transfer}(p_1,p_2,a)]\!]_{t_0} &= \texttt{payoff}(t_0,p_1,p_2) \\
\tau_{\mathrm{c}}[\![\texttt{scale}(e,c)]\!]_{t_0} &= \texttt{mult}(\tau_{\mathrm{e}}[\![e]\!]_{t_0}, \tau_{\mathrm{c}}[\![c]\!]_{t_0}) \\
\\
\tau_{\mathrm{c}}[\![\texttt{zero}]\!]_{t_0} &= 0 \\
\tau_{\mathrm{c}}[\![\texttt{translate}(t,c)]\!]_{t_0} &= \tau_{\mathrm{c}}[\![c]\!]_{\underline{\texttt{tplus}}(t_0,t)}, \quad \text{where} \\
\underline{\texttt{tplus}}(t_1,t_2) &= \begin{cases} t_1 + t_2 & \text{if } t_1, t_2 \text{ are numerals} \\ \texttt{tplus}(t_1,t_2) & \text{otherwise} \end{cases} \\
\tau_{\mathrm{c}}[\![\texttt{both}(c_0,c_1)]\!]_{t_0} &= \texttt{add}(\tau_{\mathrm{c}}[\![c_0]\!]_{t_0}, \tau_{\mathrm{c}}[\![c_1]\!]_{t_0}) \\
\tau_{\mathrm{c}}[\![\texttt{ifWithin}(e,t,c_1,c_2)]\!]_{t_0} &= \texttt{loopif}(\tau_{\mathrm{e}}[\![e]\!]_{t_0}, \tau_{\mathrm{c}}[\![c_0]\!]_{t_0}, \tau_{\mathrm{c}}[\![c_1]\!]_{t_0}, t)
\end{aligned}
$$

The important point to note here is that all the relative time shifts in CL are accumulated to the t_0 parameter. The resulting payoff expression only contains lookups in the external environment where time is given explicitly, and does not depend on nesting of time shifts as it was in the case of $\texttt{translate}(t,c)$ in CL. Such a representation allows for a more straightforward evaluation model. We also would like to emphasise that `acc` and `let` constructs are not supported by our compilation procedure. On the supported subset of the contract language, compilation functions $\tau_{\mathrm{e}}[\![e]\!]$, and $\tau_{\mathrm{c}}[\![c]\!]$ are total.

Let us show an example of the contract compilation. We consider the code of a contract in CL extended with template expressions and demonstrate how nested occurrences of `translate` are compiled to a payoff expression.

Example 2.4: We consider the following contract (`t0` and `t1` denote template variables): the party "you" transfer to the party "me" 100 USD in `t0` days in the future, and after `t1` more days "you" transfers to "me" an amount equal to the difference between the current price of the AAPL ticker and 100 USD, provided that the price of AAPL is higher then 100 USD (we use infix notation for arithmetic operations to make code more readable).

```
c =
  translate(t0,
  both(scale(100.0, transfer(you,me)),
       translate(t1,
         if(obs(AAPL,0) > 100.0,
            scale(obs(AAPL,0) - 100.0, transfer(you, me)),
            zero)))
```

This contract compiles to the following code in the Payoff Intermediate Language:

```
e =
  (100.0 * payoff(t0,you,me)) +
  if (model(AAPL,t0+t1) > 100.0,
     (model(AAPL,t0+t1) - 100.0) * payoff(t0+t1,you,me),
     0.0)
```

As one can see, all the nested occurrences of `translate` construct were accumulated from top to bottom. That is, in the `if` case we calculate payoffs and lookup for values of "AAPL" at time `(t0+t1)`.

To be able to reason about soundness of the compilation process, one needs to make a connection between the semantics of the two languages. For the expression sublanguage of CL it is simple: we can just compare the values that original expression and compiled expression evaluates to. In case of the contract language (denoted by c in Figure 2.1) the situation is different, since the semantics of contracts is given in terms of `Trace`, and expressions of the payoff intermediate language evaluate to a single value. On the other hand, we know that the compiled expression represents the sum of the contract cashflows with discount.

Before we state and sketch the proof of the soundness theorem, let us state some additional lemmas.

Lemma 2.4.1 (Delay scale). *For any $s : \mathbb{R}$, $t : \mathbb{N}$ and a trace tr, the following holds:*

$$delay(t, s \cdot tr) = s \cdot delay(t, tr)$$

Lemma 2.4.2 (Delay add). *For any $t : \mathbb{N}$ and traces tr_1,tr_2, the following holds:*

$$delay(t, tr_1 + tr_2) = delay(t, tr_1) + delay(t, tr_2)$$

Lemmas 2.4.1 and 2.4.2 correspond to the fact that the *delay* function is a linear map (see Section 2.2.2).

Lemma 2.4.3 (Common factor). *For any function $f : \mathbb{N} \to \mathbb{R}$, $s : \mathbb{R}$, and $t_0, n : \mathbb{N}$, we have the following obvious result:*

$$\sum_{t=t_0}^{t_0+n} s \cdot f(t) = s \cdot \sum_{t=t_0}^{t_0+n} f(t)$$

Lemma 2.4.4 (Split sum). *For any function $f, g : \mathbb{N} \to \mathbb{R}$, we have the following:*

$$\sum_{t=t_0}^{t_0+n} \big(f(t) + g(t)\big) = \sum_{t=t_0}^{t_0+n} f(t) + \sum_{t=t_0}^{t_0+n} g(t)$$

Lemma 2.4.5 (Sum delay). *For any trace tr $t_0, t_1, t_2 : \mathbb{N}$, discount function $d : \mathbb{N} \to \mathbb{R}$, and parties p_1, p_2, we have the following:*

$$\sum_{t=t_0}^{t_0+t_1+t_2} d(t) \cdot delay(t_0+t_1, tr)(t)(p1, p2) = \sum_{t=t_0+t_1}^{t_0+t_1+t_2} d(t) \cdot delay(t_0+t_1, tr)(t)(p_1, p_2)$$

Intuitively, Lemma 2.4.5 says that summing up the delayed trace before the delay point does not affect the result.

We assume a function $HOR : \texttt{TEnv} \times \texttt{Contr} \to \mathbb{N}$ that returns a conservative upper bound on the length of a contract. We often write $\tau_{\mathrm{e}} [\![e]\!]_0 = il$, or $\tau_{\mathrm{c}} [\![c]\!]_0 = il$ to emphasise that the compilation function returns some result. The compilation function satisfies the following properties:

Theorem 2.4.1 (Soundness)**.** *Assume parties p_1 and p_2 and discount function $d : \mathbb{N} \to \mathbb{R}$, environments $\rho \in \texttt{Env}$ and $\delta \in \texttt{TEnv}$.*

(i) If $\tau_e [\![e]\!]_0 = il$ and $\mathcal{E} [\![e]\!]_{\rho,\delta} = v_1$ and $\mathcal{IL} [\![il]\!]_{\rho,\delta,0,0,d,p_1,p_2} = v_2$ then $v_1 = v_2$.

(ii) If $\tau_c [\![c]\!]_0 = il$ and $\mathcal{C} [\![c]\!]_{\rho,\delta} = tr$, where $tr : \mathbb{N} \to \texttt{Party} \times \texttt{Party} \to \mathbb{R}$, and $\mathcal{IL} [\![il]\!]_{\rho,\delta,0,0,d,p_1,p_2} = v$ then

$$\sum_{t=0}^{\mathrm{HOR}_\delta(c)} d(t) \times tr(t)(p_1, p_2) = v$$

Proof. We will outline the proof here to show which properties we use in particular cases. This proof is completely formalised in Coq and all the details can be found in the source code.

The proof of part (i) proceeds by induction on the structure of e and mostly straightforward.

To prove part (ii) we first generalise the statement of the theorem for an arbitrary template expression t_0 : `TExpZ` in place of 0 as the initial value for the contact compilation function.[2] This is required because the compilation function aggregates all the nested time shift in the contract, and for the case of $\texttt{translate}(t, c)$ the induction hypothesis should be more general. The same approach is used to prove properties of tail-recursive functions (e.g. `fold_left`). Once the initial for the compilation function becomes t_0, we have to "compensate" this by delaying the trace of the contract. One way to do it is to use `translate` in the theorem statement. That is, assumptions become $\tau_{\mathrm{c}} [\![c]\!]_{t_0} = il$, and $\mathcal{C} [\![\texttt{translate}(\mathcal{T} [\![t_0]\!]_\delta, c)]\!]_{\rho,\delta} = tr$, and the conclusion becomes

$$\sum_{t=t_0}^{t_0+\mathrm{HOR}_\delta(c)} d(t) \times tr(t)(p_1, p_2) = v$$

The proof proceeds by induction on the structure of c. We have the following cases to consider.

Case 1 (`zero`): We use the fact that the empty trace gives the zero transfer at any point of time.

Case 2 ($\texttt{transfer}(p_1, p_2, \texttt{a})$): By case analysis on decidable equality of parties.

Case 3 ($\texttt{translate}(t, c)$): We prove this case by application of the induction hypothesis with Lemma 2.4.5.

[2] We need to generalise the initial value for `loopif` semantics from 0 to some n as well, i.e. $\mathcal{IL} [\![il]\!]_{\rho,\delta,n,0,d,p_1,p_2} = v$. This can be done in the same way as for t_0, and we omit this generalisation in the proof presented here for readability

Case 4 ($\texttt{scale}(e, c)$): In this case the sum on the left-hand side (let us call it *sigma*) is equal to a product of two values $v_1 \cdot v_2$, where v_1 comes from the expression e and v_2 from the contract c. The overall idea is to transform the sum into the product as well, so we can split the goal into two independent goals. We achieve that by using Lemmas 2.4.1 (*delay* comes from the generalisation of the theorem statement for arbitrary t_0) and 2.4.3. After that we can prove the two goals using soundness of expression compilation (part (i) of this theorem) for v_1, and the induction hypothesis for v_2.

Case 5 ($\texttt{both}(c_1, c_2)$): In this case the sigma on the left-hand side is equal to the sum of two values $v_1 \cdot v_2$, where v_1 comes from the contract c_1 and v_2 from the contract c_2. The overall idea is to transform the sigma into the sum of two sigmas, so that we can split the goal into two independent goals. Again, we achieve that by using Lemmas 2.4.2 (*delay* comes from the generalisation of the theorem statement for arbitrary t_0) and 2.4.4. After that, we can use the induction hypotheses on c_1 and c_2 to prove the two goals, but we have to do some extra work doing case analysis on $\text{HOR}_\delta(\texttt{both}(c_1, c_2))$, since it is defined as the maximum of horizons of the two contracts.

Case 6 ($\texttt{ifWithin}(e, t, c_1, c_2)$): The proof proceeds by nested induction on $n = \mathcal{T}[\![t]\!]_\delta$, for $n \in \mathbb{N}$. In the base case and in the inductive step case we perform a case analysis on the result of expression evaluation $b = \mathcal{E}[\![e]\!]_{\rho,\delta}$, for $b \in \mathbb{B}$. Moreover, in each subcase of the case analysis on b we perform a case analysis for $\text{HOR}_\delta(\texttt{ifWithin}(e, t, c_1, c_2))$ for the same reason as in the case for $\texttt{both}(c_1, c_2)$.

■

Remark 2.4.1. The first version of a soundness proof was developed for the original contract language without template expressions. The proof was somewhat easier, since the aggregation of nested time shifts introduced by $\texttt{translate}(n, c)$ constructs during compilation was implemented as addition of natural numbers, corresponding to time shifts. In the presence of template expressions, the compilation function builds a syntactic expression using the `tplus` constructor. There are some places in proofs where it was crucial to use associativity of addition to prove the goal, but this does not work for template expressions. For example, $\texttt{tplus}(\texttt{tplus}(t_1, t_2), t_3)$ is not equal to $\texttt{tplus}(t_1, \texttt{tplus}(t_2, t_3))$, because these expressions represent different syntactic trees, although semantically equivalent. Instead of restating proofs in terms of this semantic equivalence (significantly complicating the proofs), we used the following approach. The compilation function uses the *smart constructor* tplus instead of just plain construction of the template expression. This allowed us to recover the property we needed to complete the soundness proof without altering too much of its structure.

The soundness theorem (Theorem 2.4.1) makes an assumption that the compiled expression evaluates to some value. We do not develop a type system for our payoff language to ensure this property. Instead, we show that it is sufficient for a contract to be well-typed to ensure that the compiled expression always evaluates to some value.

Theorem 2.4.2 (Total semantics for compiled contracts)**.** *Assume parties p_1 and p_2 and discount function $d : \mathbb{N} \to \mathbb{R}$, well-typed external environment $\rho \in \texttt{Env}$, template environment $\delta \in \texttt{TEnv}$, and typing context Γ. We have the following two results:*

(i) for any $e \in \texttt{Exp}$, $t_0 \in \texttt{TExprZ}$ $t_0' \in \mathbb{N}$, if $\Gamma \vdash e : \tau$ $\tau_e [\![e]\!]_{t_0} = il$, then

$$\exists v,\ \mathcal{IL}\,[\![il]\!]_{\rho,\delta,t_0',0,d,p_1,p_2} = v, \quad \text{and } v \in [\![\tau]\!]$$

(ii) for any $c \in \texttt{Contr}$, t_0 t_0', if $\Gamma \vdash c$ and $\tau_c [\![c]\!]_{t_0} = il$, then

$$\exists v,\ \mathcal{IL}\,[\![il]\!]_{\rho,\delta,t_0',0,d,p_1,p_2} = v, \quad \text{and } v \in \mathbb{R}$$

Proof. The proof for the statement (i) proceeds by induction on the typing derivation for expressions (see Figure 2.3). The case for operations uses induction hypothesis, which gives a value. The part $v \in [\![\tau]\!]$ in the conclusion serves as a logical relation allowing us to get typing information required for the particular operation. The case for $\texttt{obs}(l, t)$ uses well-typedness of the external environments.

The proof for statement (ii) proceeds by induction on the typing derivation for contracts (see Figure 2.3), and uses the previously proved property of expressions (i) for $\texttt{scale}(e, c)$ and $\texttt{ifWithin}(e, t, c_1, c_2)$. The case `ifWithin` also requires a nested induction on $n = \mathcal{T}\,[\![t]\!]_\delta$, for $n \in \mathbb{N}$, and case analysis on the result of expression the evaluation $b = \mathcal{E}\,[\![e]\!]_{\rho,\delta}$, for $b \in \mathbb{B}$. ■

Notice that Theorem 2.4.2 holds for any $t_0 \in \texttt{TExprZ}$ and $t_0' \in \mathbb{N}$. These parameters do not affect totality of the semantics and can be arbitrary, but it is crucial to add appropriate delays, corresponding to arbitrary t_0 and t_0' for the compilation soundness property (see the proof of part (ii) of Theorem 2.4.1).

Theorems 2.4.1 and 2.4.2 ensure that our compilation procedure produces a payoff expression that evaluates to a value reflecting the aggregated price of a contract after discounting.

2.4.1 Avoiding recompilation

To avoid recompilation of a contract when time moves forward, we define a function cutPayoff(). This function is defined recursively on the syntax of intermediate language expressions.

$$\begin{aligned}
\mathsf{cutPayoff}(\texttt{now}) &= \texttt{now} \\
\mathsf{cutPayoff}(\texttt{model}(l,t)) &= \texttt{model}(l,t) \\
\mathsf{cutPayoff}(\texttt{if}(il_1, il_2, il_3)) &= \texttt{if}(\mathsf{cutPayoff}(il_1), \mathsf{cutPayoff}(il_2), \mathsf{cutPayoff}(il_3)) \\
\mathsf{cutPayoff}(\texttt{payoff}(t, p_1, p_2)) &= \texttt{if}(t < \texttt{now}, 0, \texttt{payoff}(t, p_1, p_2)) \\
\mathsf{cutPayoff}(\mathit{unop}(il)) &= \mathit{unop}(\mathsf{cutPayoff}(il)) \\
\mathsf{cutPayoff}(\mathit{binop}(il_1, il_2)) &= \mathit{binop}(\mathsf{cutPayoff}(il_1), \mathsf{cutPayoff}(il_2))
\end{aligned}$$

The most important case is the case for `payoff`. The function wraps `payoff` with a condition guarding whether this payoff affects the resulting value. For the remaining cases, the function recurses on subexpressions and returns otherwise unmodified expressions.

Example 2.5: Let us consider Example 2.4 again and apply the cutPayoff() function to the expression e:

```
cutPayoff(e) =
  (100.0 * disc(t0) * if(t0 < now, 0, payoff(you,me)) +
  if (model(AAPL,t0+t1) > 100.0,
     (model(AAPL,t0+t1) - 100.0) * disc(t0+t1) *
       if(t1+t0 < now, 0, payoff(you,me)),
      0.0)
```

Each payoff in the payoff expression is now guarded by the condition, comparing the time of the particular payoff with now. Notice that the templates variables t0 and t1 are mapped to concrete values in the template environment.

To be able to state a soundness property for the cutPayoff() function we again need to find a way to connect it to the semantics of CL. Since cutPayoff() deals with the dynamic behavior of the contract with respect to time, it seems natural to formulate the soundness property in this case in terms of contract reduction (Figure 2.7). The semantics of the payoff language takes the "current time" t as a parameter. We should be able to connect the t parameter to the step of contract reduction.

Remark 2.4.2. In the next lemmas we will implicitly assume parties p_1 and p_2, discount function $d : \mathbb{N} \to \mathbb{R}$, an external environment $\rho \in \texttt{Env}$, and a template environment $\delta \in \texttt{TEnv}$.

Definition 2.4.1. We say that two payoff expressions il_1 and il_2 are equivalent at (t_0, t) for if

$$\forall \rho, \delta, d, p_1, p_2.\ \ \mathcal{IL}\,[\![il_1]\!]_{\rho,\delta,t_0,t,d,p_1,p_2} = \mathcal{IL}\,[\![il_2]\!]_{\rho,\delta,t_0,t,d,p_1,p_2}$$

We write $il_1 \simeq_{(t_0,t)} il_2$ for this equivalence.

Definition 2.4.1 defines an equivalence parameterised by two parameters: $t_0 \in \mathbb{N}$ (a "counter" of loopif iterations), and t (time, up to which we want to ignore the payoffs).

Let us first start with a simple property showing that if we take "current time" to be zero, i.e. $t = 0$ in the semantic function, then the application of cutPayoff() should not have any effect.

Lemma 2.4.6. *Any* il : ILExpr *is equivalent at* $(t_0, 0)$ *to* cutPayoff(il) *for any* $n \in \mathbb{N}$

$$\textsf{cutPayoff}(il) \simeq_{(n,0)} il$$

The next lemmas show other properties when the value of a payoff expression stays the same after application of cutPayoff().

We have the following obvious property for the expression sublanguage of CL.

Lemma 2.4.7. *For any contract expression* $e \in$ Exp, *payoff expression* il, *and* $t_0 \in$ ILTExprZ, *if* $\tau_e\,[\![e]\!]_{t_0} = il$ *then*

$$\textsf{cutPayoff}(il) = il$$

Notice that for compiled contract expressions `Exp` we have a stronger property, stated using just equality of terms, and not the equivalence. The reason for this is that there are no `payoff` constructs in compiled contract expressions and application of $\mathsf{cutPayoff}()$ does not affect the payoff expression.

Lemma 2.4.8. *For any contract c, payoff expression il, $t_0 \in \texttt{ILTExprZ}$, $n \in \mathbb{N}$, current time t_{now}, if $\tau_c [\![c]\!]_{t_0} = il$, and $t_{now} \leq \mathcal{T} [\![t_0]\!]_\delta$, then*

$$\mathsf{cutPayoff}(il) \simeq_{(n,t_{now})} il$$

Lemma 2.4.9. *For any contract c, payoff expression il, $n \in \mathbb{N}$, current time t_{now}, if $t_{now} \leq n$, then*

$$\mathsf{cutPayoff}(il) \simeq_{(n,t_{now})} il$$

Since the contract reduction relation uses smart constructors, we would like to show how they interact with functions involved in the definition of soundness.

Lemma 2.4.10. *For any contract c, contract expression e, and template environment δ, if e is not a zero literal, then the following property holds:*

$$\mathrm{HOR}_\delta(c) = \mathrm{HOR}_\delta(\underline{\mathsf{scale}}(e, c))$$

Lemma 2.4.11. *For any contracts c_1, c_2 and a template environment δ, then the following property holds:*

$$\mathrm{HOR}_\delta(\texttt{both}(c_1, c_2)) = \mathrm{HOR}_\delta(\underline{\mathsf{both}}(c_1, c_2))$$

Now, we can state a theorem relating the semantics of the payoff intermediate language with the contract reduction semantics.

Theorem 2.4.3 (Contract compilation soudness wrt. contract reduction)**.** *We assume parties p_1, p_2, discount function $d : \mathbb{N} \to \mathbb{R}$. For any well-typed and template-closed contact c, i.e. we assume $\Gamma \vdash c$, and $\mathcal{TC}(c)$, an external environment $\rho' \in \texttt{Env}$ extending a partial external environment $\rho \in \texttt{Env}_\texttt{p}$, if c steps to some c' by the reduction relation $c \overset{T}{\Longrightarrow}_\rho c'$, for some transfer T, such that $\mathcal{C} [\![c']\!]_{\rho'/1} = trace$, and $\tau_c [\![c]\!]_0 = il$, then*

$$\sum_{t'=0}^{\mathrm{HOR}_\delta(c')} d(t'+1) \times trace(t') = \mathcal{IL} [\![\mathsf{cutPayoff}(P)]\!]_{\rho',\delta,0,1,d,p_1,p_2}$$

Where $\rho'/1$ denotes the external environment ρ advanced by one time step:

$$\rho'/1 = \lambda(l, i).\ \rho'(l, i+1), \quad l \in \texttt{Label}, i \in \mathbb{Z}$$

Proof. The proof proceeds by induction on the derivation for the contract reduction. We group cases according to the shape of the contract c and show only the general structure of the proof, highlighting which lemmas we use in particular cases.

Case 1 (`zero`): Both values are zeros.

Case 2 ($\texttt{transfer}(p_1, p_2, \texttt{a})$): By case analysis on decidable equality of parties. In the case parties are not equal, we observe that condition $0 < t_{now}$ is true, since $t_{now} = 1$ and that gives zeros on both sides of the equation.

Case 3 ($\texttt{translate}(t, c)$): We have two subcases: $t = 0$ and $t = n + 1$ for some $n \in \mathbb{N}$.

- $t = 0$. By induction hypothesis.
- $t = n + 1$. We observe that contract c is translated $n + 1$ steps into the future. Ignoring payoffs before time $t_{now} = 1$ will not affect the result of the evaluation of the corresponding payoff expression, since all the potential payoffs can happen only after $n + 1$ steps.

  That is, we use soundness of contract compilation (Theorem 2.4.1, (ii)) with Lemma 2.4.8 and soundness of contract reduction (see [BBE15, Theorem 11]).

Case 4 ($\texttt{scale}(e, c)$): The smart constructor $\underline{\mathsf{scale}}$ does not preserve the symbolic horizon in general, i.e. $\textsc{hor}_\delta(\texttt{scale}(e, c)) \neq \textsc{hor}_\delta(\underline{\mathsf{scale}}(, e)c)$. This is due to the "shortcut" behavior: if expression specialisation gives the zero literal, then the contract collapses to the empty one, giving zero horizon. For that reason we perform case analysis on the outcome of the expression specialisation.

- $\mathsf{sp}_\mathsf{E}(e, \rho) = 0$. We use soundness of expression specialisation (see [BBE15, Theorem 10]), Theorem 2.4.1(i), and Lemma 2.4.7 to proof this subcase.
- $\mathsf{sp}_\mathsf{E}(e, \rho) \neq 0$. We use the same idea as in the case of `scale` in the proof of the compilation soundness theorem to split the goal into two cases. We prove the first goal using Theorem 2.4.1(i). The second goal we can prove by induction hypothesis and Lemma 2.4.10, since we know that $\mathsf{sp}_\mathsf{E}(e, \rho) \neq 0$.

Case 5 ($\texttt{both}(c_1, c_2)$): We use the same idea to split the goal into two as in the proof of Theorem 2.4.1, and then use induction hypotheses with Lemma 2.4.11.

Case 6 ($\texttt{ifWithin}(e, t, c_1, c_2)$): This case consists of three subcases.

- e evaluates to $true$. We prove this subcase by the induction hypothesis, using Theorem 2.4.1(i) to show that the corresponding payoff expression also evaluates to $true$.
- e evaluates to $false$ and $t = 0$. The proof is similar to the previous subcase.
- e evaluates to $false$ and $t = n + 1$ for some $n \in \mathbb{N}$. We observe that the starting value for `ifWithin` is $n + 1$ and we know that e evaluates to $false$. This means that all the potential payoffs can happen after at least one step, and evaluating the corresponding payoff expression at time $t_{now} = 1$ will not affect the result. We complete the proof by using the compilation soundness (Theorem 2.4.1(ii)) and Lemma 2.4.9.

■

From the contract pricing perspective, the partial external environment ρ contains *historical data* (e.g., historical stock quotes) and the extended environment ρ' is a union of two environments ρ and ρ'', where ρ'' contains *simulated data*, produced by means of simulation in the pricing engine (e.g., using Monte Carlo techniques).

One also might be interested in the following property. The following two ways of using our compilation procedure give identical results:

- first reduce, compile, then evaluate;
- first compile, apply cutPayoff(), and then evaluate, specifying the appropriate value for the "current time" parameter.

Let us introduce some notation first. We fix the well-typed external environment ρ, the partial environment ρ', which is historically complete ($\rho'(l, i)$ is defined for all labels l and $i \leq 0$), and a discount function $d : \mathbb{N} \to \mathbb{R}$. Next, we assume that contracts are well-typed, and closed both with respect to variables bound by `let` and template variables, the compilation function is applied to supported constructs only, and that the reduction function, corresponding to the reduction relation is total on ρ' (see [BBE15, Theorem 11]). This gives us the following total functions:

$$red_{\rho'} : \texttt{Contr} \to \texttt{Contr}$$
$$\tau_{\mathrm{c}} \, [\![-]\!]_0 : \texttt{Contr} \to \texttt{ILExpr}$$

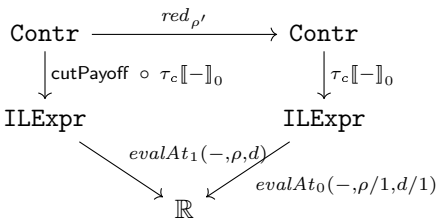
These function correspond to the contract reduction function and the contract compilation function. We also define an evaluation function for compiled payoff expressions as a shortcut for the payoff expression semantics.

$$evalAt_{\_} : \mathbb{N} \to \texttt{ILExpr} \times \texttt{Env} \times \texttt{Disc} \to \mathbb{R} + \mathbb{B}$$
$$evalAt_t(e, \rho, d) = \mathcal{IL} \, [\![e]\!]_{\rho,\emptyset,0,t,d,p_1,p_2}$$

for some parties p_1 and p_2. We know by Theorem 2.4.2 that *evalAt* is total on payoff expressions produced by the compilation function from well-typed contracts.

We summarise the property by depicting it as a commuting diagram (we give the theorem here without a proof, but emphasise that we have formalised this theorem in our Coq development).

Theorem 2.4.4. *Given notation and assumptions above, the following diagram commutes:*



Where we write $\rho/1$ and $d/1$ for shifted one step external environment and discount function, respectively.

The above diagram gives rise to the following equation:

$$evalAt_1(-, \rho, d) \circ \mathsf{cutPayoff} \circ \tau_c [\![-]\!]_0 = evalAt_0(-, \rho/1, d/1) \circ \tau_c [\![-]\!]_0 \circ red_{\rho'}$$

This property shows, that we can use our implementation in both the settings: if a contract is compiled upfront with cutPayoff(), and if a reduced contract is compiled to the payoff for each time. The former use case allows more flexibility for users. For example, one can develop a system where users define contracts directly in terms of CL working in a specialised IDE. The latter case gives performance improvement allowing for the use a set of predefined financial instruments (or contract templates). Adding a new instrument is possible, but requires recompilation.

We also point out that the path (in a diagram given in Theorem 2.4.4) $evalAt_1(-, \rho, d) \circ \mathsf{cutPayoff} \circ \tau_c [\![-]\!]_0$, requires an external environment containing all historical data from the beginning of the contract and up to t. While for the other path it is often possible to use only simulated data for pricing.

Avoiding recompilation using contract templates can significantly improve performance especially on GPGPU devices. On the other hand, additional conditional expressions are inserted into the code, which results in a number of additional checks at runtime. Experiments conducted with "hand-compiled" OpenCL code, which was semantically equivalent to the payoff language code, showed that for the simple contracts, such as European options, additional conditions, introduced by cutPayoff() do not significantly decrease performance. The estimated overhead was around 2.5 percent, while compilation time is in the order of a magnitude bigger than the total execution time.

2.5 Formalisation in Coq

Our formalisation in Coq extends the previous work [BBE15] by introducing the concept of template expressions and by developing a certified compilation technique for translating contracts to payoff expressions. The modified denotational semantics has been presented in Section 2.2.1. This modification required us to propagate changes to all the proofs affected by the change of syntax and semantics. We start this section with a description of the original formalisation, and then continue with modifications and additions made by the author of this work.

The formalisation described in [BBE15] uses an extrinsic encoding of CL. That means that syntax is represented using Coq's inductive data types, and a typing relation on these *raw* terms are given separately. For example, the type of the expression sublanguage is defined as follows.

```
Inductive Exp : Set :=
  OpE (op : Op) (args : list Exp)
| Obs (l : ObsLabel) (i : Z)
| VarE (v : Var)
| Acc (f : Exp) (d : nat) (e : Exp).
```

One of the design choices in the definition of `Exp` is to make the constructor of operations `OpE` take "code" for an operation and the list of arguments. Such an implementation makes adding new operations somewhat easier. Although, we would like to point out that this definition is a *nested* inductive definition

(see [Chl13, Section 3.8]). In such cases Coq cannot automatically derive strong enough induction principle, and it should be defined manually. In the case of `Exp` it is not hard to see, that one needs to add a generalised induction hypothesis in case of `OpE`, saying that some predicate holds for all elements in the arguments list.

Although the extrinsic encoding requires more work in terms of proving, it has a big advantage for code extraction, since simple inductive data types are easier to use in the Haskell wrapper for CL.

One of the consequences of this encoding is that semantic functions for contracts `Contr` and expressions `Exp` are partial, since they are defined on raw terms which might not be well-typed. This partiality is implemented with the `Option` type, which is equivalent to Haskell's `Maybe`. To structure the usage of these partial functions, authors define the `Option` monad and use monadic binding

```
bind : forall A B : Type, option A → (A → option B) → option B
```

to compose calls of partial functions together. The functions

```
liftM: forall A B : Type,
  (A → B) → option A → option B

liftM2 : forall A B C : Type,
  (A → B → C) → option A → option B → option C

liftM3 : forall A B C D : Type,
 (A → B → C → D) → option A → option B →
                        option C → option D
```

allow for a total function of one, two, or three arguments to be lifted to the `Option` type. The implementation includes poofs of some properties of `bind` and the lifting functions. These properties include cases for which an expression evaluates to some value.

```
bind_some : forall (A B : Type) (x : option A)
                   (v : B) (f : A → option B),
    x >>= f = Some v → exists x' : A, x = Some x' ∧ f x' = Some v
```

The similar lemmas could be proved for other functions related to the `Option` type. To simplify the work with the `Option` monad, the implementation defines tactics in the `Ltac` language (part of Coq's infrastructure). The tactics `option_inv` and `option_inv_auto` use properties of operations like `bind` and `liftM` to invert hypotheses like `e = Some v`, where e contains aforementioned functions. The implementation uses some tactics from [PdAC+16]. Particularly, the `tryfalse` tactic is widely used. It tries to resolve the current goal by looking for contradictions in assumptions, which conveniently removes impossible cases.

The original formalisation of the contract language was modified by introducing the type of template expressions

```
Parameter TVar : Set.
Inductive TExpr : Set := Tvar (t : TVar) | Tnum (n : nat).
```

We keep the type of variables abstract and do not impose any restrictions on it. Although, one could add decidability of equality for `TVar`, if required, but we do not compare template variables in our formalisation. We modify the definition of the type of contracts `Contr` such that constructors of expressions related to temporal aspects now accept `TExpr` instead of `nat` (`If` corresponds to `ifWithin`):

```
Translate : TExpr → Contr → Contr
If : Exp → TExpr → Contr → Contr → Contr.
```

and leave other constructors unmodified.

We define a template environment as a function type `TEnv := TVar → nat` similarly to the definition of the external environment. Such a definition allows for easier modification of existing code base in comparison with partial mappings. According to the definitions in Section 2.2.1 we modify the semantic function for contracts, and the symbolic horizon function to take an additional parameter of type `TEnv`. Propagation of these changes was not very problematic and almost mechanical. Although, the first attempt to parameterise the reduction relation with a template environment led to some problems, and we decided to define the reduction relation only for template-closed contracts. In most cases it is sufficient to instantiate a contract, containing template variables using the instantiation function (2.2.1), and then reduce it to a new contract. Although instantiation requires a template environment, containing all the mapping for template variables mentioned in the contract, we do not consider this a big limitation.

The definition of the payoff intermediate language (following Section 2.3.2) also uses an extrinsic encoding to represent raw terms as an inductive data type. We define one type for the payoff language expressions `ILExpr`, since there is no such separation as in CL on contracts and expressions. The definition of template expressions used in the definition of `ILExpr` is an extended version of the definition of template expressions `TExpr` used in the contract language definition.

```
Inductive ILTExpr : Set :=
  ILTplus (e1 : ILTExpr) (e2 : ILTExpr)
| ILTexpr (e : TExpr).

Inductive ILTExprZ : Set :=
  ILTplusZ (e1 : ILTExprZ) (e2 : ILTExprZ)
| ILTexprZ (e : ILTExpr)
| ILTnumZ (z : Z).
```

Notice that we use two different types of template expressions `ILTExpr` and `ILTExprZ`. The former extends the definition of `TExpr` with the addition operation, and the latter extends it further with integer literals and with the corresponding addition operation (recall that template expressions used in CL can be either natural number literals or variables). The reason why we have to extend `TExpr` with addition is that we want to accumulate time shifts introduced by `Translate` in one expression using (syntactic) addition. In the expression sublanguage of CL, observables can refer to the past by negative time indices. For that reason we introduce the `ILTExprZ` type.

The full definition of syntax for the payoff intermediate language in our Coq formalisation looks as follows:

```
Inductive ILExpr : Set :=
| ILIf : ILExpr → ILExpr → ILExpr → ILExpr
| ILFloat : R → ILExpr
| ILNat : nat → ILExpr
| ILBool : bool → ILExpr
| ILtexpr : ILTExpr → ILExpr
| ILNow  :  ILExpr
| ILModel : ObsLabel → ILTExprZ → ILExpr
| ILUnExpr : ILUnOp → ILExpr → ILExpr
| ILBinExpr : ILBinOp → ILExpr → ILExpr → ILExpr
| ILLoopIf : ILExpr → ILExpr → ILExpr → TExpr → ILExpr
| ILPayoff  : ILTExpr → Party → Party → ILExpr.
```

Notice that we use template expressions, which could represent negative numbers (`ILTExprZ`) in the constructor `ILModel`. This constructor corresponds to observable values in the contract language and allows for negative time indices corresponding to historical data.

We could have generalised our formalisation to deal with different types of template variables and add a simple type system on top of the template expression language, but we decided to keep our implementation simple, since the main goal was to demonstrate that it is possible to extend the original contract language to contract templates with temporal variables.

All the theorems and lemmas from Section 2.4 are completely formalised in our Coq development. We use a limited amount of proof automation in the soundness proofs. We use proof automation mainly in the proofs related to compilation of contract expression sublanguage, since compilation is straightforward and proofs are relatively easily to automate. Moreover, without the proof automation one would have to consider a large number of very similar cases leading to code duplication. In addition to `option_inv_auto` mentioned above, we use a tactic that helps to get rid of cases when expressions (a source expression in `Exp` and a target expression in `ILEpxr`) evaluate to values of different types (denoted by the corresponding constructor).

```
Ltac destruct_vals := repeat (match goal with
                        | [x : Val |- _] ⇒ destruct x; tryfalse
                        | [x : ILVal |- _] ⇒ destruct x; tryfalse
                        end).
```

Where the `Val` and `IVal` types corresponds to values of the contract expression sublanguage and the payoff expression language respectively.

Another tactic that significantly reduces the complexity of the proofs is the `omega` tactic from Coq's standard library. This tactic implements a decision procedure for expressions in Presburger arithmetic. That is, goals can be equations or inequations of integers, or natural numbers with addition and multiplication by a constant. The tactic uses assumptions from the current context to solve the goal automatically.

The principle we use in the organisation of the proofs is to use proof automation to solve most trivial and tedious goals and to be more explicit about the proof structure in cases requiring more sophisticated reasoning.

There are a few aspects that introduce complications to the development of proofs of the compilation properties.

- Accumulation of relative time shifts during compilation. Because of this we have to generalise our lemmas to any initial time `t0`. The same holds for the semantics of `loopif`, since there is an additional parameter in the semantics to implement iterative behavior.
- Presence of template expressions. The complications we faced with in this case are described in Remark 2.4.1. We resolves these complications with smart constructors, but it still adds some overhead.
- Conversion between types of numbers. We use integers and natural numbers (`nat` and `Z` type from the standard library). In some places, including the semantics of template expressions, we use a conversion from natural numbers to integers. This conversion makes automation with the `omega` tactic more complicated, because it requires first to use properties of conversion, which is harder to automate. With the accumulation aspect, conversions add even more overheard.
- We use contract horizon in the statement of soundness theorems, which often leads to additional case analysis in proofs.

2.5.1 Code Extraction

The Coq proof assistant allows for extracting Coq functions into programs in some functional languages [Let08]. The implementation described in [BBE15] supports code extraction of the contract type checker and contract manipulation functions into the Haskell programming language. We extend the code extraction part of the implementation with features related to contract templates and contract compilation. Particularly, we extract Haskell implementations of the following functions:

- `inst_contr` function that instantiates a given contract according to given template environment;
- `fromExp` function for compiling the contract expression sublanguage;
- `fromContr` function for compiling contract language constructs;
- `cutPayoff` function for parameterising a payoff expression with the "current time".
- `ILsem` semantic function for payoff expressions, which can be used as an interpreter.

We update the Haskell front end, exposing the contract language in convenient to use form, with combinators for contract templates. We keep the original versions of extended constructs, such as `translate` and `within` without changes and add `translateT` and `withinT` combinators supporting template variables.

Our implementation contains an extended collection of contract examples, examples of contract compilation, and evaluation of resulting payoff expressions.

2.5.2 Code Generation

To exemplify how the payoff language can be used to produce a payoff function in a subset of some general purpose language, we have implemented a code generation procedure to the Haskell programming language. That is, we have implemented the following chain of transformations:

$$\text{CL} \rightarrow \text{Payoff Intermediate Language} \rightarrow \text{Haskell}$$

We make use of the code extraction mechanism described in Section 2.5.1 to obtain a certified compilation function, which we use to translate expressions in CL to expressions in the payoff language.

The code generation procedure is (almost) a one-to-one mapping of the payoff language constructs to Haskell expressions. One primitive, which we could not map directly to Haskell build-in functions was the `loopif` construct. We have solved this issue by implementing `loopif` as a higher-order function. The implementation essentially follows the definition of the semantics of `loopif` in Coq.

```
loopif :: Int → Int → (Int → Bool) → (Int → a) → (Int → a) → a
loopif n t0 b e1 e2 = let b' = b t0 in
                      case b' of
                       True → e1 t0
                       False → case n of
                                 0 → e2 t0
                                 _ → loopif (n−1) (t0+1) b e1 e2
```

The resulting payoff function has the following signature:

```
payoff :: Map.Map ([Char], Int) Double → Map.Map [Char] Int
          → Int → Party → Party → Double
```

That is, the function takes an external environment, a template environment, current time, and two parties. The `payoff` function calls the `payoffInternal` function, which takes an additional parameter – an initial value for the `loopif` function needed for technical reasons.

Example 2.6:
We apply the code generation procedure to the expression `e` from Example 2.4. The result of code generation is given below.

```
module Examples.PayoffFunction where
import qualified Data.Map as Map
import BaseTypes
import Examples.BasePayoff
payoffInternal ext tenv t0 t_now p1 p2=
  (100.0 * (if  (X== p1 && Y== p2) then 1 else
              if  (X== p2 && Y== p1) then −1 else 0)) +
   loopif 0 t0
     (\t0→ (100.0 < (ext Map.! ("AAPL",(0 + (tenv Map.! "t1") +
                                        (tenv Map.! "t0")+ t0)))))
     (\t0→ (((ext Map.! ("AAPL",(0 + (tenv Map.! "t1") + (tenv Map.! "t0")+ t0)))
              * 100.0) * (if (X== p1 && Y== p2) then 1 else
                           if  (X== p2 && Y== p1) then −1 else 0)))
     (\t0→ 0.0)
payoff ext tenv t_now p1 p2=payoffInternal ext tenv 0 t_now p1 p2
```

As one can see, the external environment and the template environment are represented using Haskell's `Data.Map`, and `Map.!` is an infix notation for the lookup function. The Haskell code above makes use of the `loopif` function with zero as the first argument. It is possible to replace it with the regular `if` by a simple optimisation. One could also add more optimisations to our Coq implementation along with proofs of soundness.

A module declaring the `payoff` function can be used as an ordinary Haskell module as a part of the development requiring the payoff functions. For example, it could be used in the context of the FinPar benchmark [ABB+16], which contains a Haskell implementation of pricing among other routines. Moreover, the cutPayoff() function can be used to obtain a parameterised version of a payoff function in Haskell, allowing us to reproduce the contract reduction behavior.

2.6 Conclusion

This work extends the certified contract management system of [BBE15] with template expressions, which allows for drastic performance improvements and reusability in terms of the concept of instruments (i.e. contract templates). Along with changes to the contract language, we have developed a formalisation of the payoff intermediate language and a certified compilation procedure in Coq. Our approach uses an extrinsic encoding, since we are aiming at using code extraction for obtaining a correct implementation of the compiler function that translates expressions in CL to payoff expressions. We have also developed a technique allowing for capturing contract development over time.

A number of important properties, including soundness of the translation from CL to the payoff language have been proven in Coq. We have exemplified how the payoff intermediate language can be used to generate code in a target language by mapping payoff expressions to a subset of Haskell.

There are number of possibilities for future work:

- Generalise Theorems 2.4.3 and 2.4.4 to n-step contract reduction. We have defined some steps of the proofs in this generalised setting, but details still need to be worked out.

- The representation of traces as functions $\mathbb{N} \to$ `Trans` is equivalent to infinite streams of transfers. It would be interesting to explore this idea of using streams further, since observable values also can be naturally represented as streams.

- Improve the design of the payoff intermediate language to support all CL constructs. Also, adding `loopif` seems to be somewhat ad-hoc. It could be possible to a have more general language construct for iteration and compile `ifWithin` to a combination of iteration and conditions.

- Implement a translation from the payoff intermediate language to the Futhark programming language for data-parallel GPGPU computations [HSE+17]. Futhark seems to be a natural choice as a target language,

since there is an implementation of the pricing engine in Futhark, and we believe that mapping from payoff expressions to a subset of Futhark should be similar to our experience with Haskell.

- Develop a formalised infrastructure to work with external environment representations. That is, instead of finite maps as in our Haskell code generator (Section 2.5.2), one could use arrays to represent external environments. In this case one has to implement some reindexing scheme, since a naive translation of external environments could result in sparse arrays. Particularly, this will be important for targeting array languages like Futhark.

Chapter 3

Formalising Modules

In this chapter we present a number of techniques that allow for formal reasoning with nested and mutually inductive structures built up from finite maps and sets (also called semantic objects), and at the same time allow for working with binding structures over sets of variables. The techniques, which build on the theory of nominal sets combined with the ability to work with multiple isomorphic representations of finite maps, make it possible to give a formal treatment, in Coq, of a higher-order module language for Futhark, an optimising compiler targeting data-parallel architectures, such as GPGPUs [HSE+17]. We want to emphasise that the main focus of this chapter is on a formal development in the Coq proof assistant: encoding of semantic objects, formal treatment of issues related to variable binding, and proof techniques applied.

The rest of this chapter is structured as follows. In Section 3.1 we present the motivation and background for development of a module system for the Futhark language. In Section 3.3 we introduce a formal system for the module language specification. in Section 3.2, we provide a well known result demonstrating normalisation of the simply typed lambda-calculus (STLC) using a logical relation argument. The purpose of this exposition is to motivate how we can late use a similar technique to prove the normalisation of static interpretation of the module language. We give basic definitions from the theory of nominal sets and discuss motivations and applications of nominal techniques to the module language formalisation in Section 3.4. In the same section we develop an example in a simplified setting to demonstrate how nominal techniques apply to our formalisation. We discuss our Coq development in Section 3.5. The aim of this section is to highlight specifics of the development, and show applications of reasoning techniques developed to solve issues related to representation of the structures given in the previous section. Our development presented in Section 3.5 is the first treatment of the static interpretation of modules in the style of [Els99] in the Coq proof assistant. Our Coq formalisation covers definitions required to state and prove an important property of the static interpretation: static interpretation for well-typed modules terminates. While another important property - the static interpretation procedure preserves the typing of target language expressions - is not covered by our implementation and left as future work.

3.1 Motivation

Modules in the style of Standard ML and OCaml provide a powerful abstraction mechanism allowing for writing generic highly parameterised code. A common issue with an abstraction mechanism is that it can introduce a runtime overhead. For some application domains it is important to have static guarantees that module abstractions introduce no overhead. This can be done by statically interpreting the module system expressions at compile time. This technique is similar to the way C++ templates are eliminated at compile time with the difference that using modules give more static guarantees by means of a type system. The presented work extends the previous work to higher-order modules [Els99].

As an application of the abstraction mechanism provided by higher-order modules we consider a module language implemented on top of the monomorphic, first-order functional data-parallel language Futhark, which features a number of polymorphic second-order array combinators (SOACs) with parallel semantics, such as **map**, **reduce**, **scan**, and **filter**, but has no support for user-defined polymorphic higher-order functions. The module language allows for defining certain kind of polymorphic functions in Futhark with the guarantee that, at compile time, module level language constructs will be compiled away. That is, the module language gives rise to highly reusable components, which, for instance, form the grounds of a Basis Library for Futhark.

Example 3.1: For the purpose of demonstrating static interpretation in action, consider the (contrived) example Futhark program.

```
module type MT = {
  module F: (X:{ val b:int } → { val f:int→int })
}
module H = funct (M:MT) ⇒ M.F { val b = 8 }
module Main =
  H ( { module F =
          funct (X:{ val b:int }) ⇒ { fun f(x:int) = X.b+x }
      })
fun main (a:int) : int = Main.f a
```

The program declares a module type `MT` and a higher-order module `H`, which is applied to a module containing a parameterised module `F`. The result of the module application is a module containing a function `f` of type `int → int`. The contained function is called in the `main` function with the input to the program. Static interpretation partially evaluates the program to achieve the following result.

```
val b = 8
fun f (x:int) = b + x
fun main (a:int) = f a
```

The code snipped presents monomorphic target code, which can be composed, analysed, and compiled without any module language considerations. This feature provides the target language implementor with the essential meta-level abstraction property that the module language features are orthogonal to the domain of the source language.

3.2 Normalisation in the Call-by-Value Simply-Typed Lambda Calculus

In this section we present a well-known result that simply-typed lambda calculus (STLC) is normalising. We use a big-step semantics with explicit closures, and assume a call-by-value evaluation strategy for STLC. We use STLC as analogy for the static interpretation of a module language, which we will present formally in subsequent sections. For that reason, the argument usually used to prove normalisation of STLC in some adapted form is also applicable to the module language. We use Tait's method of logical relations [Tai67] in the proof we present in this section.

We assume countably infinite set of variables, ranged over by x, and i ranges over integers.

$$\tau \in Ty ::= int \mid \tau_1 \to \tau_2 \tag{3.1}$$

$$e \in Lam ::= i \mid x \mid \lambda x.e \mid e_1 e_2 \tag{3.2}$$

A context Γ maps variables to types.

$$\frac{}{\Gamma \vdash i : int}\ \text{TY-INT} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ \text{TY-VAR}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \text{TY-LAM} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_2}\ \text{TY-APP}$$

We define a big-step call-be-value operational semantics with explicit closures. Evaluation contexts E map variables to values. Values can be either closures, or integer literals.

$$v \in Val ::= \texttt{Cl}\ E\ x\ e \mid i$$

$$\frac{}{E \vdash i \Rightarrow i}\ \text{EV-INT} \qquad \frac{}{E \vdash \lambda x.e \Longrightarrow \texttt{Cl}\ E\ x\ e}\ \text{EV-LAM} \qquad \frac{E(x) = v}{E \vdash x \Longrightarrow v}\ \text{EV-LAM}$$

$$\frac{E \vdash e_1 \Longrightarrow \texttt{Cl}\ E_0\ x\ e_0 \qquad E \vdash e_2 \Longrightarrow v_0 \qquad E_0[x \mapsto v_0] \vdash e_0 \Longrightarrow v}{E \vdash e_1 e_2 \Longrightarrow v}\ \text{EV-APP}$$

We define a logical relation, which we will use in the proof of normalisation in Figure 3.1. Similarly, we define a relation on typing and evaluation contexts:

$$\frac{dom(E) = dom(\Gamma) \qquad \forall x.\ E(x) = v \wedge \Gamma(x) = \tau \Rightarrow v \models \tau}{E \models \Gamma} \tag{3.3}$$

Lemma 3.2.1. *For any typing context Γ, evaluation context E, variable x, value v and type τ, if $E \models \Gamma$ and $v \models \tau$, then $E[x \mapsto v] \models \Gamma, x : \tau$.*

$$\frac{i \in \mathbb{Z}}{i \models int}\ \text{LR-INT}$$

$$\frac{v = \texttt{Cl}\ E\ x\ e \qquad (\forall v_1.\ v_1 \models \tau_1 \Rightarrow \exists v_2.\ E[x \mapsto v_1] \vdash e_0 \Rightarrow v_2 \wedge v_2 \models \tau_2)}{v \models \tau_1 \to \tau_2}\ \text{LR-ARR}$$

Figure 3.1: Logical relation.

Theorem 3.2.1 (Normalisation[1])**.** *For any typing context* Γ*, evaluation context* E*, term* e *and type* τ*, if* $\Gamma \overset{\mathcal{T}}{\vdash} e : \tau$ *and* $\Gamma \overset{\mathcal{R}}{\models} E$ *then there exists a value* v*, such that*

$$E \vdash e \Longrightarrow v \quad \textit{and} \quad v \models \tau$$

$\mathcal{T}$ *and* $\mathcal{R}$ *denote a typing derivation and a logical relation derivation respectively.*

Proof. The proof proceeds by induction on the typing derivation $\mathcal{T}$. We consider cases for lambda abstraction and application.

Case 1 (TY-LAM): $\mathcal{T} = \dfrac{\Gamma, x : \tau_1 \overset{\mathcal{T}_1}{\vdash} e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$

We take $v = \texttt{Cl}\ E\ x\ e$. We get $E \vdash e \Longrightarrow \texttt{Cl}\ E\ x\ e$ from the evaluation rule for lambda-abstraction EV-LAM. We have to show $\texttt{Cl}\ E\ x\ e \models \tau_1 \to \tau_2$. By the rule of logical relation for the arrow type, suffices to show that assuming v_1, s.t. $v_1 \overset{\mathcal{R}_1}{\models} \tau_1$ we have $\exists v_2.\ E[x \mapsto v_1] \vdash e_0 \Rightarrow v_2 \wedge v_2 \models \tau_2$.

From Lemma 3.2.1, with $\mathcal{R}$ from assumptions and $\mathcal{R}_1$, we get $E[x \mapsto v_1] \overset{\mathcal{R}_1}{\models} \Gamma, x : \tau_1$. We complete the proof of this case by using the induction hypothesis on $\mathcal{T}_1$ with $\mathcal{R}_1$.

Case 2 (TY-APP): $\mathcal{T} = \dfrac{\Gamma \overset{\mathcal{T}_1}{\vdash} e_1 : \tau_1 \to \tau_2 \qquad \Gamma \overset{\mathcal{T}_2}{\vdash} e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_2}$

By using the induction hypothesis on $\mathcal{T}_1$ with $\mathcal{R}$, we get: there exists some v_1, s.t.

$$E \overset{\mathcal{E}_1}{\vdash} e_1 \Longrightarrow v_1 \text{ and } v_1 \overset{\mathcal{R}_1}{\models} \tau_1 \to \tau_2$$

By using the induction hypothesis on $\mathcal{T}_2$ with $\mathcal{R}$, we get: there exists some v_2, s.t.

$$E \overset{\mathcal{E}_2}{\vdash} e_2 \Longrightarrow v_2 \text{ and } v_2 \overset{\mathcal{R}_2}{\models} \tau_1$$

From $\mathcal{R}_1$, we get x, e_0, E_0, s.t.

$$v = \texttt{Cl}\ E_0\ x\ e_0$$

[1] We have developed a standalone formalisation of this theorem in Coq.
See `https://annenkov.github.io/stlcnorm/Stlc.stlc.html`

and

$$\forall v'.\ v' \models \tau_1 \Rightarrow \exists v''.\ E[x \mapsto v'] \vdash e_0 \Rightarrow v'' \wedge v'' \models \tau_2) \tag{3.4}$$

From (3.4) with v_2 and $\mathcal{R}_2$, we get: there exists some v_3, s.t.

$$E[x \mapsto v_2] \overset{\mathcal{E}_3}{\vdash} e_0 \Rightarrow v_3 \text{ and } v_3 \overset{\mathcal{R}_3}{\models} \tau_2$$

Now, take $v = v_3$ and use the rule for application (EV-APP) to construct the required derivation.

$$\frac{E \overset{\mathcal{E}_1}{\vdash} e_1 \Longrightarrow \texttt{Cl}\ E_0\ x\ e_0 \qquad E \overset{\mathcal{E}_2}{\vdash} e_2 \Longrightarrow v_2 \qquad E_0[x \mapsto v_2] \vdash e_0 \Longrightarrow v_3}{E \vdash e_1 e_2 \Longrightarrow v_3}$$

From from $\mathcal{R}_3$, we can conclude $v_3 \models \tau_2$ as required.

■

Theorem 3.2.1 can be used to show that a well-typed closed term always evaluates to some value. For example, we have the following corollary.

Corollary 3.2.1. *Every closed term of type int evaluates to an integer literal. That is, if* $\{\} \vdash t : int$ *then there exists* $i : int$, *s.t.* $\{\} \vdash e \Longrightarrow i$. *Here,* $\{\}$ *is the empty context*

Proof. In Theorem 3.2.1, take Γ and E to be the empty context $\{\}$. Then the claim follows immediately, since $\{\} \models \{\}$ is trivially satisfied. ■

3.3 Formal Specification

The module language can be considered parameterised over a core language, which, for the purpose of the presentation, is a simple functional language. We assume countably infinite sets of *type identifiers* (*tid*), *value identifiers* (*vid*), and *module identifiers* (*mid*). For each of the above identifier sets X, we define the associated set of *long identifiers* Long$X$, inductively with X as the base set and *mid.longx* as the inductive case with *longx* $\in$ Long$X$ and *mid* being a module identifier. For the module language, we also assume a denumerably infinite set of *module type identifiers* (*mtid*). Long identifiers, such as $x.y.z$, allow users to use traditional dot-notation for accessing components deep within modules and the separation of identifier classes makes it clear in what syntactic category an identifier belongs.

The simple core language is defined by notions of *type expressions* (*ty*), *core language expressions* (*exp*), and *core language declarations* (*dec*):

$$\begin{array}{lcl} ty & ::= & longtid \mid ty_1 \to ty_2 \\ exp & ::= & longvid \mid \lambda vid \to exp \mid exp_1\ exp_2 \mid exp : ty \\ dec & ::= & \textbf{val}\ vid = exp \end{array}$$

The core language can be understood entirely in isolation from the module language except that long identifiers may be used to access values and types in modules.

The grammar for the module language is given in Figure 3.2. The module language is separated into a language for specifying module types (*mty*) and

$$
\begin{array}{lcl}
mty & ::= & \texttt{\{}\ spec\ \texttt{\}} \\
 & | & mtid \\
 & | & mid : mty_1 \to mty_2 \\
 & | & mty\ \textbf{with}\ longtid = ty \\
spec & ::= & \textbf{val}\ vid : ty \\
 & | & \textbf{type}\ tid \\
 & | & \textbf{module}\ mid : mty \\
 & | & \textbf{include}\ mty \\
 & | & spec_1\ spec_2 \mid \epsilon
\end{array}
\qquad
\begin{array}{lcl}
mexp & ::= & \texttt{\{}\ mdec\ \texttt{\}} \\
 & | & mid \mid mexp\,.\,mid \\
 & | & \textbf{funct}\ mid : mty \Rightarrow mexp \\
 & | & longmid\ (mexp) \\
mdec & ::= & dec \\
 & | & \textbf{type}\ tid = ty \\
 & | & \textbf{module}\ mid = mexp \\
 & | & \textbf{module type}\ mtid = mty \\
 & | & \textbf{open}\ mexp \\
 & | & mdec_1\ mdec_2 \mid \epsilon
\end{array}
$$

Figure 3.2: Grammar for the module language excluding derived forms.

a language for declaring modules (*mdec*). The language for module types is a two-level language with sub-languages for specifying module components and for expressing module types. Similarly, the language for declaring (i.e., defining) modules is a two-level language for declaring module components and for expressing module manipulations. At the very toplevel, a program is simply a module declaration, possibly consisting of a sequence of module declarations where later declarations may depend on earlier declarations.

As will become apparent from the typing rules, in declarations of the form $mdec_1\ mdec_2$, identifiers declared by $mdec_1$ are considered bound in $mdec_2$ (similar considerations hold for composing specifications and programs).

3.3.1 Semantic objects

For the static semantics, we assume a countably infinite set TSet of *type variables* (t). A *semantic type* (or simply a type), ranged over by τ, takes the form:

$$\tau \quad ::= \quad t \mid \tau_1 \to \tau_2$$

Types relate straightforwardly to syntactic types with the difference that syntactic types contain type identifiers and semantic types contain type variables. This difference is essential in that it enables the support for type parameterisation and type abstraction.

At the core level, a *value environment* (VE) maps value identifiers (*vid*) to types and a *type environment* (TE) maps type identifiers (*tid*) to types.

The module language semantic objects are shown in Figure 3.3. The semantic objects constitute a number of mutually dependent inductive definitions. An *environment* (E) is a quadruple (TE, VE, ME, G) of a type environment TE, a variable environment VE, a *module environment* (ME), which maps module identifiers to modules, and a *module type environment* (G), which maps module type identifiers to module types. A *module* is either an environment E, representing a non-parameterised module, or a *parameterised module type* F, which is an object $\forall T.(E, \Sigma)$, for which the type variables in T are considered bound. A *module type* (Σ) is a pair, written $\exists T.M$, of a set of type variables T and a module M. In a module type $\exists T.M$, type variables in T are considered bound and we consider module types identical up-to renaming of bound variables and removal of type variables that do not appear in M. When

$$
\begin{array}{rcl}
E = (TE\,, VE\,, ME\,, G) & \in & \mathrm{Env} = \mathrm{TEnv} \times \mathrm{VEnv} \times \mathrm{MEnv} \times \mathrm{MTEnv} \\
ME & \in & \mathrm{MEnv} = \mathrm{Mid} \xrightarrow{\mathrm{fin}} \mathrm{Mod} \\
M & \in & \mathrm{Mod} = \mathrm{Env} \cup \mathrm{FunSig} \\
F = \forall T.(E, \Sigma) & \in & \mathrm{FunSig} = \mathsf{Fin}(\mathrm{TSet}) \times \mathrm{Env} \times \mathrm{MTy} \\
\Sigma = \exists T.M & \in & \mathrm{MTy} = \mathsf{Fin}(\mathrm{TSet}) \times \mathrm{Mod} \\
G & \in & \mathrm{MTEnv} = \mathrm{MTid} \xrightarrow{\mathrm{fin}} \mathrm{MTy}
\end{array}
$$

Figure 3.3: Module language semantic objects. Parameterised module types (F) and module types (Σ) are parameterised over finite sets of type variables (written $\mathsf{Fin}(\mathrm{TSet})$), ranged over by T.

T is empty, we often write M instead of $\exists\emptyset.M$. We consider module function types $\forall T.(E, \Sigma)$ identical up-to renaming of bound type variables and removal of type variables in T that do not occur free in (E, Σ).

When X is some tuple and when x is some identifier, we shall often write $X(x)$ for the result of looking up x in the appropriate projected finite map in X. Moreover, when long$x$ is some long identifier, we write $X(\mathrm{long}x)$ to denote the lookup in X, possibly inductively through module environments.

Definition 3.3.1. When X and Y are finite maps, the *modification* of X by Y, written $X+Y$, is the map with $\mathrm{Dom}(X+Y) = \mathrm{Dom}\ X \cup \mathrm{Dom}\ Y$ and values

$$(X+Y)(x) = \begin{cases} Y(x) & \text{if } x \in \mathrm{Dom}\ Y \\ X(x) & \text{otherwise} \end{cases}$$

The notion of modification is extended pointwise to tuples, as are operations such as Dom, $\cap$, and $\cup$.

Definition 3.3.2. A finite map X *extends* another finite map X', written $X \sqsupseteq X'$, if $\mathrm{Dom}\ X \supseteq \mathrm{Dom}\ X'$ and $X(x) = X'(x)$ for all $x \in \mathrm{Dom}\ X'$.

Given a particular kind of environment, such as a module environment ME, we shall often be implicit about its injection $(\{\}, \{\}, ME\,, \{\})$ into environments of type Env. Moreover, given an identifier, such as *tid*, its class specifies exactly that, given some type τ, $\{tid \mapsto \tau\}$ denotes a type environment of type TE, which again, by the above convention, can be injected implicitly into an environment of type Env.

As an example, if `t` is a type identifier, `a` and `b` are value identifiers, and `A` is a module identifier, we can write $\{\mathtt{t} \mapsto t\} + \{\mathtt{A} \mapsto \{\mathtt{a} \mapsto t\}\}$ for specifying the environment $E = (\{\mathtt{t} \mapsto t\}, \{\}, \{\mathtt{A} \mapsto E'\}, \{\})$, where $E' = (\{\}, \{\mathtt{a} \mapsto t\}, \{\}, \{\})$ and where $E \sqsupseteq \{\mathtt{t} \mapsto t\}$. Moreover, looking up the long identifier `A.a` in E, written $E(\mathtt{A.a})$, yields t.

3.3.2 Elaboration

The elaboration rules for the core language (Figure 3.4) illustrate the interaction between the module language and the core language through the concept of long identifiers.

Type Expressions

$$\boxed{E \vdash ty : \tau}$$

$$\frac{E(longtid) = \tau}{E \vdash longtid : \tau} \ (5) \qquad \frac{E \vdash ty_i : \tau_i \quad i = [1,2]}{E \vdash ty_1 \to ty_2 : \tau_1 \to \tau_2} \ (6)$$

Core language expressions

$$\boxed{E \vdash exp : \tau}$$

$$\frac{E(longvid) = \tau}{E \vdash longvid : \tau} \ (7) \qquad \frac{E \vdash exp : \tau \qquad E \vdash ty : \tau}{E \vdash exp \ \mathbf{:} \ ty : \tau} \ (8)$$

$$\frac{E + \{vid \mapsto \tau\} \vdash exp : \tau'}{E \vdash \lambda vid \to exp : \tau \to \tau'} \ (9) \qquad \frac{E \vdash exp_1 : \tau \to \tau' \qquad E \vdash exp_2 : \tau}{E \vdash exp_1 \ exp_2 : \tau'} \ (10)$$

Figure 3.4: Elaboration rules for the core language.

Module Types

$$\boxed{E \vdash mty : \Sigma}$$

$$\frac{E(mtid) = \Sigma}{E \vdash mtid : \Sigma} \ (11) \qquad \frac{\begin{array}{c} E \vdash ty : \tau \qquad E'(longtid) = t \qquad t \in T \\ E \vdash mty : \exists T.E' \qquad \Sigma = \exists (T \setminus \{t\}).(E'[\tau/t]) \end{array}}{E \vdash mty \ \mathbf{with} \ longtid = ty : \Sigma} \ (12)$$

$$\frac{E \vdash spec : \Sigma}{E \vdash \{ \ spec \ \} : \Sigma} \ (13) \qquad \frac{\begin{array}{c} E \vdash mty_1 : \exists T.E' \qquad T \cap (\mathrm{tvs}(E) \cup T') = \emptyset \\ E + \{mid \mapsto E'\} \vdash mty_2 : \exists T'.M \end{array}}{E \vdash mid \ \mathbf{:} \ mty_1 \to \ mty_2 : \forall T.(E', \exists T'.M)} \ (14)$$

Module Specifications

$$\boxed{E \vdash spec : \exists T.E'}$$

$$\frac{}{E \vdash \mathbf{type} \ tid : \exists \{t\}.\{tid \mapsto t\}} \ (15) \qquad \frac{E \vdash ty : \tau}{E \vdash \mathbf{val} \ vid \ \mathbf{:} \ ty : \{vid \mapsto \tau\}} \ (16)$$

$$\frac{E \vdash mty : \exists T.M}{E \vdash \mathbf{module} \ mid \ \mathbf{:} \ mty : \exists T.\{mid \mapsto M\}} \ (17)$$

$$\frac{E \vdash mty : \exists T.E'}{E \vdash \mathbf{include} \ mty : \exists T.E'} \ (18)$$

$$\frac{\begin{array}{c} E \vdash spec_1 : \exists T_1.E_1 \qquad E + E_1 \vdash spec_2 : \exists T_2.E_2 \\ T_1 \cap (\mathrm{tvs}(E) \cup T_2) = \emptyset \qquad \mathrm{Dom}\ E_1 \cap \mathrm{Dom}\ E_2 = \emptyset \end{array}}{E \vdash spec_1 \ spec_2 : \exists (T_1 \cup T_2).(E_1 + E_2)} \ (19) \qquad \frac{}{E \vdash \epsilon : \{\}} \ (20)$$

Figure 3.5: Elaboration rules for module types and module specifications. This sub-language does not directly depend on the rules for module expressions and module declaration.

Elaboration of module types and specifications is defined as a mutual inductive relation allowing inferences among sentences of the forms $E \vdash mty : \Sigma$ and $E \vdash spec : \exists T.E'$. The rules are presented in Figure 3.5. There is a subtle difference between module type expressions (*mty*) and specifications (*spec*). Whereas module type expressions may elaborate to parameterised module types, specifications only elaborate to non-parameterised module types, which may, however, contain parameterised modules inside. Thus, in the specification rule for including module types, we require that the included module is a non-parameterised module type.

An essential aspects of the semantic technique is that of requiring, for instance, that the sets T_1 and T_2 in Rule (19) are disjoint. This property can always be satisfied by α-renaming, which is applied often (and in the Coq implementation, explicitly) when proving properties of the language (see Example 3.9 and Remark 3.5.5 in Section 3.5.3).

The elaboration rules for module language expressions and declarations are given in Figure 3.6 and allow inferences among sentences of the forms $E \vdash mdec : \Sigma$ and $E \vdash mexp : \exists T.E$. The rules make use of the previously introduced rules for module type expressions and core language declarations and types. Similarly to the elaboration difference between module type expressions and specifications, module expressions may elaborate to general module types, of the form $\exists T.M$, whereas module declarations elaborate to non-parameterised module types of the form $\exists T.E$.

The by far most complicated rule is Rule (25), the rule for application of a parameterised module. The rule looks up a parameterised module type $\forall T_0.(E_0, \Sigma_0)$ for the long module identifier in the environment and seeks to match the parameter module type $\exists T_0.E_0$ against a cut-down version (according to the enrichment relation) of the module type resulting from elaborating the argument module expression. The result of elaborating the application is the result module type perhaps with additional abstract type variables stemming from elaborating the argument module expression. The need for also quantifying over the type set T in the result module type comes from the desire to prove a property that if $E \vdash mexp : \exists T.E'$ then $\mathrm{tvs}(E') \subseteq \mathrm{tvs}(E) \cup T$.

3.3.3 Enrichment

Next, we introduce a notion of *enrichment*. Intuitively, the enrichment relation for semantic objects is a generalised version of environment extension (Definition 3.3.2). We write $E' \succ E$ for E' enriches E meaning that E' contains the same elements as or more elements then E. The formal specification of the enrichment relation is given in Figure 3.7.

Notice that enrichment for parameterised modules is contravariant in parameter environments. Notice also the special treatment of module type environments. Because a module type cannot specify bindings of module types, we can safely require that when $E' \succ E$, the module type environment in E is empty.

3.3.4 Target Language

We assume a denumerably infinite set LSet of *labels*, ranged over by l. Target expressions are basically identical to core level expressions with the modifica-

Module Expressions

$$\boxed{E \vdash mexp : \Sigma}$$

$$\frac{E \vdash mdec : \Sigma}{E \vdash \texttt{\{}\ mdec\ \texttt{\}} : \Sigma}\ (21) \qquad \frac{E \vdash mexp : \exists T.E' \quad E'(mid) = E''}{E \vdash mexp\,\texttt{.}\,mid : \exists T.E''}\ (22)$$

$$\frac{E \vdash mty : \exists T.E' \quad E + \{mid \mapsto E'\} \vdash mexp : \Sigma \quad T \cap \mathrm{tvs}(E) = \emptyset \quad F = \forall T.(E', \Sigma)}{E \vdash \textbf{funct}\ mid\ \textbf{:}\ mty \Rightarrow mexp : \exists\emptyset.F}\ (23) \qquad \frac{E(mid) = E'}{E \vdash mid : E'}\ (24)$$

$$\frac{E \vdash mexp : \exists T.E' \quad T \cap T' = \emptyset \quad E(longmid) \geq (E'', \exists T'.E''') \quad E' \succ E'' \quad (T \cup T') \cap \mathrm{tvs}(E) = \emptyset}{E \vdash longmid\ \texttt{(}\ mexp\ \texttt{)} : \exists(T \cup T').E'''}\ (25)$$

Module Declarations

$$\boxed{E \vdash mdec : \exists T.E'}$$

$$\frac{mdec = dec \quad E \vdash dec : E'}{E \vdash mdec : E'}\ (26) \qquad \frac{E \vdash ty : \tau}{E \vdash \textbf{type}\ tid = ty : \{tid \mapsto \tau\}}\ (27)$$

$$\frac{E \vdash mexp : \exists T.M}{E \vdash \textbf{module}\ mid = mexp : \exists T.\{mid \mapsto M\}}\ (28)$$

$$\frac{E \vdash mexp : \Sigma}{E \vdash \textbf{open}\ mexp : \Sigma}\ (29)$$

$$\frac{E \vdash mty : \Sigma}{E \vdash \textbf{module type}\ mtid = mty : \exists\emptyset.\{mtid \mapsto \Sigma\}}\ (30) \qquad \frac{}{E \vdash \epsilon : \{\}}\ (31)$$

$$\frac{T_1 \cap (\mathrm{tvs}(E) \cup T_2) = \emptyset \quad E \vdash mdec_1 : \exists T_1.E_1 \quad E + E_1 \vdash mdec_2 : \exists T_2.E_2}{E \vdash mdec_1\ mdec_2 : \exists(T_1 \cup T_2).(E_1 + E_2)}\ (32)$$

Figure 3.6: Elaboration rules for module language expressions and declarations.

$$\frac{E' = (TE', VE', ME', G') \qquad E = (TE, VE, ME, \{\}) \qquad VE' \sqsupseteq VE \qquad TE' \sqsupseteq TE \qquad ME' \succ ME)}{E' \succ E}$$

$$\frac{\text{Dom } ME' \supseteq \text{Dom } ME \qquad \forall mid \in \text{Dom } ME\,.\; ME'(mid) \succ ME(mid)}{ME' \succ ME}$$

$$\frac{M' = E' \qquad M = E \qquad E' \succ E}{M' \succ M}$$

$$\frac{M' = \forall T'.(E', \Sigma') \qquad M = \forall T.(E, \Sigma) \qquad T' = T \qquad E \succ E' \qquad \Sigma' \succ \Sigma}{M' \succ M}$$

$$\frac{M' \succ M \qquad \Sigma' = \exists T.M' \qquad \Sigma = \exists T.M}{\Sigma' \succ \Sigma}$$

Figure 3.7: The enrichment relation.

$$\frac{\Gamma \vdash c_1 : \Gamma_1 \qquad \Gamma + \Gamma_1 \vdash c_2 : \Gamma_2}{\Gamma \vdash c_1 \;;\; c_2 : \Gamma_1 + \Gamma_2} \quad (33) \qquad \frac{}{\Gamma \vdash \epsilon : \{\}} \quad (34)$$

$$\frac{\Gamma \vdash ex : \tau}{\Gamma \vdash \textbf{val}\ l = ex : \{l \mapsto \tau\}} \quad (35)$$

Figure 3.8: Type rules for the target language. For the purpose of the presentation, the target language is simple and mimics closely the source language with the difference that long identifiers are replaced with labels for referring to previously defined value declarations.

tion that value identifiers are replaced with labels. For the simple core language that we are considering, *target expressions* (ex) and *target code* (c) take the form:

$$\begin{array}{lcl} ex & ::= & l \mid \lambda l \to ex \mid ex_1\ ex_2 \\ c & ::= & \textbf{val}\ l = ex \mid c_1 \;;\; c_2 \mid \epsilon \end{array}$$

The type system for the target language is simple (for the purpose of this work) and allows inferences among sentences of the forms $\Gamma \vdash ex : \tau$ and $\Gamma \vdash c : \Gamma'$, which are read: "In the context Γ, the expression ex has type τ" and "in the context Γ, the target code c declares the context Γ'. Contexts Γ map labels to types. The type system for the target language is presented in Figure 3.8.

3.3.5 Interpretation Objects

In the following, we shall use the term *name* to refer to either a type variable t or a label l. We write NSet to refer to the disjoint union of TSet and LSet. Moreover, we use N to range over finite subsets of NSet.

An *interpretation value environment* ($\mathcal{VE}$) maps value identifiers to a label and an associated type. An *interpretation environment* ($\mathcal{E}$) is a quadruple $(TE, \mathcal{VE}, \mathcal{ME}, G)$ of a type environment, an interpretation value environment, an interpretation module environment, and a module type environment. An *interpretation module environment* ($\mathcal{ME}$) maps module identifiers to module interpretations. A *module interpretation* ($\mathcal{M}$) is either an interpretation environment $\mathcal{E}$ or a functor closure Φ. A *functor closure* (Φ) is a triple $(\mathcal{E}, F, \lambda mid \Rightarrow mexp)$ of an interpretation environment, a parameterised module type, and a representation of a parameterised module expression. Finally, an *interpretation target object* ($\exists N.(\mathcal{E}, c)$) is a triple of a name set, an interpretation environment, and a target code object.

3.3.6 Interpretation Erasure

For establishing a link between interpretation objects and elaboration objects, we introduce the concept of interpretation erasure. Given an interpretation object O, we define the *interpretation erasure* of O, written $\overline{O}$, as follows:

$$\begin{array}{rcl}
\overline{(TE, \mathcal{VE}, \mathcal{ME}, G)} & = & (TE, \overline{\mathcal{VE}}, \overline{\mathcal{ME}}, G) \\
\overline{(\mathcal{E}, F, \lambda mid \Rightarrow mexp)} & = & F \\
\overline{\{vid_i \mapsto l_i : \tau_i\}^n} & = & \{vid_i \mapsto \tau_i\}^n \\
\\
\overline{\{mid_i \mapsto \mathcal{M}_i\}^n} & = & \{mid_i \mapsto \overline{\mathcal{M}_i}\}^n \\
\overline{\exists N.(\mathcal{E}, c)} & = & \exists(\mathrm{TSet} \cap N).\overline{\mathcal{E}}
\end{array}$$

3.3.7 Core Language Compilation

Core language expressions and declarations are compiled into target language expressions and declarations, respectively. The rules specifying the compilation allow inferences among sentences of the forms (1) $\mathcal{E} \vdash exp \Rightarrow ex, \tau$ and (2) $\mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c)$. The rules are given in Figure 3.9.

The rules track type information and it is straightforward to establish the following property of the compilation:

Lemma 3.3.1. *If* $\mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c)$ *then* $\overline{\mathcal{E}} \vdash dec : \overline{\exists N.(\mathcal{E}', c)}$.

3.3.8 Environment Filtering

Corresponding to the notion of enrichment for elaboration, we introduce a notion of filtering for the purpose of static interpretation, which filters interpretation environments to contain components as specified by an elaboration environment. Filtering is essential to the interpretation rule for applications of parameterised modules and is defined mutual inductively based on the structure of elaboration environments and elaboration module environments.

More formally, the *filtering* of an interpretation environment $\mathcal{E}$ to an elaboration environment E results in another interpretation environment $\mathcal{E}'$ with only elements from $\mathcal{E}$ that are also present in E. The filtering relation is defined by a number of inference rules that allow inferences among sentences of the forms (1) $\vdash \mathcal{E} :: E \Rightarrow \mathcal{E}'$, (2) $\vdash \mathcal{VE} :: VE \Rightarrow \mathcal{VE}'$, (3) $\vdash \mathcal{ME} :: ME \Rightarrow \mathcal{ME}'$, and (4) $\vdash \mathcal{M} :: M \Rightarrow \mathcal{M}'$. The inference rules for filtering are presented in

Compiling Expressions

$$\boxed{\mathcal{E} \vdash exp \Rightarrow ex, \tau}$$

$$\frac{\mathcal{E}(longvid) = (l, \tau)}{\mathcal{E} \vdash longvid \Rightarrow l, \tau} \ (36) \qquad \frac{\mathcal{E} + \{vid \mapsto (l, \tau)\} \vdash exp \Rightarrow ex, \tau'}{\mathcal{E} \vdash \lambda vid \rightarrow exp \Rightarrow \lambda l \rightarrow ex, \tau \rightarrow \tau'} \ (37)$$

$$\frac{\mathcal{E} \vdash exp_1 \Rightarrow ex_1, \tau \rightarrow \tau' \qquad \mathcal{E} \vdash exp_2 \Rightarrow ex_2, \tau}{\mathcal{E} \vdash exp_1 exp_2 \Rightarrow ex_1 \ ex_2, \tau'} \ (38)$$

$$\frac{\mathcal{E} \vdash exp \Rightarrow ex, \tau \qquad \overline{\mathcal{E}} \vdash ty : \tau}{\mathcal{E} \vdash exp \ \mathbf{:} \ ty \Rightarrow ex, \tau} \ (39)$$

Compiling Declarations

$$\boxed{\mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c)}$$

$$\frac{\mathcal{E} \vdash exp \Rightarrow ex, \tau \qquad l \notin \text{names}(\mathcal{E})}{\mathcal{E} \vdash \mathbf{val}\ vid = exp \Rightarrow \exists\{l\}.(\{vid \mapsto (l, \tau)\}, \mathbf{val}\ l = ex)} \ (40)$$

Figure 3.9: Core language compilation.

Figure 3.10. It is a straightforward to establish the connection between filtering and enrichment.

Lemma 3.3.2 (Filtering to enrichment)**.** *It is a straightforward exercise to demonstrate that if* $\vdash \mathcal{E} :: E \Rightarrow \mathcal{E}'$ *then it holds that* $\overline{\mathcal{E}} \succ E$.

3.3.9 Static Interpretation Rules

Static interpretation of the module language is defined by a number of mutually inductive inference rules allowing inferences among sentences of the forms (1) $\mathcal{E} \vdash mexp \Rightarrow \Psi$ and (2) $\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c)$, which state that in an interpretation environment $\mathcal{E}$, static interpretation of a module expression *mexp* results in an interpretation target object Ψ, and static interpretation of a module declaration *mdec* results in an interpretation target object $\exists N.(\mathcal{E}', c)$. The rules for static interpretation are presented in Figure 3.11.

3.3.10 Static Interpretation Normalisation

The proof technique we use to prove the static interpretation normalisation property is similar to the one we showed in Section 3.2. That is, we first define an appropriate logical relation (Figure 3.12), which we call the *consistency relation*, and based on this relation, we can state a property establishing that static interpretation is possible and terminates for all elaborating programs. We consider only the termination property of static interpretation, since in the current Coq formalisation we have implemented it in this form. In general, one would like to add the type soundness property, which states that target programs are appropriately typed. We leave this property for future extensions of our formalisation.

Environments

$$\boxed{\vdash \mathcal{E} :: E \Rightarrow \mathcal{E}'}$$

$$\frac{\begin{array}{c}\vdash \mathcal{V}E :: VE \Rightarrow \mathcal{V}E' \\ \vdash \mathcal{M}E :: ME \Rightarrow \mathcal{M}E' \qquad TE \succ TE'\end{array}}{\vdash (TE, \mathcal{V}E, \mathcal{M}E, \{\}) :: (TE', VE, ME, \{\}) \Rightarrow (TE', \mathcal{V}E', \mathcal{M}E', \{\})} \quad (41)$$

Value Environments

$$\boxed{\vdash \mathcal{V}E :: VE \Rightarrow \mathcal{V}E'}$$

$$\frac{m \geq n}{\vdash \{vid_i \mapsto l_i : \tau_i\}^m :: \{vid_i \mapsto \tau_i\}^n \Rightarrow \{vid_i \mapsto l_i : \tau_i\}^n} \quad (42)$$

Module Environments

$$\boxed{\vdash \mathcal{M}E :: ME \Rightarrow \mathcal{M}E'}$$

$$\frac{m \geq n \qquad \vdash \mathcal{M}_i :: M_i \Rightarrow \mathcal{M}'_i \quad i = 1..n}{\vdash \{mid_i \mapsto \mathcal{M}_i\}^m :: \{mid_i \mapsto M_i\}^n \Rightarrow \{mid_i \mapsto \mathcal{M}'_i\}^n} \quad (43)$$

Module Interpretations

$$\boxed{\vdash \mathcal{M} :: M \Rightarrow \mathcal{M}'}$$

$$\frac{\mathcal{M} = \mathcal{E} \qquad \vdash \mathcal{E} :: E \Rightarrow \mathcal{E}'}{\vdash \mathcal{M} :: E \Rightarrow \mathcal{E}'} \quad (44)$$

$$\frac{\Phi = (\mathcal{E}, F', \lambda mid \Rightarrow mexp) \qquad F' \succ F}{\vdash \Phi :: F \Rightarrow (\mathcal{E}, F, \lambda mid \Rightarrow mexp)} \quad (45)$$

Figure 3.10: Filtering relation specifying how an interpretation environment can be constrained by an elaboration environment to form a restricted interpretation environment.

Before we state the theorem and sketch its proof, let us formulate auxiliary lemmas, related to properties of relations and operations involved in the definition of the type consistency relation.

Lemma 3.3.3 (Lookup consistency)**.** *If looking up a module for some mid in any semantic object E gives a module M, and E is related to some $\mathcal{E}$ by the consistency relation (Figure 3.12), then looking up for the same mid in the interpretation environment $\mathcal{E}$ must give some module interpretation $\mathcal{M}$, such that M is consistent with $\mathcal{M}$.*

That is, if $E \models \mathcal{E}$, and $E(mid)$, then $\exists \mathcal{M}.\ \mathcal{E}(mid) = \mathcal{M} \wedge M \models \mathcal{M}$.

Lemma 3.3.4 (Uniformness)**.** *This lemma can be seen as some an inversion principle for the consistency relation: if we know something about the shape of a module, we know what the shape of a corresponding module interpretation is.*

That is, if M is a module, and M is consistent with some module interpretation $\mathcal{M}$, then:

Module Expressions

$$\boxed{\mathcal{E} \vdash mexp \Rightarrow \Psi}$$

$$\frac{\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c)}{\mathcal{E} \vdash \texttt{\{}\ mdec\ \texttt{\}} \Rightarrow \exists N.(\mathcal{E}', c)}\ (46) \qquad \frac{\mathcal{E}(mid) = \mathcal{E}'}{\mathcal{E} \vdash mid \Rightarrow \exists \emptyset.(\mathcal{E}', \epsilon)}\ (47)$$

$$\frac{\begin{array}{c}\mathcal{E}'(mid) = \mathcal{E}'' \\ \mathcal{E} \vdash mexp \Rightarrow \exists N.(\mathcal{E}', c)\end{array}}{\mathcal{E} \vdash mexp\,\texttt{.}\,mid \Rightarrow \exists N.(\mathcal{E}'', c)}\ (48)$$

$$\frac{\begin{array}{c}\overline{\mathcal{E}} \vdash mty : \exists T.E \qquad T \cap \mathrm{names}(\mathcal{E}) = \emptyset \\ \overline{\mathcal{E}} + \{mid \mapsto E\} \vdash mexp : \Sigma \qquad F = \forall T.(E, \Sigma) \\ \Phi = (\mathcal{E}, F, \lambda mid \Rightarrow mexp)\end{array}}{\mathcal{E} \vdash \textbf{funct}\ mid\ \textbf{:}\ mty \Rightarrow mexp \Rightarrow \exists \emptyset.\Phi}\ (49)$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash mexp \Rightarrow \exists N.(\mathcal{E}', c) \qquad (N \cup N') \cap \mathrm{names}(\mathcal{E}) = \emptyset \qquad N \cap N' = \emptyset \\ \mathcal{E}(longmid) = (\mathcal{E}_0, F, \lambda mid \Rightarrow mexp') \qquad F \geq (E', \exists T'.E'') \qquad T' \subseteq N' \\ \vdash \mathcal{E}' :: E' \Rightarrow \mathcal{E}'' \qquad \mathcal{E}_0 + \{mid \mapsto \mathcal{E}''\} \vdash mexp' \Rightarrow \exists N'.(\mathcal{E}''', c')\end{array}}{\mathcal{E} \vdash longmid\texttt{(}\ mexp\ \texttt{)} \Rightarrow \exists (N \cup N').(\mathcal{E}''', c\ \texttt{;}\ c')}\ (50)$$

Module declarations

$$\boxed{\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c)}$$

$$\frac{mdec = dec \qquad \mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c)}{\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c)}\ (51)$$

$$\frac{\overline{\mathcal{E}} \vdash ty : \tau}{\mathcal{E} \vdash \textbf{type}\ tid = ty \Rightarrow \exists \emptyset.(\{tid \mapsto \tau\}, \epsilon)}\ (52)$$

$$\frac{\mathcal{E} \vdash mexp \Rightarrow \exists N.(\Phi, c)}{\mathcal{E} \vdash \textbf{module}\ mid = mexp \Rightarrow \exists N.(\{mid \mapsto \Phi\}, c)}\ (53)$$

$$\frac{\overline{\mathcal{E}} \vdash mty : \Sigma \qquad \mathcal{E}' = \{mtid \mapsto \Sigma\}}{\mathcal{E} \vdash \textbf{module type}\ mtid = mty \Rightarrow \exists \emptyset.(\mathcal{E}', \epsilon)}\ (54)$$

$$\frac{\mathcal{E} \vdash mexp \Rightarrow \Psi}{\mathcal{E} \vdash \textbf{open}\ mexp \Rightarrow \Psi}\ (55)$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash mdec_1 \Rightarrow \exists N_1.(\mathcal{E}_1, c_1) \\ \mathcal{E} + \mathcal{E}_1 \vdash mdec_2 \Rightarrow \exists N_2.(\mathcal{E}_2, c_2) \\ N_1 \cap (\mathrm{names}(\mathcal{E}) \cup N_2) = \emptyset\end{array}}{\mathcal{E} \vdash mdec_1\ mdec_2 \Rightarrow \exists (N_1 \cup N_2).(\mathcal{E}_1 + \mathcal{E}_2, c_1\ \texttt{;}\ c_2)}\ (56)$$

$$\frac{}{\mathcal{E} \vdash \epsilon \Rightarrow \exists \emptyset.(\{\}, \epsilon)}\ (57)$$

Figure 3.11: Static interpretation rules for module expressions and module declarations.

$$\frac{\mathcal{E} = (TE\,, \mathcal{V}E, \mathcal{M}E, G) \qquad VE \models \mathcal{V}E \qquad ME \models \mathcal{M}E}{(TE\,, VE\,, ME\,, G) \models \mathcal{E}}$$

$$\frac{\mathrm{Dom}ME = \mathrm{Dom}\mathcal{M}E \qquad \forall mid \in \mathrm{Dom}ME\,, ME\,(mid) \models \mathcal{M}E(mid)}{ME \models \mathcal{M}E}$$

$$\frac{\mathrm{Dom}\,VE = \mathrm{Dom}\mathcal{V}E \qquad \forall x \in \mathrm{Dom}\,VE\,, \tau = VE\,(x) \wedge (l, \tau) = \mathcal{V}E(x)}{VE \models \mathcal{V}E}$$

$$\frac{M = E \qquad \mathcal{M} = \mathcal{E} \qquad E \models \mathcal{E}}{M \models \mathcal{M}}$$

$$\frac{\begin{array}{c} M = \forall T.(E, \exists T'.M') \qquad \mathcal{M} = (\mathcal{E}, \forall T.(E, \exists T'.M'), \lambda mid \Rightarrow mexp) \\ (\forall \mathcal{E}', E \models \mathcal{E}' \implies \exists N', \mathcal{M}', c.\ \ (\mathcal{E} + \{mid \mapsto \mathcal{E}'\}) \vdash mexp \Rightarrow \exists N'.(\mathcal{M}', c) \wedge M' \models \mathcal{M}' \end{array}}{M \models_S \mathcal{M}}$$

Figure 3.12: Type consistency logical relation.

(i) if M is a non-parameterised module, then $\mathcal{M}$ is also a non-parameterised module interpretation for some interpretation environment $\mathcal{E}$. That is, if $M = E$, and $M \models \mathcal{M}$, then $\exists \mathcal{E}.\mathcal{M} = \mathcal{E}$.

(ii) if M is a functor, then $\mathcal{M}$ is a functor closure, which is consistent with the module M. That is, if $M = \forall T.(E, \exists T'.M')$ and $M \models \mathcal{M}$, then $\mathcal{M} = (\mathcal{E}, \forall T.(E, \Sigma), \lambda mid \Rightarrow mexp)$, and the corresponding consistency condition holds (we write $\implies$ for implication):

$$\begin{array}{l} \forall \mathcal{E}', E \models \mathcal{E}' \implies \\ \quad \exists N', \mathcal{M}', c.\ \ ((\mathcal{E} + \{mid \mapsto \mathcal{E}'\}) \vdash mexp \Rightarrow \exists N'.(\mathcal{M}', c)) \wedge M' \models \mathcal{M}' \end{array}$$

Lemma 3.3.5 (Type consistency to erasure)**.** *If some semantic object E is consistent with some interpretation environment $\mathcal{E}$, then erasure of $\mathcal{E}$ gives us exactly E.*

That is, if $E \models \mathcal{E}$ then $\overline{\mathcal{E}} = E$.

Lemma 3.3.6 (Consistency and environment extension (cf. Lemma 3.2.1))**.** *If some semantic object E is consistent with some interpretation environment $\mathcal{E}$, and some module M is consistent with some module interpretation $\mathcal{M}$, then for some module identifier mid, the extension of E with a mapping $mid \mapsto M$ is consistent with $\mathcal{E}$ extended with the mapping $mid \mapsto \mathcal{M}$.*

That is, if $E \models \mathcal{E}$ and $M \models \mathcal{M}$ then $(E + \{mid \mapsto M\}) \models (\mathcal{E} + \{mid \mapsto \mathcal{M}\})$

Lemma 3.3.7 (Consistency of environment modification)**.** *For any semantic objects E_1, E_2, and interpretation environments $\mathcal{E}_1$, $\mathcal{E}_2$, if $E_1 \models \mathcal{E}_1$ and $E_2 \models \mathcal{E}_2$, then $(E_1 + E_2) \models (\mathcal{E}_1 + \mathcal{E}_2)$.*

Lemma 3.3.8 (Termination of the declarations compilation)**.** *For any environments E and E', interpretation environment $\mathcal{E}$, and declaration dec, if $E \vdash dec : E'$, and $E \models \mathcal{E}$, then there exist N, $\mathcal{E}'$, c, s.t. $\mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c)$ and $E' \models \mathcal{E}'$.*

Lemma 3.3.9 (Consistency of filtering)**.** *For any environments E' and E, interpretation environment $\mathcal{E}'$, if $E' \succ E$, and $E' \models \mathcal{E}'$ then there exist an interpretation environment $\mathcal{E}$, s.t. $\mathcal{E}' :: E \Rightarrow \mathcal{E}$ and $E \models \mathcal{E}$.*

The idea of the proof of Lemma 3.3.9 is to *construct* the environment $\mathcal{E}$. The definition of the filtering relation does not define a particular algorithm for constructing a filtered environment, but instead serves as a specification. One can define a function that actually implements filtering and show that this function satisfies the specification give by the definition in Figure 3.10. Essentially, we take this approach in our Coq development.

Theorem 3.3.1. *(Static Interpretation Normalization)*
If $E \vdash \overset{\mathcal{D}_{mexp}}{mexp} : \exists T.M$ and $E \overset{\mathcal{C}}{\models} \mathcal{E}$ then there exists N, $\mathcal{M}$, c such that

$$\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{M}, c), \text{ and } M \models \mathcal{M}$$

and (mutually)
if $E \vdash \overset{\mathcal{D}_{mdec}}{mdec} : \exists T.E'$ and $E \overset{\mathcal{C}}{\models} \mathcal{E}$ then there exists N, $\mathcal{E}'$, c such that

$$\mathcal{E} \vdash mdec \Rightarrow \exists N.(\mathcal{E}', c), \text{ and } E' \models \mathcal{E}'$$

Proof. The proof sketch presented here is following our development in the Coq proof assistant. We omit some details, since they worked out fully in the formalisation, and we aim here to show the overall structure of the proof and refer to auxiliary lemmas, which are crucial to use in particular cases. We also emphasise the simplifying assumptions we made for our development.

The proof proceeds by mutual induction over derivations $\mathcal{D}_{mdec}$ and $\mathcal{D}_{mexp}$.

Case 1 ($\dfrac{E \vdash \overset{\mathcal{D}_{mdec}}{mdec} : \Sigma}{E \vdash \{\ mdec\ \} : \Sigma}$):

By induction hypothesis on $\mathcal{D}_{mdec}$ with $\mathcal{C}$, we get N, $\mathcal{E}$ c, and we take $\mathcal{M} = \mathcal{E}$ and apply Rule (51).

Case 2 ($\dfrac{E(mid) = E'}{E \vdash mid : E'}$):
By lemma 3.3.3, we get $\mathcal{M}'$, s.t. $M \models \mathcal{M}'$. Take $N = \emptyset$, $\mathcal{M} = \mathcal{M}'$, and $c = \epsilon$. We get the derivation for the interpretation by applying Rule (47), and we already have $M \models \mathcal{M}'$ from having applied Lemma 3.3.3 earlier.

Case 3 ($\dfrac{E \vdash \overset{\mathcal{D}_{mexp}}{mexp} : \exists T.E' \qquad E'(mid) = E''}{E \vdash mexp\,.\,mid : \exists T.E''}$):
By induction hypothesis on $\mathcal{D}_{mexp}$ with $\mathcal{C}$, we get N, $\mathcal{M}$, c, s.t. $\mathcal{E} \vdash mexp \Rightarrow \exists N.(\mathcal{M}, c)$, and $M \models \mathcal{M}$, where M is a non-parameterised module E'. By Lemma 3.3.4, we get $E' \models \mathcal{E}'$ for some $\mathcal{E}'$.

Now, by applying the lookup consistency lemma (Lemma 3.3.3), we get all the required pieces to construct a derivation of static interpretation using Rule (48).

We already have $M \models \mathcal{M}'$ from the induction hypothesis.

Case 4 ($\dfrac{E \vdash mty : \exists\emptyset.E' \qquad E + \{mid \mapsto E'\} \overset{\mathcal{D}_{mexp}}{\vdash} mexp : \Sigma \qquad F = \forall\emptyset.(E', \Sigma)}{E \vdash \textbf{funct}\ mid\ \textbf{:}\ mty \Rightarrow mexp : \exists\emptyset.F}$):

Since we are in the simplified setting here, we instantiate universally and existentially quantified variables with empty sets in the rule.

Take $N = \emptyset$, $\mathcal{M} = (\mathcal{E}, F, \lambda \Rightarrow mexp)$ (a functor closure), $c = \epsilon$. From assumptions we have $E \overset{\mathcal{C}}{\models} \mathcal{E}$. From $\mathcal{C}$ with the consistency to erasure lemma (Lemma 3.3.5), we get $\overline{\mathcal{E}} = E$. Now, we apply Rule (49).

Proving $M \models \mathcal{M}$ in this case requires some work. We get a required premise for the corresponding rule of the consistency relation from the induction hypothesis with the consistency and the environment extension lemma (Lemma 3.3.6) (cf. Theorem 3.2.1, Case TY-LAM).

Case 5 ($\dfrac{E_0 \overset{\mathcal{D}_{mexp}}{\vdash} mexp : \exists\emptyset.E \qquad E_0(longmid) \geq (E', \exists\emptyset.E'') \qquad E \succ E'}{E_0 \vdash longmid\ (\ mexp\ ) : \exists\emptyset.E''}$):

This is the most complicated case of our proof. Again, we consider a simplified setting where sets of variables in binding positions are empty. Moreover, instead of instantiation $E_0(longmid) \geq (E', \exists\emptyset.E'')$ we use simple equality $E_0(longmid) = (E', \exists\emptyset.E'')$.

First, we use the lookup consistency lemma (Lemma 3.3.3) to get $\mathcal{M}_0$, s.t. $\mathcal{E}(longid) = \mathcal{M}_0$ and $M \models \mathcal{M}_0$.

From the uniformness lemma (Lemma 3.3.4(ii)) we know that $\mathcal{M}_0$ is a functor closure $(\mathcal{E}_1, M, \lambda mid \Rightarrow mexp_1)$ for some $\mathcal{E}_1$ and $mexp_1$. From this lemma we also get the consistency condition; we will denote it as $\mathcal{H}_c$ and use it later.

Next, from the induction hypothesis on $\mathcal{D}_{mexp}$ with $\mathcal{C}$, we get N_1, $\mathcal{M}_1$, c_1, s.t. $\mathcal{E} \vdash mexp \Rightarrow \exists N_1.(\mathcal{M}_1, c_1)$ and $E \models \mathcal{M}_1$. From the uniformness lemma 3.3.4(i) we know that $\mathcal{M}_1$ is also some non-parameterised module interpretation $\mathcal{E}'$, and so $E \overset{\mathcal{H}_e}{\models} \mathcal{E}'$.

From the consistency of filtering lemma (Lemma 3.3.9), we get $\mathcal{E}''$, s.t. $\mathcal{E}' :: E' \Rightarrow \mathcal{E}''$ and $E' \overset{\mathcal{C}'}{\models} \mathcal{E}''$. We can use the consistency condition $\mathcal{H}_c$ with $\mathcal{C}'$ to get N_2, $\mathcal{M}_2$, c_2, s.t. $(\mathcal{E}' + \{mid \mapsto \mathcal{E}''\}) \vdash mexp_1 \Rightarrow \exists N_2.(\mathcal{M}_2, c_2)$ and $E'' \overset{\mathcal{C}''}{\models} \mathcal{M}_2$.

Now, we take $N = N_1 \cup N_2$, $\mathcal{M} = \mathcal{M}_2$, $c = c_1; c_2$ in our goal. We construct the required derivation by applying Rule (50).

The consistency part follows from $\mathcal{C}''$. (cf. Theorem 3.2.1, Case TY-APP. Although in the present proof we do not have two induction hypotheses and instead of the first induction hypothesis we look up for the $longid$).

Case 6 ($\dfrac{mdec = dec \qquad E \vdash dec : E'}{E \vdash mdec : E'}$):

By the consistency to erasure lemma (Lemma 3.3.5) with $\mathcal{C}$, we get $\overline{\mathcal{E}} = E$. By the termination of the declarations compilation lemma 3.3.8, we get N,$\mathcal{E}'$,c, s.t.

$$\mathcal{E} \vdash dec \Rightarrow \exists N.(\mathcal{E}', c) \tag{3.58}$$

$$E \models \mathcal{E}' \tag{3.59}$$

We construct the required derivation by applying Rule (51) with (3.58), and we have $E \models \mathcal{E}'$ from (3.59).

Case 7 ($\dfrac{E \vdash ty : \tau}{E \vdash \textbf{type}\ tid = ty : \{tid \mapsto \tau\}}$):
By the consistency to erasure lemma 3.3.5 with $\mathcal{C}$, we get $\overline{\mathcal{E}} = E$. Take $N = \emptyset$, $\mathcal{E}' = \{tid \mapsto \tau\}$, $c = \epsilon$.
We construct the required derivation by applying Rule (52). We get $\{tid \mapsto \tau\} \models \{tid \mapsto \tau\}$, since the environments are equal.

Case 8 ($\dfrac{E \vdash mexp : \exists T.M}{E \vdash \textbf{module}\ mid = mexp : \exists T.\{mid \mapsto M\}}$):
By induction hypothesis on $\mathcal{D}_{mexp}$ with $\mathcal{C}$, we get N, $\mathcal{M}$, c, s.t. $\mathcal{E} \vdash mexp \Rightarrow \exists N.(\mathcal{M}, c)$ and $M \models \mathcal{M}$. Take $\mathcal{E}' = \{mid \mapsto \mathcal{M}\}$. We construct the required derivation by applying Rule (53).

We get $\{mid \mapsto M\} \models \{tid \mapsto \mathcal{M}\}$ by the consistency and environment extension lemma (Lemma 3.3.6) with $E = \mathcal{E} = \{\}$, and the fact that $\{\} \models \{\}$.

Case 9 ($\dfrac{E \vdash \overset{\mathcal{D}_{mexp}}{mexp} : \Sigma}{E \vdash \textbf{open}\ mexp : \Sigma}$):
By induction hypothesis on $\mathcal{D}_{mexp}$ with $\mathcal{C}$, we get required pieces to construct the derivation using Rule (55).

Case 10 ($\dfrac{E \vdash mty : \Sigma}{E \vdash \textbf{module type}\ mtid = mty : \exists\emptyset.\{mtid \mapsto \Sigma\}}$):
Take $N = \emptyset$, $\mathcal{E}' = \{mtid \mapsto \Sigma\}$, $c = \epsilon$. By the consistency to erasure lemma (Lemma 3.3.5) with $\mathcal{C}$, we get $\overline{\mathcal{E}} = E$. We construct the required derivation by applying Rule (54).

We get $\{mtid \mapsto \Sigma\} \models \{mtid \mapsto \Sigma\}$, since the environments are equal.

Case 11 ($\dfrac{T_1 \cap (\text{tvs}(E) \cup T_2) = \emptyset \qquad E \vdash \overset{\mathcal{D}_{mdec_1}}{mdec_1} : \exists T_1.E_1 \qquad E + E_1 \vdash \overset{\mathcal{D}_{mdec_2}}{mdec_2} : \exists T_2.E_2}{E \vdash mdec_1\ mdec_2 : \exists(T_1 \cup T_2).(E_1 + E_2)}$):
In this case we do not make simplifying assumptions and leave sets T_1 and T_2 non-empty along with corresponding sets in the static interpretation rule to show where we need to apply the bound variable convention (see Remark 3.5.5).

By induction hypotheses on $\mathcal{D}_{mdec_1}$ and $\mathcal{D}_{mdec_2}$ with $\mathcal{C}$, we get

$$N_1, \mathcal{E}_1, c_1, \text{ s.t. } \mathcal{E} \vdash mdec_1 \Rightarrow \exists N_1.(\mathcal{E}_1, c_1) \text{ and } E_1 \models \mathcal{E}_1,$$
$$N_2, \mathcal{E}_2, c_2, \text{ s.t. } \mathcal{E} \vdash mdec_2 \Rightarrow \exists N_2.(\mathcal{E}_2, c_2) \text{ and } E_2 \models \mathcal{E}_2.$$

We can always α-rename $\exists N_1.(\mathcal{E}_1, c_1)$ and $\exists N_2.(\mathcal{E}_2, c_2)$ in such a way that N_1 and N_2 will satisfy the disjointness condition in Rule (56).
Take $N = N_1 \cup N_2$, $\mathcal{E}' = \mathcal{E}_1 + \mathcal{E}_2$, $c = c_1; c_2$. We construct the required derivation by applying Rule (56).
We get $(E_1 + E_2) \models (\mathcal{E}_1 + \mathcal{E}_2)$ from the consistency of environment modification lemma (Lemma 3.3.7).

Case 12 ($\frac{}{E \vdash \epsilon : \{\}}$):
Take $N = \emptyset$, $\mathcal{E}' = \{\}$, $c = \epsilon$. We construct the required derivation by applying Rule (57).

We get $\{\} \models \{\}$ trivially.

■

3.4 Variable Binding and Nominal Techniques

In this section we are going to give a general introduction to nominal sets [GP02, Pit13], and we give motivations why we have decided to use nominal techniques in our formalisation. We outline definitions of relevant concepts such as atoms, permutation of atoms, freshness relation, equivariance, and so on.

Such aspects as freshness of variables and α-renaming are often left implicit in pen-and-paper formalisations, but in proof assistants one has to pay attention to all the details related to these aspects. Moreover, it is a well-known fact, that an implementation of variable binding conventions in proof assistants often requires significant efforts. There are multiple ways to approach variable binding and related issues in proof assistants, such as de Bruijn indices [dB72], locally named and locally nameless representation [MP93, Gor94, Cha12], parametric higher-order abstract syntax [Chl08], and so on. One of the goals of our formalisation of the module system in Coq is to keep it close to the presentation given in Section 3.3. Since this representation uses names, changing to another way of treating bound variables (like de Bruijn indices) will require us to deviate from this representation. Moreover, since our setting is different from well-studied systems such as the lambda-calculus (mutual inductive definitions, sets of variables in binding positions), we would like to use a first-order representation as a more flexible approach applicable for various structures with variable binding. The approach based on nominal sets seems to be a good fit in our situation.

Broadly speaking, the theory of nominal sets is a theory of names involved in different data structures, covering such aspects as variable binding, scope, and freshness of variables. Nominal sets offer a solid foundation for expressing *independence* of data structures on the particular choices of bound variables [Pit16]. The theory of nominal sets based on the idea of permutations of variables and notion of finite support.[2] The theory gives a uniform approach to deal with bound variables and allows for generalisation of binders to various structures [Clo13] We are interested in the particular generalisation where one can bind a set of variables at once. Moreover, since nominal sets offer a uniform approach it can be applied to various structures even if they are quite different from well-studied systems such as the lambda calculus. Our goal is to define a nominal set semantic objects (see Section 3.3.1) and use nominal reasoning techniques in our formalisation.

[2]Similarly to the development in Agda [Cho15] we do not consider the notion of *unique* smallest finite support. This allows us to develop required notions constructively. For the detailed discussion about nominal sets in constructive set theory see [Swa17].

First, we give basic definitions from the theory of nominal sets and show how these notions can be implemented in the proof assistant Coq (see Section 3.5.3).

Definition 3.4.1 (Atoms)**.** Let $\mathbb{A}$ be a set of *atoms*, if it satisfies two axioms:

- elements of $\mathbb{A}$ has decidable equality, i.e. $\forall a, b \in \mathbb{A}.(a = b) \vee (a \neq b)$;
- $\mathbb{A}$ is countably infinite, i.e. for any finite set of atoms F, $\exists b \in \mathbb{A}.b \notin F$.

Definition 3.4.2 (Permutation)**.** We write $Perm\ \mathbb{A}$ for the set of all finitely supported permutations $\pi : \mathbb{A} \to \mathbb{A}$. That is, π is a bijection on $\mathbb{A}$ with the finite support property: there exists a finite set of atoms F, s.t. $\forall a \notin F.\pi a = a$. We call F the *support* of the permutation π.

Intuitively, Definition 3.4.2 says that π is a bijection which permutes only elements of some finite subset of $\mathbb{A}$ and leaves other elements outside of F untouched.

The set $Perm\ \mathbb{A}$ has a group structure with function composition as group operation, the identity function (treated as a permutation) id as a neutral element, and the inverse function (since permutations are bijections) as a group inverse.

Definition 3.4.3 (Transposition)**.** There is an elementary permutation which we call a transposition:

$$(a\ b)c :\equiv \begin{cases} a, \text{if } b = c \\ b, \text{if } a = c \\ c, \text{otherwise} \end{cases}$$

The transposition $(a\ b)$, being a permutation, has support set $\{a, b\}$. Any permutation can be non-uniquely factored through a sequence of transpositions. Transpositions are involutions, that is, being applied twice they "cancel out" each other: $(a\ b) \circ (a\ b) = id$. Transpositions can be generalised from single atoms to n-tuples of atoms.

Definition 3.4.4 (Generalised transposition)**.** Let $\overrightarrow{a} :\equiv (a_1, a_2, \ldots, a_n)$, $\overrightarrow{b} :\equiv (b_1, b_2, \ldots, b_n)$, then we define a generalised transposition of $\overrightarrow{a}$ and $\overrightarrow{b}$ as a composition of simple transpositions:

$$(\overrightarrow{a}\ \overrightarrow{b}) :\equiv (a_1\ b_1) \circ (a_2\ b_2) \circ \cdots \circ (a_n\ b_n)$$

Definition 3.4.5 (Action)**.** We define the *action* of a $Perm\ \mathbb{A}$ on a set X as a binary operation $- \cdot - : Perm\ \mathbb{A} \to X \to X$ with the following properties:

- for any $x \in X$, $id \cdot x = x$
- for any $x \in X$, $\pi_1, \pi_2 \in Perm\ \mathbb{A}$, $\pi_1 \cdot (\pi_2 \cdot x) = (\pi_1 \circ \pi_2) \cdot x$

An action of π on some x basically allows one to "apply" a permutation to occurrences of atoms "inside" x.

Definition 3.4.6 (Support)**.** Consider a finite subset F of atoms $\mathbb{A}$, and a subgroup of permutations $Perm_F\ \mathbb{A}$ satisfying $\forall a \in F.\pi a = a$. We say that F *supports* an element $x \in X$ if for any permutation π in $Perm_F\ \mathbb{A}$ we have $\pi \cdot x = x$. We write *supp* x for the support of x.

We can also give an alternative characterisation of support using transpositions: if for any two elements *outside* of the a finite subset F of $\mathbb{A}$, it holds that if $(a\ b) \cdot x = x$ for an element $x \in X$, then F supports x.

The characterisation in terms of transposition gives an intuitive understanding of support as the finite set of atoms that may occur "inside" x, since if we pick two elements outside of the support of x, then the action of the transposition of these elements will have no effect on x.[3]

Definition 3.4.7 (Nominal set)**.** A nominal set $\mathbf{X}$ is a set X together with the action $- \cdot - : \mathbb{A} \to X \to X$, such that each element of $x \in X$ there exists a finite subset S of atoms $\mathbb{A}$ supporting x.

Definition 3.4.8 (Freshness)**.** We say that an atom $a \in \mathbb{A}$ is fresh for an element of a nominal set $x \in X$ if there exists some finite support S of x such that $a \notin S$.

Definition 3.4.8 is the usual notion of freshness for a single atom. We can also define a generalised notion of freshness following [Clo13], which will be useful in the context of our formalisation.

Definition 3.4.9 (Generalised freshness)**.** We say that the set of atoms of $x \in X$ is fresh for the set of atoms of $y \in Y$ (we also can say that y is fresh for x) if for some finite support S_1 of x and some finite support S_2 of y the following holds: $S_1 \cap S_2 = \emptyset$. That is, S_1 and S_2 are disjoint.

We will use the following notation for the freshness relation: $x\#y$, which one can read as "x is fresh for y". We will use the same notation for both freshness and generalised freshness relations, and say explicitly, which notation is used in case of ambiguity.

Remark 3.4.1. In our Coq development, each implementation of a nominal set module type contains a *supp* function that accepts an element of the nominal set and returns its finite support. Since we are not requiring for a nominal set to have the smallest support, we cannot prove some general properties as we would be able to do with the smallest support requirement. For example, we cannot show that any *supp* function is equivariant, because this is not true in general. Instead, for all the definitions of nominal sets used in our formalisation we choose the definitions of *supp* in such a way that they satisfy the equivariance property. For every such definition of *supp* we provide a separate proof of equivariance. The same applies to the freshness relation (since it depends on *supp*): for each nominal set in our formalisation we have to show that it is equivariant.

[3] Since we are not defining a *smallest* support, *supp* x may contain more elements than the set of all atoms occurring in x

Example 3.2: The set of lambda terms *Lam* (see (3.2) in Section 3.2, assuming that variables $v \in \mathbb{A}$) is a nominal set **Lam** with the action:

$$\begin{aligned} \pi \cdot i &:\equiv i \\ \pi \cdot v &:\equiv \pi v \\ \pi \cdot (\lambda x.e) &:\equiv \lambda(\pi x.\pi \cdot e) \\ \pi \cdot (e_1 e_2) &:\equiv (\pi \cdot e_1)(\pi \cdot e_2) \end{aligned}$$

The action of π is applied uniformly to all occurrences of variables: binding, bound and free. Support is defined as follows:

$$\begin{aligned} \mathit{supp}\ i &:\equiv \emptyset \\ \mathit{supp}\ v &:\equiv \{v\} \\ \mathit{supp}\ (\lambda x.e) &:\equiv \{x\} \cup \mathit{supp}\ e) \\ \mathit{supp}\ (e_1 e_2) &:\equiv (\mathit{supp}\ e_1) \cup (\mathit{supp}\ e_2) \end{aligned}$$

That is, the support of a lambda term is the set of *all* variables occurring in the term.

Example 3.3: The set of atoms $\mathbb{A}$ is a nominal set **A**. In this case the action on $a : \mathbb{A}$ is just an application of a permutation:

$$\pi \cdot a :\equiv \pi a$$

The support of $a : \mathbb{A}$ is the singleton set $\{a\}$.

Example 3.4: The set of finite sets of atoms $\mathrm{Fin}_{\mathbb{A}}$ is a nominal set $\mathbf{Fin}_{\mathbb{A}}$. In this case the action of a permutation π on $X \in \mathrm{Fin}_{\mathbb{A}}$ can be defined as an application of π to each element of the set:

$$\pi \cdot X :\equiv \{\pi a \mid a \in X\}$$

The support of $X \in \mathrm{Fin}_{\mathbb{A}}$ is the finite set X itself.

As a running example, which is relevant in our setting of a module system formalisation, let us consider a simplified version of semantic objects (Figure 3.13). We consider only module specifications with a "flat" environment structure, avoiding mutual definitions for simplicity.

Example 3.5: Types $\mathcal{T}$ form a nominal set **T** with the action given by

$$\begin{aligned} \pi \cdot (\tau_1 \to \tau_2) &:\equiv (\pi \cdot \tau_1) \to (\pi \cdot \tau_2) \\ \pi \cdot t &:\equiv \pi t \end{aligned}$$

And support given by

$$\begin{aligned} \mathit{supp}\ (\tau_1 \to \tau_2) &:\equiv (\mathit{supp}\ \tau_1) \cup (\mathit{supp}\ \tau_2) \\ \mathit{supp}\ t &:\equiv \{t\} \end{aligned}$$

$$t \in \mathbb{A}$$
$$\tau \in \mathcal{T} ::= t \mid \tau_1 \to \tau_2$$
$$E \in \text{Env} = tid \xrightarrow{\text{fin}} \tau$$
$$\Sigma = \exists T.E \in \text{MTy} = \text{Fin}_{\mathbb{A}}(\text{TSet}) \times \text{Env}$$

Type Expressions

$$\boxed{E \vdash ty : \tau}$$

$$ty ::= tid \mid \texttt{Arr}\ ty_1\ ty_2$$

$$\frac{E(tid) = \tau}{E \vdash tid : \tau}\ \text{TY-TID} \qquad \frac{E \vdash ty_i : \tau_i \qquad i = [1,2]}{E \vdash \texttt{Arr}\ ty_1\ ty_2 : \tau_1 \to \tau_2}\ \text{TY-ARR}$$

Module Specifications

$$\boxed{E \vdash spec : \exists T.E'}$$

$$spec ::= \textbf{type}\ tid \mid \textbf{type}\ tid = ty \mid spec_1; spec_2$$

$$\frac{}{E \vdash \textbf{type}\ tid : \exists\{t\}.\{tid \mapsto t\}}\ \text{SPEC-TYPE}$$

$$\frac{E \vdash ty : \tau}{E \vdash \textbf{type}\ tid = \tau : \exists\{\}.\{tid \mapsto \tau\}}\ \text{SPEC-TYPE-ASSGN}$$

$$\frac{E \vdash spec_1 : \exists T_1.E_1 \qquad E + E_1 \vdash spec_2 : \exists T_2.E_2 \qquad T_1 \# (E, T_2) \qquad \text{Dom}\ E_1 \cap \text{Dom}\ E_2 = \emptyset}{E \vdash spec_1; spec_2 : \exists(T_1 \cup T_2).(E_1 + E_2)}\ \text{SPEC-SEQ}$$

Figure 3.13: Simplified semantic objects and elaboration rules for type expressions and module specifications.

Example 3.6: Environments (or finite maps) Env, mapping type identifiers to types τ form a nominal set **Env** with the action given by

$$\pi \cdot e :\equiv \{(k \mapsto \pi \cdot_{\mathcal{T}} \tau) \mid (k \mapsto \tau) \in e\}$$

We get a new finite map by applying the permutation action to every element in the codomain. Subscript $\mathcal{T}$ on the action of permutation π means that we apply the action on nominal set **T**.

The support is defined as follows:

$$supp\ e :\equiv \bigcup(\{supp\ \tau \mid (k \mapsto \tau) \in e\})$$

That is, the support of e is a union of sets given by supports of all elements in the codomain.

From the examples above we can define a nominal set for the type of module signatures Σ, which are pairs of a finite set of existentially quantified variables T and an environment E where these variables may occur.

Example 3.7: We define a nominal set **MTy** of module signatures MTy equipped with the action defined in terms of actions on nominal sets $\mathbf{Fin}_{\mathbb{A}}$ and **Env**

$$\pi \cdot (\exists T.E) :\equiv \exists(\pi \cdot_{\mathrm{Fin}_{\mathbb{A}}} T).(\pi \cdot_{\mathrm{Env}} E)$$

The same applies to the definition of support:

$$supp\ (\exists T.E) :\equiv (supp_{\mathrm{Fin}_{\mathbb{A}}}\ T) \cup (supp_{\mathrm{Env}}\ E)$$

From now on we will omit explicit annotations on the action of a permutation and on the support, since it can be determined from the type of argument.

An important notion in the theory of nominal sets is the notion of *equivariance*.

Definition 3.4.10 (Equivariant functions)**.** For two nominal sets **X** and **Y**, a function between the carrier sets $f : X \to Y$ is called equivariant if it has the following property:

$$\forall x \in X, \pi \in Perm\ \mathbb{A}.\ f\ (\pi \cdot x) = \pi \cdot (f\ x)$$

That is, f preserves the action of a permutation π.

Definition 3.4.11 (The category of nominal sets)**.** Nominal sets form a category **Nom** where objects are nominal sets and morphisms are equivariant functions.

The **Nom** category is Cartesian closed. Particularly, our example of **MTy** being a nominal set boils down to the fact that **Nom** has finite products.

The same way as we defined the notion of equivariant functions above (definition 3.4.10), one can define the notion of equivariant relation.

Definition 3.4.12 (Equivariant relations)**.** For two nominal sets **X** and **Y** a relation $\mathcal{R} \subseteq X \times Y$ is equivariant if

$$\forall x \in X, y \in Y, \pi \in Perm\ \mathbb{A}.\ x\mathcal{R}y \Rightarrow (\pi \cdot x)\mathcal{R}(\pi \cdot y)$$

Lemma 3.4.1. *The following operations are equivariant:*

- *the union of two finite sets X and Y:*

$$\pi \cdot X \cup Y = (\pi \cdot X) \cup (\pi \cdot Y)$$

- *the intersection of two finite sets X and Y:*

$$\pi \cdot X \cap Y = (\pi \cdot X) \cap (\pi \cdot Y)$$

- *the modification of environment E_1 by environment E_2 (see Definition 3.3.1):*

$$\pi \cdot (E_1 + E_2) = (\pi \cdot E_1) + (\pi \cdot E_2)$$

Lemma 3.4.2. *The generalised freshness relation for nominal sets $\mathbf{Fin}_{\mathbb{A}}$ and* **Env***, with finite supports for the elements given by the supp function defined in Examples 3.4 and 3.6, is equivariant. That is, for elements x and y of these nominal sets we have the following:*

$$x \# y \Rightarrow (\pi \cdot x) \# (\pi \cdot y)$$

Remark 3.4.2. We write $X = Y$, where X and Y are two finite sets, for extensional equality of sets, i.e. $x \in X \iff y \in Y$. The same holds for environments.

Let us consider proofs of equivariance of elaboration relations (see Figure 3.13. We present detailed proofs of two lemmas below.

Lemma 3.4.3. *The elaboration relation for type expressions $E \vdash ty : \tau$ (see Figure 3.13) is equivariant. That is, for any π, E, ty and τ*

$$E \vdash \overset{\mathcal{T}}{ty} : \tau \Rightarrow (\pi \cdot E) \vdash ty : (\pi \cdot \tau)$$

Proof. By induction on derivation $\mathcal{T}$.

Case 1: $\mathcal{T} = \dfrac{E(tid) = \tau}{E \vdash tid : \tau}$ TY-TID

From the assumption $E(k) = v$, and the definition of the action on environments, we get $(\pi \cdot E)(tid) = \pi \cdot \tau$.

Now, we can construct the required derivation

$$\frac{(\pi \cdot E)(tid) = \pi \cdot \tau}{(\pi \cdot E) \vdash tid : \pi \cdot \tau} \text{ TY-TID}$$

Case 2: $\mathcal{T} = \dfrac{E \vdash \overset{\mathcal{T}_i}{ty_i} : \tau_i \qquad i = [1,2]}{E \vdash \texttt{Arr}\ ty_1\ ty_2 : \tau_1 \to \tau_2}$ TY-ARR

By induction hypotheses, we get two derivations:
$\mathcal{T}_1' = (\pi \cdot E) \vdash tid : \pi \cdot \tau_1$
$\mathcal{T}_2' = (\pi \cdot E) \vdash tid : \pi \cdot \tau_2$
We construct the required derivation from $\mathcal{T}_1'$ and $\mathcal{T}_2'$ using the rule TY-ARR.

■

Lemma 3.4.4. *The elaboration relation for module specifications $E \vdash spec : \exists T.E'$ (see Figure 3.13) is equivariant. That is, for any π, E, E', spec and T*

$$E \vdash \overset{\mathcal{T}}{spec} : \exists T.E' \Rightarrow (\pi \cdot E) \vdash spec : \exists(\pi \cdot T).(\pi \cdot E)$$

Proof. By induction on the derivation $\mathcal{E}$.

Case 1: $\mathcal{E} = \dfrac{}{E \vdash \textbf{type}\ tid : \exists\{t\}.\{tid \mapsto t\}}$ SPEC-TYPE

We use the fact that $\pi \cdot \{x\} = \{\pi \cdot x\}$. The same holds for the singleton environment $\pi \cdot \{k \mapsto v\} = \{k \mapsto \pi \cdot x\}$. We construct the required derivation using these facts and the rule SPEC-TYPE.

Case 2: $\mathcal{E} = \dfrac{E \vdash \overset{\mathcal{T}}{ty} : \tau}{E \vdash \textbf{type}\ tid = \tau : \exists\{\}.\{tid \mapsto \tau\}}$ SPEC-TYPE-ASSGN.

We know that $\pi \cdot \{\} = \{\}$, and $\pi \cdot \{k \mapsto v\} = \{k \mapsto \pi \cdot x\}$. From equivariance of the elaboration relation for type expressions (Lemma 3.4.3) with $\mathcal{T}$, we get $\mathcal{T}' = (\pi \cdot E) \vdash ty : (\pi \cdot \tau)$.

Now, we can construct the required derivation using the rule SPEC-TYPE-ASSGN.

Case 3: $\mathcal{E} = \dfrac{E \vdash \overset{\mathcal{E}_1}{spec_1} : \exists T_1.E_1 \qquad E + E_1 \vdash \overset{\mathcal{E}_2}{spec_2} : \exists T_2.E_2 \qquad T_1 \# (E, T_2) \qquad \text{Dom } E_1 \cap \text{Dom } E_2 = \emptyset}{E \vdash spec_1; spec_2 : \exists(T_1 \cup T_2).(E_1 + E_2)}$ SPEC-SEQ

We have to construct a derivation for

$$(\pi \cdot E) \vdash spec_1; spec_2 : \pi \cdot (\exists(T_1 \cup T_2).(E_1 + E_2))$$

Using the definition of the action of π on module signatures MTy, and the facts that the union operation on sets and the modification operation on environments are equivariant, we transform our goal to the following form:

$$(\pi \cdot E) \vdash spec_1; spec_2 : (\exists(\pi \cdot T_1 \cup \pi \cdot T_2).(\pi \cdot E_1 + \pi \cdot E_2))$$

Next, by equivariance of the freshness relation for nominal sets $\mathbf{Fin}_{\mathbb{A}}$ and **Env** (Lemma 3.4.2), we get

$$(\pi \cdot T_1) \# (\pi \cdot E, \pi \cdot T_2) \tag{3.60}$$

we also get

$$\text{Dom } (\pi \cdot E_1) \cap \text{Dom } (\pi \cdot E_2) = \emptyset \tag{3.61}$$

since the permutation action on environments does not affect the domain. By induction hypothesis on $\mathcal{E}_1$, we get

$$\mathcal{E}_1' = (\pi \cdot E) \vdash spec_1 : \exists(\pi \cdot T_1).(\pi \cdot E_1)$$

By induction hypothesis on $\mathcal{E}_2$, we get

$$\mathcal{E}_2' = (\pi \cdot E + E_1) \vdash spec_2 : (\pi \cdot T_2).(\pi \cdot E_2)$$

Using the equivariance of the modification operation on environments, we get

$$\mathcal{E}_2'' = (\pi \cdot E + \pi \cdot E_1) \vdash spec_2 : (\pi \cdot T_2).(\pi \cdot E_2)$$

We get the required derivation using Rule SPEC-SEQ with $\mathcal{E}_1'$, $\mathcal{E}_2''$, (3.60), and (3.61).

We observe, that the proof of equivariance boils down to showing that operations and relations involved in the relation definition are themselves equivariant (see [Pit13, Section 7.3]).

■

The notion of α-equivalence is often used to say that two terms are equal "up to" renaming of bound variables. Using α-equivalence one can express independence of particular choices for names of bound variables. Using nominal techniques, we can give a definition of α-equivalence just in terms of freshness and permutation of variables.

Example 3.8: First, let us consider a traditional setting of the lambda calculus. The following inductively defined relation specifies α-equivalence of lambda terms [GP02]:

$$\frac{}{a =_\alpha a} \qquad \frac{t_1 =_\alpha t_1' \qquad t_2 =_\alpha t_2'}{t_1 t_2 =_\alpha t_1' t_2'}$$

$$\frac{(a_1\ b) \cdot t_1 =_\alpha (a_2\ b) \cdot t_2 \qquad b\#(a_1, a_2, t_1, t_2)}{\lambda a_1.t_1 =_\alpha \lambda a_2.t_2}$$

It has been shown in [GP02] that the definition of α-equivalence in terms of permutations corresponds to the usual notion of α-equivalence on lambda terms. Moreover, the support of $t \in Lam/_{=_\alpha}$ (where $Lam/_{=_\alpha}$ is a quotient set) is exactly the set of free variables of t.

For our running example of simplified semantic objects given in Figure 3.13, we could consider a characterisation of α-equivalence in terms of generalised transpositions (Definition 3.4.4). In this case, we would have to fix some (arbitrary) order for the sets of variables in binding positions. We write $\overrightarrow{T}$ for the set of variables T ordered according to some fixed order. We can define an α-equivalence relation on Env as follows:

$$\frac{(\overrightarrow{T_1}\ \overrightarrow{T}) \cdot E_1 \qquad (\overrightarrow{T_2}\ \overrightarrow{T}) \cdot E_2 \qquad T\#(T_1, T_2, E_1, E_2) \qquad \mathit{card}\ T_1 = \mathit{card}\ T_2 = \mathit{card}\ T}{\exists T_1.E_1 =_\alpha \exists T_2.E_2}$$

In this definition we assume that cardinalities of the finite sets of variables T_1, T_2, and T are equal.

Unfortunately, since we have to fix some arbitrary order on sets of variables, some properties of generalised transpositions does not play well with the idea of sets of variables to be unordered. For example, if we have sets $T = T_1 \cup T_2$ and $T' = T_1' \cup T_2'$, where T_1 is disjoint from T_2, and T_1' is disjoint from T_2' this property fails to hold:

$$(\overrightarrow{(T_1 \cup T_2)}\ \overrightarrow{(T_1' \cup T_2')}) \neq (\overrightarrow{T_1}\ \overrightarrow{T_1'}) \circ (\overrightarrow{T_2}\ \overrightarrow{T_2'})$$

Instead of defining α-equivalence in terms of generalised transpositions, we give a more general definition that imposes constraints on the permutation applied, instead of using transpositions explicitly.

$$\frac{\pi \cdot (\exists T_1.E_1) = \exists T_2.E_2 \qquad \forall a.a \in (supp\ E_1 - T_1) \Rightarrow \pi a = a}{\exists T_1.E_1 =_\alpha \exists T_2.E_2} \tag{3.62}$$

The constraint on the permutation π says that the permutation is an identity on free variables of E_1, that is, it affects only bound variables in E_1. Since $\pi \cdot (\exists T_1.E_1) = \exists T_2.E_2 = \exists(\pi \cdot T_1).(\pi \cdot E_1)$,

Rule (3.62) says that two module signatures are α-equivalent for any permutation π such that it affects only bound variables of E_1 and makes components of the signatures equal. This definition does not depend on the particular way of constructing a permutation π. Particularly, in certain situations, when we need such a permutation, we can use a generalised transposition to construct it.

All the examples related to the simplified setting of module specifications given in Figure 3.13 generalises to the full setting of mutually dependent definitions of semantic objects given in Section 3.3.1. We will provide some details of how nominal techniques apply to the full setting when discussing our formalisation in the Coq proof assistant (see section 3.5.3).

In the conclusion of this section, we would like to mention the following. The use of nominal techniques could provide us with the mechanism for convenient reasoning about structures with binders using an induction principle that incorporates the idea of α-conversion. As it has been shown (see [Pit06] and [UBN07]), for equivariant relations one can derive a stronger induction principle with an additional finitely supported *induction context* C. Instantiating the induction context appropriately, one can obtain the required freshness conditions for cases where it is required.

Even for certain formulations of the simply-typed lambda calculus, to be able to prove the weakening property in a proof assistant, one has to use α-conversion explicitly. That is, if we defined the typing rule for the case of lambda abstraction in the following form

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \qquad x \notin dom(\Gamma)}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

then in the proof of the weakening lemma we would have to show that $x \notin dom(\Gamma')$ for $\Gamma \subseteq \Gamma'$, knowing that $x \notin dom(\Gamma)$. This is usually done informally: by variable convention we always can pick such x, but in the formalisation one has to make precise why such a renaming is possible. Since the typing relation is equivariant (and relations used in side conditions as well), one can use a strengthened induction principle. The induction principle, adapting the definition from [UBN07] looks as follows. For any predicate $P\ C\ \Gamma\ e\ \tau$, where

C is an additional induction context, we have:

$$\frac{\begin{array}{r}\forall C\ \Gamma\ n,\ P\ C\ \Gamma\ n\ Int \\ \forall C\ \Gamma\ x\ \tau,\ \Gamma(x) = \tau \Rightarrow P\ C\ \Gamma\ x\ \tau \\ \forall C\ \Gamma\ x\ e\ \tau_1\ \tau_2,\ x\#\Gamma \Rightarrow x\#C \Rightarrow \\ \qquad \Gamma, x : \tau_1 \vdash e : \tau_2 \Rightarrow (\forall C, P\ C\ (\Gamma, x : \tau_1)\ e\ \tau_1) \Rightarrow \\ \qquad P\ C\ \Gamma\ (\lambda x.e)(\tau_1 \to \tau_2) \\ \forall C\ \Gamma\ e_1\ e_2\ \tau_1\ \tau_2,\ \Gamma \vdash e_1 : \tau_1 \to \tau_2 \Rightarrow (\forall C, P\ C\ \Gamma\ e_1\ (\tau_1 \to \tau_2)) \Rightarrow \\ \qquad \Gamma \vdash e_2 : \tau_1 \Rightarrow (\forall C, P\ C\ \Gamma\ e_2\ \tau_1) \Rightarrow \\ \qquad P\ C\ \Gamma\ (e_1\ e_2)\ \tau_2\end{array}}{\forall\ C\ \Gamma\ e\ \tau,\ \Gamma \vdash e : \tau \Rightarrow P\ C\ \Gamma\ e\ \tau}$$

Notice an additional condition $x\#C$ in the case of lambda abstraction. Using this induction principle, the case for $\lambda x.e$ could be proved by instantiating the context C with Γ'.

3.5 Formalisation in Coq

We have formalised, in the Coq proof assistant, essential parts of the definitions given in Section 3.3 along with the proof of static interpretation normalisation. We have taken an extrinsic approach [BHKM12], as opposed to an intrinsic one, to the representation of the core language, the module language, and the target language, which keeps our implementation close to the approach presented in the paper. The extrinsic encoding has an advantage of being more suitable for code extraction to obtain a certified implementation.

We use two extra axioms in our Coq formalisation: the axiom of functional extensionality and the axiom of proof irrelevance. Both axioms are consistent with the theoretical foundation of the Coq proof assistant - Calculus of Inductive Constructions (CIC). See Remark 3.5.1 for the details about the proof irrelevance axiom.

That is, we have implemented the abstract syntax as simple inductive data types and given separate inductive definitions for relations such as elaboration, typing, and so on. The semantic objects from Section 3.3.1 have been implemented as mutually defined inductive types using Coq's `with` clause. The same approach is used in definitions of relations on semantic objects.

For proof automation, we make use of the `crush` tactic from [Chl13] and some tactics from [PdAC+16]. We have striven to keep the structure of proofs explicit, using automation to resolve only the most tedious and repetitive cases. When proving goals involving dependent types (like vectors) we use tactics `dependent destruction` and `dependent induction` from the `Program.Equality` module.

In the following, we are going to discuss issues related to some details of the implementation in Coq. Particularly, we will discuss the definitions of environments (finite maps) with properties that simplify proof development, some issues related to the conservative strict positivity checks in Coq for the definition of semantic objects, and induction principles used in proofs. We describe the nominal sets implementation and provide details of definitions

of particular nominal sets relevant for our development. We also present an implementation of the logical relation and highlight some details related to the proof of static interpretation normalisation (Theorem 3.3.1).

We will use Coq's syntax for most of the definitions in the reminder of this chapter, including lemmas and theorems, which directly correspond to our implementation. In the proof sketches that follow, we are aiming at showing the overall ideas and intuitions, which will serve as a guide to our Coq implementation.

3.5.1 Semantic Objects Representation

As described in Section 3.3, semantic objects are represented using finite maps (which we also call *environments*) and sets (see Figure 3.3). Indeed, the implementation makes use of Coq's standard library implementations of such objects. For environments we define a module type, which specifies operations on environments used in our implementation following our naming convention. We define a module, corresponding to this module type by mapping operations from our custom module type to the operations from the standard library. Specifically, we use the `FMapList` implementations of the `FMap` interface. In the `FMapList` implementation, the underlying data structure is a list of pairs `list (Key * A)`, where `Key` is the type of keys (a domain) and `A` is the type of values (a codomain). This list is equipped with the property of being ordered according to a strict order of keys (we assume that the type `Key` has a strict order). The list and the property are packed together using Coq's records, which can be seen as a generalisation of Σ-types. We use the definition from the `FMapList` module from the Coq standard library:

Definition 3.5.1.
```
Record slist (A : Type) :=
  {this :> list A; sorted : sort (ltk A) this}.
```

Here, the `this` field denotes an underlying list, and the `sorted` field represents the property of the list being ordered with respect to a strict order on keys:

```
Definition ltk : forall A : Type, Key * A → Key * A → Prop :=
fun (A : Type) (p p' : Key * A) ⇒ Key.lt (fst p) (fst p')
```

Coq automatically generates a constructor for the defined record:

```
Build_slist: forall (A : Type) (xs : list A),
             sort (ltk A) this → slist A
```

In our implementation, we instantiate the `FMapList` module from the standard library by the name `En`. We use `En.t` to refer to the environment type constructor, which corresponds internally to `slist` from Definition 3.5.1.

The property given by the `sorted` field of the `slist` record gives us the canonical representation of environments. The canonical representation allows us to prove an important property, which makes proofs involving environments easier to write: if two environments contain the same mappings, they are propositionaly equal in Coq. We call this property *environments extensionality*:

Definition 3.5.2 (Environments extensionality)**.** For any type `A : Type` and for any two environments `E E' : En.t A`, we have

```
(forall k : Key, look k E = look k E') → E = E'
```

Here the equals sign "=" refers to the Coq propositional equality, and

```
look : Key → En.t A → option A
```

is a lookup function.

Having this property for environments, one can use all of Coq's rewriting machinery instead of using a setoid equality.

Sometimes, using the concrete representation of some abstract notion can be helpful to prove properties by computation. For example, let us consider the following lemma (we use the notation ++ for the environment modification operation):

```
Lemma plus_empty_r: forall {A} (e : En.t A),
  e ++ empty = e.
Proof.
  reflexivity.
Qed.
```

The proof of this lemma makes use of the concrete representation of environments as lists with a well-formedness condition. In fact, this lemma is just a definitional equality, which is why we can prove it by `reflexivity` directly. The same trick does not work for the slightly different lemma:

```
Lemma plus_empty_l: forall {A} (e : En.t A),
    empty ++ e = e.
```

To prove this lemma using the concrete representation would require us to do induction and use properties of the `fold_left` function on lists, since this is how environment modification (addition) is defined. Instead, we can use environment extensionality to prove this lemma by using the specification of the environment modification operation.

In general, abstracting from the concrete representation has an advantage that we can change from one representation to another, but in this case we lose a computational behavior. As we will see in Section 3.5.4 it is not always possible to use an abstract representation.

Remark 3.5.1. In order to prove the environments extensionality property, we have to add the axiom of *proof irrelevance*:

```
proof_irrelevance : forall (P:Prop) (p1 p2:P), p1 = p2.
```

One can read this axiom as "any two inhabitants (proofs) of the same proposition are equal". Although, we could have taken another approach, and define the property of list to be ordered as a boolean-valued predicate, instead of `Prop`-valued. Then the property of a list `xs` being ordered could look like this: `sorted : isSorted xs ltk = true`. In this case we could have used the fact, that for types with decidable equality (type `bool`, in our case), uniqueness of identity proofs is provable (the Hedberg's theorem [Hed98]). We use this approach to prove the extensionality property for the `MSetList` (more modern) implementation of finite sets from the standard library (the well-formedness

predicate is boolean-valued in this implementation). The implementation of `FMapList` still states sortedness condition in `Prop`. To enable easier reuse of the standard library implementation, we have chosen to assume proof irrelevance to prove extensionality of environments.

Having environments as described above, we can now try to define semantic objects using Coq's mechanism of mutually inductive definitions. First of all, we define type environments in the obvious way (value environments `VEnv` are defined similarly):

```
Definition TEnv := AEnv Ty.
```

We would like to give a definition of semantic objects in the following way:

```
Inductive Env :=
    | EnvCtr : TEnv → VEnv → MEnv → MTEnv → Env
with MEnv :=
    | MEnvCtr : AEnv Mod → MEnv
with MTEnv :=
    | MTEnvCtr : AEnv MTy → MTEnv
with Mod :=
    | NonParamMod : Env → Mod
    | Ftor : TSet → Env → MTy → Mod
with MTy :=
    | MSigma : TSet → Mod → MTy.
```

Unfortunately, this definition does not work because of the conservative strict positivity check implementation in Coq. Specifically in definitions of `MEnv` and `MTEnv` we use the `VEnv` type constructor. Coq does not accept such a definition as strictly positive.

Remark 3.5.2. One simple example that violates strict positivity of inductive definitions is a definition of lambda terms that uses Coq's function space to represent lambda-abstraction:

```
Inductive term : Set :=
| App : term → term → term
| Abs : (term → term) → term.
```

This definition violates strict positivity, since the constructor `Abs` takes a function from `term` to `term`. That is, the inductive type being defined occurs in the negative position, to the left of an arrow. If Coq allowed such definitions, its underlying theory would become unsound, since one could than write non-terminating definitions (see examples in [Chl13, Section 3.6]). For that reason Coq performs a check for strict positivity, and this check is an overapproximation. That is, there are some definitions, which are strictly positive, but Coq is unable to recognize that. In the case of semantic objects, this is exactly the case, as we will see later.

If we drop the condition in the definition of `AEnv` that the list is sorted, then we can write a definition of `Env`. That is, we could have an association list for `MEnv` and `MTEnv` instead of proper environments, and than we would have to add a well-formedness condition to all the theorems related to semantic

objects. In [RRD10] the authors have chosen the approach with the separate well-formedness condition in the Coq formalisation. In our case such a design decision would lead to significant complications due to the presence of mutual inductive definitions. From a practical point of view, dependent types are useful exactly for propagating certain invariants associated with the data structure [Ler09]. For that reason, we would like to keep the well-formedness condition packed together with the underlying association list.

Instead, we introduce an isomorphic *pair-of-vectors* representation of environments, where the first vector is an ordered vector of keys and the second vector is a vector of values. By vector we mean the following inductive family of types indexed by natural numbers, as it is defined in the `Vector` module from the Coq standard library:

Definition 3.5.3.

```
Inductive Vec A : nat → Type :=
|nil : Vec A 0
|cons : forall (h:A) (n:nat), Vec A n → Vec A (S n).
```

The important feature of vectors is that they carry information about their lengths in the type "by construction". An alternative way to pack a list with its length is by using subset types:

```
Definition VecAlt A n := {xs : list A | length xs = n }
```

Such a definition would lead to the same problem as before: the definition of semantic objects would not pass the strict positivity check.

Separating keys and values in different vectors allows us to define semantic objects in a way acceptable for Coq's strict positivity checker. The idea of using an isomorphic representation is similar to Wadler's notion of *views* [Wad87]. We can see the two representations of environments as two views associated with the abstract "type" of environments, which corresponds to a module specifying operations on and properties of environments.

More precisely, we define our new representation of environments using Coq's records and subset types:

```
Definition skeys n := {vs : Vec Key n | vsort Key.lt vs }.

Record VecEnv (A : Type) :=
  mkVecEnv { v_size : nat;
             keys   : skeys v_size;
             vals   : Vec A v_size }.
```

This definition again uses dependent types to maintain two invariants, specifying (*i*) that the vector of keys is sorted, and (*ii*) that the vector of keys and the vector of values have the same length. Notice that using the subset types for the definition of `skeys` will not cause problems for the strict positivity check, because the type of keys is independent of the mutual inductive structure of semantic objects. The reason why we need the lengths of two vectors to be the same is the following. We are aiming to define our alternative representation of environments in such a way that we have enough information to show the

isomorphism between the two representations without adding any side conditions. In particular, if we did not include the condition on lengths of vectors and used just lists, we would have to add an extra side condition when defining a conversion function from the pair-of-vector representation to the one from the standard library. That would again lead to the same problem as with adding a well-formedness side condition to all the theorems related to semantic objects.

Before we start showing the isomorphism between the two representations, let us give some useful definitions related to propositional equality, since we use will have to reason about equality of dependent types. The Homotopy Type Theory book [Uni13] establishes the terminology and provides definition of basic notions, which we are going to use when we talk about equalities.

Definition 3.5.4 (Transport)**.** We call the following function *transport*:

```
transport : forall {A : Type} {a b : A} {P : A → Type},
            a = b → P a → P b
```

Informally, one can read this definition as "if `a` is equal to `b` and `P a` holds, then `P b` also holds". Following [Uni13], we are going to call `p : a = b` a *path* between `a` and `b`.

Definition 3.5.5 (Path concatenation)**.** Paths may be concatenated, which corresponds to transitivity of equality:

```
path_concat : forall {A : Type} {x y z : A},
              x = y → y = z → x = z
```

Definition 3.5.6 (Action on paths)**.** The application of `f` to a path (or action on paths) :

```
ap : forall {A B : Type} {a a' : A}
            (f : A → B) (p : a = a'),
     f a = f a'.
```

Now, we give some useful properties of transport.

Lemma 3.5.1 (Lemma 2.3.9 in [Uni13])**.** *For any* `A : Type`*, type family* `B : A → Type`*,* `x y z : A`*,* `u : B x`*, and paths* `p : x = y` *and* `q : y = z` *we have*

```
transport q (transport p u) = transport (path_concat p q) u.
```

Lemma 3.5.1 says, that transporting something twice, first along the path `p` and then along the path `q` is equal to transporting once, but along the concatenated path `path_concat p q`. The next lemma allows us to move transport in and out of the application.

Lemma 3.5.2 (Lemma 2.3.11 in [Uni13])**.** *For any* `A : Type`*, type families* `F,G : A → Type`*,* `a,a' : A`*,* `u : F x`*, dependent function* `f : forall (a : A), F a → G a`*, and path* `p : a = a'` *we have*

```
f (transport p u) = transport p (f u).
```

Lemma 3.5.3. *For any* `A : Type`, `x y : A`, *and* `p : x = y` *the following equations hold:*

(i) `path_concat p (eq_sym p) = eq_refl`

(ii) `path_concat (eq_sym p) p = eq_refl`

Where `(eq_sym p) : y = x`

Having lemmas about equality at hand, we can show the isomorphism between the two representations. The idea of the approach is simple: we use a well-known list-of-pairs to pair-of-lists correspondence. Although, for us it is a bit more subtle, since we use vectors instead of lists, and we have to maintain an additional invariant — vector of keys is sorted. First of all, we define zip and unzip operations on vectors. The zip operation on vectors has the following type:

```
vzip : forall {A B n}, Vec A n → Vec B n → Vec (A * B) n
```

The important difference from the `zip` function on lists is that dependent types allow us to ensure that the two input vectors and the output vector have the same size. Writing definitions like this in Coq sometimes is a non-trivial task. Pattern-matching in Coq requires additional work to pass all required information for the definition to type-check. That is, we use a *convoy* pattern [Chl13] to propagate the information that the length of two input vectors is the same during pattern-matching. The implementation of the `vzip` function in inspired by [Bre15].

```
Definition vzip {A B : Type} :
  forall {n}, Vec A n → Vec B n → Vec (A * B) n :=
  fix zip {n} vs := match vs in Vec _ m
                    return Vec B m → Vec (A * B) m with
  | [] ⇒ fun vs' ⇒ []
  | cons _ v n0 tl ⇒
    fun vs' ⇒
      (match vs' in Vec _ m'
         return (S n0 = m' → match m' with 0 ⇒
                (unit : Type) | S _ ⇒ Vec _ _ end) with
       | [] ⇒ fun _ ⇒ tt
       | cons _ v' n1 tl' ⇒
         fun H ⇒
         cons _ (v,v') _
         (zip tl (transport (eq_add_S _ _ (eq_sym H)) tl')))
  end) eq_refl
end.
```

There are two details to point out in this definition. First, after matching the first vector against the `cons` constructor, we already know that the second input vector cannot be empty, although Coq still requires us to exhaustively cover all the cases. We apply the usual trick here: use a `return` clause to

specify that in the impossible case we return a trivial type `unit`. The second detail is the recursive call. The second argument (the tail of the second vector) is not exactly of the right type. It has type `Vec B n1`, but we need a term of type `Vec B n0`, where n0 is the length of the first vector. At this point we use the convoy pattern to make a connection between the lengths of the two vectors: the return type of the match on `vs'` is a function, taking equality as an argument:

```
S n0  = m' → match m' with
             | 0 ⇒ (unit : Type)
             | S _ ⇒ Vec _ _ end.
```

Here `n0` is bound to the length of the first vector `vs`, and we apply this function to `eq_refl` to indicate that we expect lengths of the two vectors to be equal. Thus, in the `cons` case for the second vector, we now have `H : S n0 = S n1` at our disposal, and we can use it to construct a term of the right type for the second argument of the recursive call. In order to do that, we have to transport `tl'` along the equality `p : n0 = n1` (we will discuss transport later, when we state the isomorphism between the two representations of environments). We get `p` from `H` using injectivity of constructors (we use the `eq_add_s` lemma from the standard library).

The definition for the `vunzip` function is simpler, and does not involve complications with dependent pattern-matching.

```
Definition vunzip {A B : Type} :
    forall {n}, (Vec (A * B) n) → (Vec A n * Vec B n) :=
  fix unzip {n} vs := match vs with
  | [] ⇒ ([],[])
  | (a,b) :: tl ⇒ (a :: fst (unzip tl), b :: snd (unzip tl))
end.
```

Now, we can show that vzip/vunzip are inverses of each other:

Lemma 3.5.4 (vzip/vunzip inverses)**.** *For the functions* `vzip` *and* `vunzip`*, defined above, and any* `n`*,* `A`*,* `B`*, the following holds:*

(i) for any vector of pairs `kvs : Vec (A*B) n` *we have*
`(fun p ⇒ vzip (fst p) (snd p)) (vunzip kvs) = kvs`

(ii) for any two vectors `vs : Vec A n` *and* `vs' : Vec A n` *we have*
`vunzip (vzip vs vs') = (vs, vs')`

Proof. We show

(i) By induction on `kvs`.

(ii) By induction on `vs` and performing dependent case analysis (using the `dependent destruction` tactics) on `vs'`.

■

So far, we have shown a conversion between a pair of vectors and a vector of pairs, but we need another "layer" of conversion functions, which also compose to identity in both directions.

Definition 3.5.7. Conversion function between lists and vectors:

```
to_list : forall (A : Type) (n : nat), Vec A n → list A
of_list : forall (A : Type) (xs : list A), Vec A (length xs)
```

We need an auxiliary lemma, before we start showing that these two functions are inverses of each other.

Lemma 3.5.5. *For any* `n m : nat`*, a vector* `vs : Vec A n`*,* `a : A`*,* `S` *the successor constructor of* `nat`*, and a path* `p : n = m` *we have*

```
transport (ap S p) (a :: vs) = a :: transport p vs.
```

Proof. First, let us check if the statement type-checks. On the right-hand side we have `(transport p vs) : Vec A m` (transporting `Vec A n` along the path `p : n = m`). Applying the `cons` constructor gives us (according to the Definition 3.5.3) `(a :: transport p vs) : Vec A (S m)`. For the left-hand side we have `(a :: vs) : Vec A (S n)`, and we use action on paths of the successor constructor for natural numbers to get the right path : `(ap S p) : S n = S m`. Now, it is easy to see that the whole left-hand side has the same type as the right-hand side: `transport (ap S p) (a :: vs) : Vec A (S m)`.

We prove the lemma by case analysis on `p`: it suffices to consider the case, when `m` is `n` and `p` is `eq_refl`. Then our goal reduces to

```
transport (ap S eq_refl) (a :: vs) = a :: transport eq_refl vs
```

Both, `transport` and `ap` compute on `eq_refl`. After simplification, we get

```
a :: vs = a :: vs
```

as required. ■

Composition of `to_list` and `of_list` in one direction is easy to state and prove to be the identity. Specifically, the proof of the fact that for any `A : Type` and `xs : list A`, `to_list (of_list xs) = xs` can be found in the standard library. Let us focus on the other direction (there is no proof of this property in the standard library). Even to state it requires some work. If we did it naively, as "for any `n : nat`, `A : Type`, and `vs : Vec A n`, `(of_list (to_list vs)) = vs`", then our definition would not type-check and we would get the error "The term "`vs`" has type "`Vec A n`" while it is expected to have type "`Vec A (length (to_list vs))`"". The reason for that error is that the right-hand side and the left-hand side of the equation are of different type. We can fix this problem using the following observation: if we pass a vector `vs : Vec A n` to the `to_list` function, then the resulting list will be exactly of length `n`.

```
to_list_length : forall (n : nat) (A : Type) (vs : Vec A n),
                 n = length (to_list vs)
```

Remark 3.5.3. Notice that in our Coq implementation we make the definition of `to_list_length` transparent by using the `Defined` keyword instead of `Qed`. This way, we make the term, witnessing the equality, available for simplification, so we can exploit definitional qualities in some proofs.

Now, we are ready to state properties of `to_list` and `of_list`.

Lemma 3.5.6. *For the functions* `to_list` *and* `of_list` *(Definition 3.5.7), we have the following equations:*

(i) for any `A : Type` *and* `xs : list A` *we have*
`to_list (of_list xs) = xs`

(ii) for any `A : Type`, `n : nat`, `vs : Vec A n` *we have*
`of_list (to_list vs) = transport (to_list_length vs) vs`

Proof. We show

(i) by induction on `xs`. In the base case, both sides of the equation evaluate to the empty list. In the induction step, we simplify the goal and rewrite it using the induction hypothesis.

(ii) by induction on `vs`. In the base case both sides of the equation evaluate to the empty list. In the induction step, we apply the following equational reasoning:

```
  h :: of_list (to_list vs)
= transport (to_list_length (h :: vs)) (h :: vs)
= {by definitional equality (see Remark 3.5.3)}
= transport (ap S (to_list_length vs)) (h :: vs)
= {by lemma 3.5.5}
= h :: transport (to_list_length vs) vs
= {by induction hypothesis}
= h :: of_list (to_list vs)
```

■

Another invariant, which should be preserved by the conversion functions is the order of the elements. We have proved several auxiliary lemmas showing that the order is preserved by the functions `vzip`/`vunzip` and the functions `to_list`/`of_list`. Proofs of these lemmas are quite straightforward, since none of these functions rearrange the elements.

We define the conversion function between the two representations of environments using the definitions above. We use `En.t` to refer to the standard library implementation of environments corresponding to Definition 3.5.1:

```
Definition toOrdEnv {A : Type} (ve : VecEnv A) : En.t A :=
  match ve with
  | mkVecEnv _ _ (exist _ ks sorted_ks) vs ⇒
      let skvs := vzip_sorted ks vs sorted_ks
      in Build_slist (to_list_sorted (vzip' (ks,vs)) skvs)
```

```
  end.

Definition fromOrdEnv {A : Type} (oe : En.t A) : VecEnv A :=
  match oe with
  | @En.Build_slist _ xs xs_sort ⇒
    let kvs := vunzip (of_list xs) in
    let vs := snd kvs in
    let skvs := vunzip_sorted (of_list xs)
                              (of_list_sorted _ xs_sort)
    in mkVecEnv A _ (exist _ (fst kvs) skvs) vs
  end.
```

Notice that in the definition of `toOrdEnv` we use the function

```
zip' : forall {A B : Type}, Vec A n * Vec B n → Vec (A * B) a,
```

which is a curried version of the `zip` function.

Let us prove congruence lemmas, which we are going to use in our proof that the functions `toOdrEnv`/`fromOrdEnv` are inverses of each other.

Lemma 3.5.7 (Congruence for ordered-list environments)**.** *For any* `A : Type`*, lists* `xs ys: list (Key.t * A)`*, which are sorted according to the strict order of keys* `sxs : sort ltk xs`*,* `sys : sort ltk ys`*, if* `xs = ys` *then two records, corresponding to the environments are equal:*
`{| En.this := xs; En.sorted := sxs |} =`
`{| En.this := ys; En.sorted := sys |}.`

Proof. Essentially, this is the same property as for subset types in presence of proof irrelevance. The first components are equal by assumption, and `sxs = sys` by proof irrelevance. ■

Lemma 3.5.8 (Congruence for pair-of-vectors environments)**.** *For any* `A : Type`*,* `n,n' : A`*, two sorted vectors of keys*
`ks : Vec Key.t n`*,* `ks_sort : vsort Key.lt ks`*,*
`ks' : Vec Key.t n'`*,* `ks_sort' : vsort Key.lt ks'`*,*
two vectors of values `vs : Vec A n` *and* `vs' : Vec A n'`*, and path* `p : n = n'`*, if* `(transport p ks) = ks'` *and* `(transport p vs) = vs'` *then*

```
  {| v_size := n;  keys := exist _ ks ks_sort;   vals := vs  |} =
  {| v_size := n'; keys := exist _ ks' ks_sort'; vals := vs' |}.
```

Proof. The proof follows essentially the same patterns as the proof of Lemma 3.5.7. First, by case analysis on `p`, we get `ks = ks'` and `vs = vs'`, because transport computes on `eq_refl`. It only remains to be shown, that `ks_sort = ks_sort'`, but this equality holds by proof irrelevance. ■

Lemma 3.5.9 (toOrdEnv/fromOrdEnv inverses)**.** *For any* `A`*,* `ve : VecEnv A`*, and* `oe : En.t` *we have*

(i) `(toOrdEnv (fromOrdEnv oe)) = oe`

(ii) `(fromOrdEnv (toOrdEnv ve)) = ve`

Proof. We prove

(i) using Lemmas 3.5.7, 3.5.6(i) and 3.5.4(i);

(ii) as follows. This direction is a bit harder to prove, since we have to reason about equality of vectors. We use Lemma 3.5.8, which gives us two goals that are quite similar. We prove them using lemmas about transport (Lemmas 3.5.2 and 3.5.1) to bring together paths that give reflexivity by Lemma 3.5.3, and Lemmas 3.5.6(ii) and 3.5.4(ii).

■

We could have defined similar operations on pair-of-vectors environments and prove all the required properties as for the environments from the standard library, but this would be a quite time consuming process. Instead, we use Coq's coercion mechanism. The coercion functions are precisely the functions given by the isomorphism between the two representations.

```
Coercion _to {A} := toOrdEnv (A:=A).
Coercion _from {A} := fromOrdEnv (A:=A).
```

In most situations, Coq inserts coercion functions automatically in an expected way, which simplifies development using a pair-of-vectors representation of environments. However, if there is some ambiguity in what way coercions could be applied, we have to fallback to manual application of the coercion functions.

Our Coq development shows that in most of the proofs involving the pair-of-vectors environments it suffices to just use Lemma 3.5.9.

We can now define semantic objects using a pair-of-vectors representation without violating Coq's strict positivity check. The isomorphism between the two representations ensures that we can transfer properties of one representation to the other.

Definition 3.5.8 (Semantic objects)**.**

```
Inductive Env :=
| EnvCtr : TEnv → VEnv → MEnv → MTEnv → Env
with MEnv :=
| MEnvCtr : VecEnv Mod → MEnv
with MTEnv :=
| MTEnvCtr : VecEnv MTy → MTEnv
with Mod :=
| NonParamMod : Env → Mod
| Ftor : TSet → Env → MTy → Mod
with MTy :=
| MSigma : TSet → Mod → MTy.
```

The definition of interpretation environments has a similar structure and uses the same approach with a pair-of-vector representation.

Definition 3.5.9 (Interpretation environments)**.**

```
Definition IVEnv := EnvMod.t (label*Ty).

Inductive IEnv :=
| IEnvCtr : TEnv → IVEnv → IMEnv → MTEnv → IEnv
with IMEnv :=
| IMEnvCtr : (VecEnv.VecEnv IMod) → IMEnv
with IMod :=
| INonParamMod : IEnv → IMod
| IFtor : IEnv → TSet → Env → MTy → mid → mexp → IMod.
```

Operations on Semantic Objects

We briefly describe operations on semantic objects, such as lookup for different types of identifiers, including *long* identifiers, which represent paths in the nested module structure. Most of the operations are just liftings of corresponding operations of "flat" environments, such as described in the subsection 3.3.1. All the implicit injections and operations on components of semantic objects, mentioned in Section 3.3.1, are made explicit in our Coq implementation.

We consider several examples of definitions to sketch the overall idea. First, we start with the concept of long identifiers. Long identifiers are defined as an inductive data type with two constructors: one for the type, value or module identifier, and the other one to build a path out of a sequence of module identifiers.

```
Inductive longtid :=
| Tid_longtid : tid → longtid
| Long_longtid : mid → longtid → longtid.
```

This definition shows how the long identifiers for type lookup are defined. Our implementation contains two more similar definitions of long identifiers, `longvid` and `longmid` for looking up values and modules, respectively.

The lookup function can be defined by recursion on the structure of a long identifier.

```
Fixpoint lookLongTid (longk : longtid) (e : Env) : option Ty :=
  match e  with
  | EnvCtr te _ me _ ⇒
    match longk with
    | Tid_longtid k ⇒ look k te
    | Long_longtid m_id longk' ⇒
      match (lookMid m_id me) with
      | Some (NonParamMod e') ⇒ lookLongTid longk' e'
      | _ ⇒ None
      end
    end
  end.
```

For the values and modules components of the semantic objects we define the similar functions, following the name convention.

```
  lookLongVid (longk : longvid) (e : Env) : option Ty
  lookLongMid (longk : longmid) (e : Env) : option Mod
```

The lookup function for "flat" environments

```
look : forall A : Type, En.key → En.t A → option A
```

is polymorphic with respect to the type of values in the environment. Environments containing modules and module types (`MEnv` and `MTEnv`) are part of the mutually recursive structure and therefore wrapped in constructors. Before looking up in these environments we pattern-match on the respective constructor to "unwrap" the environment and then apply the `look` function. The `add` operation (the operation that adds a new mapping to an environment) in case of the pair-of-vectors environment types `MEnv` and `MTEnv`, first transports an environment through the isomorphism, applies the usual `add` (of standard library implementation of environments) and then transports the result back. The operation of environment modification for the semantic objects is defined componentwise.

3.5.2 Induction Principles

In order to prove theorems by induction over the structure of semantic objects, or relations containing mutual definitions, Coq's `Scheme` command is used to generate suitable induction principles. For some of the definitions, such as those for semantic objects and interpretation environments, the generated induction principles are not sufficiently strong, which is caused by the presence of nested inductive types; some constructors take environments as parameters, and the environments, being essentially list-like structures, make the whole definition a nested inductive definition. For each of these cases, a suitable induction principle is defined manually, following essentially the same approach as in [Chl13, Section 3.8].

Let us consider an induction principle for the semantic objects given by Definition 3.5.8. Usually, Coq generates an induction principle from the definition of an inductive data type, but nested inductive definitions are not covered by this procedure. In the case of `MEnv` (the case of `MTEnv` is similar), we want the induction hypothesis saying that some predicate `P : Mod → Prop` holds for all the values in the pair-of-vectors environment argument of the `MEnvCtr` constructor. Since we are interested only in a predicate on values in the environment (a codomain), we can use the projection from the pair-of-vectors representation to a vector of values:

```
vals : forall {A : Type} (v : VecEnv A), Vec A (v_size A v)
```

That is, we just need a predicate stating that some property holds for all the elements in a vector. There are at least two ways to define such a predicate: by recursion and by induction. We are going to use the inductive variant of the predicate from the `Vector` module of the standard library:

```
Inductive Forall {A} (P: A → Prop)
          : forall {n} (v: t A n), Prop :=
 |Forall_nil: Forall P []
 |Forall_cons {n} x (v: t A n) : P x → Forall P v →
                                 Forall P (x::v).
```

With this definition of the `Forall` predicate, we can define a sufficiently strong induction principle for semantic objects:

```
Definition Env_mut' (P : Env → Prop) (P0 : MEnv → Prop)
  (P1 : MTEnv → Prop)
  (P2 : Mod → Prop) (P3 : MTy → Prop)
  (f : forall (t : TEnv) (v : VEnv) (m : MEnv),
       P0 m → forall m0 : MTEnv, P1 m0 → P (EnvCtr t v m m0))
  (f0 : forall (t : VecEnv Mod),
      Forall P2 t → P0 (MEnvCtr t))
  (f1 : forall (t : VecEnv MTy),
          Forall P3 t → P1 (MTEnvCtr t))
  (f2 : forall e : Env, P e → P2 (NonParamMod e))
  (f3 : forall (t : TSet) (e : Env),
        P e → forall m : MTy, P3 m → P2 (Ftor t e m))
  (f4 : forall (t : TSet) (m : Mod), P2 m → P3 (MSigma t m)) :=
fix F (e : Env) : P e :=
  match e as e0 return (P e0) with
  | EnvCtr t v m m0 ⇒ f t v m (F0 m) m0 (F1 m0)
  end
with F0 (m : MEnv) : P0 m :=
  match m as m0 return (P0 m0) with
    | @MEnvCtr t ⇒ let fix step {n} (ms : Vec Mod n) : Forall P2 ms :=
                       match ms in Vec _ n' return @Forall _ P2 n' ms with
                         | [] ⇒ Forall_nil P2
                         | y :: l ⇒
                           @Forall_cons _ P2 _ y _ (F2 y) (step l)
                       end
                   in f0 t (step (vals _ t))
  end
with F1 (m : MTEnv) : P1 m :=
  match m as m0 return (P1 m0) with
  | @MTEnvCtr t ⇒ let fix step {n} (ms : Vec MTy n) : Forall P3 ms :=
                       match ms with
                         | [] ⇒ Forall_nil P3
                         | y :: l ⇒
                           @Forall_cons _ _ _ y l (F3 y) (step l)
                       end
                  in f1 t (step t)
  end
with F2 (m : Mod) : P2 m :=
  match m as m0 return (P2 m0) with
  | NonParamMod e ⇒ f2 e (F e)
  | Ftor t e m0 ⇒ f3 t e (F e) m0 (F3 m0)
  end
with F3 (m : MTy) : P3 m :=
  match m as m0 return (P3 m0) with
  | MSigma t m0 ⇒ f4 t m0 (F2 m0)
  end
for F.
```

A large portion of this induction principle is generated by the `Scheme` command. The important modifications we have had to do manually are concerned

with the `f0` and `f1` cases (and the `F0` and `F1` cases of the proof term respectively), where we use the `Forall` predicate to specify the desired property.

3.5.3 Nominal Techniques in Coq

There are existing developments of nominal techniques for proof assistants. Probably, the most developed one is the Nominal Isabelle package for the Isabelle proof assistant [UT05], which includes generalised name abstraction [UK11]. On the other hand, for the Coq proof assistant, there are no packages for nominal techniques in the standard distribution. Probably, the only known work on nominal techniques in Coq is [ABW07]. This work is mostly focused on the case of simply typed lambda calculus and does not cover generalised name abstraction.

We have developed an implementation of notions described in Section 3.4 using Coq's module system along with dependent records. Ideally, we would like to use only dependent records in our implementation, but unfortunately, finite sets from the Coq's standard library are implemented as parameterised modules. We wanted to use the standard library in our development as much as possible to avoid extra efforts spent on implementing standard functionality.

We started with the definition of atoms. Since the definition of atoms involves finite sets, and since the nominal techniques use finite sets extensively, we decided to use an `MSet` implementation of finite sets from Coq's standard library. We expose finite sets through our own module type, which adds required operations and properties missing in the `MSet` interface:

- set disjointness relation;
- set extensionality;
- map operation on sets.

We call our module type of finite sets `SetExtT` and the implementation of this module type `SetExt`.

We define the following module type for atoms:

```
Module Type Atom.
  Declare Module V : SetExtT.
  Axiom Atom_inf : forall (X : V.t), {x : V.elt | ~ V.In x X}.
End Atom.
```

We use `V.t` for the type of finite sets and `V.elt` for the type of elements. The `Atom_inf` axiom says that for any given finite subset of atoms, one can always find an element, which is not in this finite subset. We use subset types to specify that there exists such an element. It is important to use `Type` in this definition and not `Prop`, since if we defined the `Atom_inf` as `forall (X : V.t), exists x : V.elt,~ V.In x X`, we would not be able to use `Atom_inf` to construct functions that generate fresh atoms. This is due to limitations on `Prop`, allowing to eliminate propositions only to `Prop` again. In other words, proofs can be used to construct other proofs, but not programs.

We define a parameterised module `Nominal` that accepts an implementation of the `Atom` module type. Definitions below are contained in the `Nominal` module.

Let us first give definitions of notions required to define a permutation. We define predicates `is_inj` and `is_surj`, representing injectivity and surjectivity of a function, respectively, as follows:

```
Definition is_inj  {A B : Type} (f : A → B) : Prop :=
  forall (x y : A), f x = f y →  x = y.
Definition is_surj {A B : Type} (f : A → B) : Prop :=
  forall (y : B), exists (x : A), f x = y.
```

We then say that a function `f` is bijective if it is both injective and surjective:

```
Definition is_biject {A B} (f : A → B) :=
  (is_inj f) ∧ (is_surj f).
```

We define a predicate `has_fin_support` of a function `f` as

```
Definition has_fin_supp f :=
    exists S, (forall t, ∼ V.In t S →  f t = t).
```

Finally, we define a finitely supported permutation using Coq's dependent records:

```
Record Perm :=
      { perm : V.elt →  V.elt;
        is_biject_perm :  is_biject perm;
        has_fin_supp_perm : has_fin_supp perm}.
```

That is, to define an inhabitant of `Perm`, one needs to provide a function and proofs of two properties: that the function is a bijection, and that is has a finite support. We call the projection `perm` of the `Perm` record the *underlying function* of a permutation.

Notice that since we defined properties of the underlying function of a permutation as inhabitants of type `Prop`, in presence of the proof irrelevance axiom, we can prove that two permutations are equal if their underlying functions are equal.

As an example, let us define first an identity permutation. We take the identity function `id` as an underlying permutation function. Proofs of required properties of permutation are simple, and we use the `refine` tactic here to construct the permutation. The `refine` tactic allows one to provide parts of the definition leaving other parts as "holes" that generate proof obligations.

```
Definition id_perm : Perm.
  refine ({| perm:=id; is_biject_perm := _; has_fin_supp_perm := _ |}).
  + split. auto. refine (fun y ⇒  ex_intro _ y _);reflexivity.
  + exists V.empty;intros;auto.
Defined.
```

Next, we define the composition of permutations. We use the same approach here, namely the `refine` tactic with a partially constructed record, corresponding to the permutation.

```
Definition perm_comp (p p' : Perm) : Perm.
  refine ({| perm:= (perm p) ∘ (perm p');
             is_biject_perm := _;
             has_fin_supp_perm := _ |}).
```

```
    (* Proofs are omitted *)
Defined.
```

We omit proofs of obligations generated by `refine` here. The proof that composition is a bijection boils down to facts that injectivity and surjectivity are preserved by function composition. For the finite support we choose the union of supports of the composed permutations. We define the following notation for the composition of permutations:

```
Notation "p ∘p p'" := (perm_comp p p') (at level 40).
```

The transposition function of two atoms follows Definition 3.4.3:

```
Definition swap_fn (a b c : V.elt) : V.elt :=
  if (V.E.eq_dec a c) then b
    else (if (V.E.eq_dec b c) then a
      else c).
```

We prove that the `swap_fn` function is a bijection and that its finite support is the two element set $\{a, b\}$. By packing the `swap_fn` function with these proofs we define the corresponding instance of `Perm`.

To define the underlying function of a generalised transposition (Definition 3.4.4) we use the `fold_right` function that accumulates elementary swaps by composing them as functions starting from the identity function:

```
Definition swap_iter_fn (vs : list (V.elt * V.elt))
  : V.elt → V.elt :=
    fold_right (fun (e' : (V.elt * V.elt)) (f : V.elt → V.elt) ⇒
                let (e1,e2) := e' in f ∘ (swap_fn e1 e2)) id vs.
```

We prove that this function satisfies properties of finitary permutations. That is, `swap_iter` is bijective and has a finite support, which is a set of all variables from the argument list. With these properties at hand we can construct an inhabitant of the `Perm` type.

Our implementation also provides the functionality for generation of fresh names for a given set of atoms. The freshness relation is defined as it is described in Section 3.4 (Definitions 3.4.8 and 3.4.9).

```
Definition fresh a A :=   V.In a A.
Definition all_fresh (x y : V.t) :=
    forall k,  (V.In k x ∧ V.In k y).

Infix "#" := fresh (at level 40) : a_scope.
Infix "#" := all_fresh (at level 40) : as_scope.

Delimit Scope a_scope with Atom.
```

We overload the notation for the freshness relation to use it in both cases: for a single atom and for a set of atoms.

We start with generating one fresh name along with the proof of freshness. First, we define a type of functions from the type of finite sets to the type of atoms, with the property that when applied to a finite set, it returns a fresh atom (with respect to the given set). We use subset types to equip a function with the property:

```
Definition FreshFn a :=
  {f : V.t → V.elt | forall x, ((f x) # a)%Atom.
```

We use explicit scope annotation here to point out which freshness relation we use.

Now, we can define a function that takes a set of atoms and returns a fresh atom with the proof of freshness. In order to obtain a fresh atom we use the fact that the set of atoms is countably infinite:

```
Definition fresh_fn : forall a : V.t, FreshFn a :=
fun a ⇒ let H := Atom.Atom_inf a in
         exist (fun f : t → elt ⇒ forall x : t, (f x * a)%Atom)
               (fun _ : t ⇒ proj1_sig H)
               (fun _ : t ⇒ proj2_sig H).
```

In this way we abstract the mechanism of fresh atoms generation, since it will work with any implementations of module type `Atom` used to instantiate the `Nominal` module.

Next, we generalise the `fresh_fn` to return a set of fresh atoms with the proof of freshness. Again, we first define a subset type that packs together a finite set, the property of freshness, and the cardinality of the set of fresh atoms:

```
Definition AllFresh a n :=
   { b : V.t | (b * a) ∧ V.cardinal b = n }.
```

To generate a set of `n` fresh atoms (with respect to some finite set `X`), we have to recursively pass the set `X` with a newly generated atom added to the set to ensure freshness of all `n` new atoms. We define a function that generates a set of `n` atoms by recursion on `n`:

```
Fixpoint get_freshs_internal (X : V.t) (n : nat) : V.t :=
  match n with
  | O ⇒ empty
  | S n' ⇒ let fatom := (proj1_sig (fresh_fn X)) X in
            add fatom (get_freshs_internal (add fatom X) n')
  end.
```

By induction on `n` one can show that these atoms are indeed fresh and the cardinality of the resulting set is equal to `n`. Finally, we can define a function that returns a value of type `AllFresh a n` for some finite set `a` and a natural number `n`:

```
Definition get_freshs (X : V.t) (n : nat) : AllFresh X n :=
  exist _ (get_freshs_internal X n)
          (conj (get_freshs_internal_all_fresh n X)
                (get_freshs_cardinality n X)).
```

To implement a nominal set we use Coq's module system. That is, we define a module type of nominal sets, which follows Definition 3.4.7 and includes such components as the type of elements, the action of a finitary permutation on the elements of this type (along with properties of the action), and the finite support.

```
Module Type NominalSet.
    Import V.

    Parameter X : Type.

    Parameter action : Perm → X → X.
    Notation "r @ x" := (action r x) (at level 80).

    Axiom action_id : forall (x : X), (id_perm @ x) = x.
    Axiom action_compose : forall (x : X) (r r' : Perm),
                           (r @ (r' @ x)) = ((r ∘p r') @ x).

    Parameter supp : X → V.t.
    Axiom supp_spec : forall (r : Perm) (x : X),
      (forall (a : elt), In a (supp x) → (perm r) a = a) →
         (r @ x) = x.

End NominalSet.
```

Remark 3.5.4. The module type `NominalSet` allows for defining a support function `supp` that returns a finite support of an element that is not necessarily the smallest one, meaning that not all the definitions of `supp` will be equivariant functions. In each instance of `NominalSet` in our Coq development there is an obvious way to define a `supp` function such that it returns a smallest support of an element. Although, this is not enforced by the definition of the nominal set that we have. One can remedy this by adding an explicit constraint on a `supp` function to `NominalSet` saying that the function must be equivariant.

Our `Nominal` module includes an implementation of `NominalSet` for the type of finite sets of atoms in the way it is given by Definition 3.4. This module also includes additional facts like an action on a singleton set, equivariance of the union and intersection operation on finite sets, and equivariance of the set disjointness relation.

The running example of simplified semantic objects from Section 3.4 is implemented as a nominal set using the `NominalSet` signature. Our implementation includes both definitions of α-equivalence: the one using generalised transpositions and the other with a condition on a permutation. Moreover, we have developed a proof of equivariance of the elaboration relation in the simplified setting (Figure 3.13).

We have defined a nominal set of full semantic objects. One interesting aspect to point out is that the action and the support, being defined as fixpoints, were accepted by Coq, despite the fact that they call `map` and `fold_right` functions for the case of module environments `MEnv` and module type environments `MTEnv`. We provide one example of the permutation action definition for semantic objects:

```
Fixpoint action (p : Perm) (E : X) :=
  match E with
  | EnvCtr te ve me mte ⇒
      EnvCtr (PermPlainEnv.action p te)
             (PermPlainEnv.action p ve)
```

```
            (action_me p me) (action_mte p mte)
  end
with action_me p me :=
  match me with
  | MEnvCtr {| v_size := nn;  keys := ks; vals := vs |} ⇒
    MEnvCtr {| v_size := nn;  keys := ks;
               vals := (map (action_mod p) vs) |}
   end
with action_mte (p : Perm) mte:=
  match mte with
  | MTEnvCtr {| v_size := nn;  keys := ks; vals := vs |} ⇒
    MTEnvCtr {| v_size := nn;  keys := ks;
                vals := (map (action_mty p) vs) |}
  end
with action_mod p (md : Mod) : Mod :=
   match md with
   | NonParamMod e ⇒ NonParamMod (action p e)
   | Ftor ts e mty ⇒ Ftor (PFin.action p ts)
                          (action p e)
                          (action_mty p mty)
   end
with action_mty p (mty : MTy) : MTy :=
  match mty with
  | MSigma ts m ⇒ MSigma (PFin.action p ts) (action_mod p m)
  end.
```

The definition above uses previously constructed nominal sets for "flat" environments `PermPlainEnv`, finite sets `PFin`, and the `map` function on vectors to apply actions `action_mod` and `action_mty` to all values in the corresponding environments.

The implementation of `NominalSet` for the interpretation environments follows the same pattern, since they have a structure similar to semantic objects. Some components of the interpretation environments definition includes pieces of semantic objects, such as `Env`, `MTEnv`, `MTy`, and we use respective functions from the nominal set of semantic objects in the definition of the action and the support for interpretation environments.

We define the α-equivalence relation on semantic objects using a similar approach as in Definition 3.62 in Section 3.4, following the mutual inductive structure of semantic objects.

We use the following notation for the difference operation on sets, the action of a permutation on finite sets, and the action of a permutation on modules.

```
Infix ":-:" := Atom.V.diff (at level 40).
Notation "r @ x" := (PFin.action r x) (at level 80) : set_scope.
Notation "r @ x" :=
    (PermSemOb.action_mod r x) (at level 80) : env_scope.

Delimit Scope set_scope with S.
Delimit Scope env_scope with E.
```

The α-equivalence relation is defined as follows.

```
Inductive ae_env : Env → Env → Prop :=
| ae_env_c : forall (ve' ve : VEnv) (te' te : TEnv)
                    (me' me : MEnv) (mte' mte : MTEnv),
    ve' = ve →
    te' = te →
    ae_menv me' me →
    ae_mte mte' mte →
    ae_env (EnvCtr te' ve' me' mte') (EnvCtr te ve me mte)
  with
  ae_menv : MEnv → MEnv → Prop :=
  | ae_menv_c : forall (me' me : VE.VecEnv Mod),
      (forall mid (e' e : Mod),
          look mid (_to me') = Some e' →
          look mid (_to me) = Some e →
          ae_mod e' e) →
      ae_menv (MEnvCtr me') (MEnvCtr me)
  with
  ae_mte : MTEnv → MTEnv → Prop :=
  | ae_mte_c : forall (mte' mte : VE.VecEnv MTy),
      (forall mtid (e' e : MTy),
          look mtid (_to mte') = Some e' →
          look mtid (_to mte) = Some e →
          ae_mty e' e) →
      ae_mte (MTEnvCtr mte') (MTEnvCtr mte)
  with
  ae_mod : Mod → Mod → Prop :=
  | ae_mod_np : forall e' e,
      ae_env e' e → ae_mod (NonParamMod e') (NonParamMod e)
  | ae_mod_ftor : forall t e e' mty mty',
      ae_env e e' →
      ae_mty mty mty' →
      ae_mod (Ftor t e' mty') (Ftor t e mty)
 with
 ae_mty : MTy → MTy → Prop :=
 | ae_mty_c : forall m m',
     forall (T T' : Atom.V.t) p,
       (forall a, Atom.V.In a ((PermSemOb.supp_mod m) :-: T)
                  → (perm p) a = a) →
    ae_mod m' (p @ m)%E →
    T' = (p @ T)%S →
    ae_mty (MSigma T' m') (MSigma T m).
```

The essential part of this definition is the `ae_mty` case. It allows for relating module types up to permutations that affect only variables distinct form the set `T`.

Let us show an example, where explicit α-equivalence is needed in order to elaborate a module declaration.

Example 3.9: Let $F = \forall\emptyset.(\{\}, \exists\{x\}.\{a \mapsto x\})$ be a functor and $E = \{f \mapsto F\}$ be an environment containing this functor. We want to elaborate a sequence

of module declarations

$$\textbf{module } m_1 = f(\epsilon); \textbf{module } m_2 = f(\epsilon)$$

in the environment E into the following module type

$$\exists\{x, y\}.\{m_1 \mapsto \{a \mapsto x\}, \{m_2 \mapsto \{a \mapsto y\}\}$$

Two different functor applications give us results that could differ in names of bound variables. According to Rule (31) we have to build two derivations (side conditions are trivially satisfied in this case). We start with the first one.

$$\frac{\dfrac{E \vdash \epsilon : \{\} \qquad E(f) = (\{\}, \exists\{x\}.\{a \mapsto x\})}{E \vdash f(\epsilon) : \exists\{x\}.\{a \mapsto x\}}}{E \vdash \textbf{module } m_1 = f(\epsilon) : \exists\{x\}.\{m_1 \mapsto \{a \mapsto x\}\}}$$

The derivation for the second declaration in the sequence is not very different. The key observation here is that we can only derive the following:

$$E + \{a \mapsto x\} \vdash \textbf{module } m_2 = f(\epsilon) : \exists\{x\}.\{m_2 \mapsto \{a \mapsto x\}\}$$

The reason for this is that we are looking up f in the environment $E + \{a \mapsto x\}$, which gives the same result as in the environment E. Now, we have to apply α-renaming. Otherwise it would be impossible to satisfy the condition $T_1 \cap (\text{tvs}(E) \cup T_2) = \emptyset$ in (31). That is, we have to rename x in the second derivation.

The possibility of α-renaming as illustrated in Example 3.9 is usually implicit on paper, but in the Coq formalisation, we have to include it explicitly in the rule. Rule (31) becomes the following.

$$\frac{\begin{array}{c} E \vdash mdec_1 : \exists T_1.E_1 \qquad E + E_1 \vdash mdec_2 : \exists T_2.E_2 \\ \exists T_1.E_1 =_\alpha \exists T_1'.E_1' \qquad \exists T_2.E_2 =_\alpha \exists T_2'.E_2' \\ T_1' \# T_2' \qquad T_1' \# \text{tvs}(E) \end{array}}{E \vdash mdec_1\ mdec_2 : \exists(T_1' \cup T_2').(E_1' + E_2')} \quad (63)$$

Notice, that we also express the side condition on variable sets disjointness using the freshness relation. Reflecting these changes to our implementation allows us to build a derivation for Example 3.9.

Remark 3.5.5. We have formalised the proof of Theorem 3.3.1 in a simplified setting, by taking sets of variables in binding positions to be empty. For sequencing rules (rules (31) and (19)), addition of α-equivalence is not required for the proof of Theorem 3.3.1, since we *assume* elaborate modules. However, to *build* a derivation for static interpretation we must be able to α-rename sets of variables N and N' in Rule (56) appropriately. In order to achieve this, we can add additional premises allowing for α-rename such as those in Rule (63). The same applies to Lemma 3.3.8 (see Rule (39)).

Currently, the proof of Theorem 3.3.1 ignores issues related to α-conversion. However, we have an implementation of the rules with explicit α-equivalence demonstrating Example 3.9; these changes are not yet incorporated into the proof of Theorem 3.3.1.

3.5.4 Proof of Normalisation of Static Interpretation

The proof of static interpretation normalisation is carried out as it is described in Section 3.3.10. The logical relation is implemented as a fixpoint rather than as an inductive relation. The reason for this representation is essential. If the relation was defined as an inductive predicate, the definition would not pass the strict positivity constraint for inductive definitions in Coq. From the definition of our logical relation, it is straightforward to establish that the relation is well-formed because it is decreasing structurally in its left argument. For this reason, it can be expressed as a fixpoint definition, using also Coq's anonymous fix-construct, corresponding to the nested structure of the semantic objects. Unfortunately, we cannot keep our environment representation completely abstract, since we define the logical relation recursively on the structure of environments. Restrictions on fixpoint definitions in Coq require us to use a nested fixpoint on underlying structures in the definition of environments. Again, we use a pair-of-vectors view to define a corresponding nested fixpoint in the definition of the consistency relation of (Figure 3.12).

```
Fixpoint consistent_IEnv (E:Env) (IE:IEnv) : Prop :=
  match E, IE with
    EnvCtr TE VE ME MTE, IEnvCtr TE' IVE IME MTE' ⇒
    TE = TE'
    ∧ consistent_IVEnv VE IVE
    ∧ consistent_IMEnv ME IME
    ∧ MTE = MTE'
  end
with consistent_IMEnv (ME:MEnv) (IME:IMEnv) : Prop :=
  match ME, IME with
    MEnvCtr me, IMEnvCtr ime ⇒
    dom (SemObjects.VE._to me) = dom (SemObjects.VE._to ime) ∧
    match me,ime with
      VecEnv.mkVecEnv _  nn (exist _ ks _) vs,
      VecEnv.mkVecEnv _  nn' (exist _ ks' _) vs' ⇒
      (fix con_step {n n'} (l : Vec Mod n) (ll : Vec IMod n')
         : Prop :=
           match l,ll with
           | [],[] ⇒ True
           | m :: tl, im :: tl'  ⇒
             con_step tl tl' ∧  consistent_IMod m im
           | _,_ ⇒ False
           end) nn nn' vs vs'
    end
  end
with consistent_IMod (M:Mod) (IM:IMod) : Prop :=
  match M with
    NonParamMod E ⇒
    match IM with
      INonParamMod IE ⇒ consistent_IEnv E IE
    | IFtor _ _ _ _ _ _ ⇒ False
    end
  | Ftor T0 E (MSigma T M) ⇒
```

```
    exists IE0 mid mexp,
    IM = IFtor IE0 T0 E (MSigma T M) mid mexp
    ∧ forall IE,
       consistent_IEnv E IE →
        exists N IM c,
         (Mexp_int (addIEnvMid mid (INonParamMod IE) IE0) mexp N IM c
         ∧ consistent_IMod M IM)
end.
```

There are several design decisions that we want to emphasise. Instead of forcing two vectors to be of the same length in the `consistent_IMEnv` function, we just return False in case vectors are not aligned. This definition also makes use of a *concrete* representation of module type environments as a pair-of-vectors. We would not be able to use any *abstract* representation of environments (like a type constructor exposed through the module type) in the definition. The reason for that is that we would have to use a recursor provided by the abstract representation to define the inner fixpoint and Coq would not accept the definition as terminating.

Remark 3.5.6. Instead of the inner fixpoint `con_step` we would like to use a separately defined function stating that some predicate `(P : A → B → Prop)` holds for two vectors componentwise. The function is defined as follows:

```
Fixpoint Forall2_fix {A B} (P : A → B → Prop) (n n' : nat)
             (l : Vec A n) (ll : Vec B n') : Prop :=
      match l,ll with
      | [],[] ⇒ True
      | m :: tl, im :: tl' ⇒
        Forall2_fix P _ _ tl tl' ∧ P m im
      | _,_ ⇒ False
      end.
```

Unfortunately, this does not work for our definition. Probably, Coq does not unfold `Forall2_fix` here to see that the argument is decreasing. Although, sometimes Coq is able to do some unfoldings, when checking a definition for termination (see definition of `ntsize` in [Chl13, Section 2.8] and the definition of `action` in Section 3.5.3).

Although we cannot use `Forall2_fix` directly in the definition of the consistency relation, we can prove that the nested fixpoint `con_step` corresponds to `Forall2_fix`. This way we can use properties of `Forall2_fix` in the proofs related to the consistency relation. Particularly, we are interested in converting this "intensional" representation to the extensional one that relates two environments using environment membership:

```
Definition EnvRel {A B} (P : A → B → Prop)
                        (E : En.t A) (E' : En.t B) : Prop :=
  (forall k, In _ k E ↔ In _ k E')
  ∧ (forall k v v', look k E = Some v →
                    look k E' = Some v' →
                    P v v').
```

In most proofs we use properties of environments, which are stated in terms of `look`, and it is very unwieldy to use `Forall2_fix` in such proofs, since in this case proofs have to be carried out by induction on the structure of the underlying vector environment.

Putting it all together, we define equations allowing us to fold the nested fixpoint in the definition of the consistency relation:

```
Lemma Forall2_fix_fold_unfold {A B} (P : A → B → Prop) :
        Forall2_fix P =
        (fix ff {n n'} (l : Vec A n) (ll : Vec B n') : Prop :=
           match l,ll with
           | [],[] ⇒ True
           | v :: tl, v' :: tl' ⇒
             ff tl tl' ∧ P v v'
           | _,_ ⇒ False
           end).

Lemma ForallEnv2_fold_unfold {A B} (P : A → B → Prop) ve ve' :
        ForallEnv2_fix P ve ve'=
        match ve,ve' with
          VecEnv.mkVecEnv _  n (exist _ ks _) vs,
          VecEnv.mkVecEnv _  n' (exist _ ks' _) vs' ⇒
             Forall2_fix P n n' vs vs'
        end.
```

We also provide a logical equivalence between `Forall2_fix` and `EnvRel`:

```
Lemma ForallEnv2_fix_EnvRel_iff (A B : Type)
                                (P : A → B → Prop)
                                (ve : VecEnv A) (ve' : VecEnv B)
 : (dom ve = dom ve' ∧ ForallEnv2_fix P ve ve') ↔ EnvRel P ve ve'.
```

In proofs involving the consistency relation we use the following pattern:

- rewrite using fold/unfold lemmas;
- rewrite by `ForallEnv2_fix_EnvRel_iff`

In this way we bridge the gap between the "intensional" recursive definition and the "extensional" definition, allowing for more convenient reasoning.

We have kept the Coq development close to the representation in this chapter. However, the filtering relation in Figure 3.10 does not define a filtering algorithm directly, but serves as a *specification* for it. In the proof of normalisation of static interpretation, we have to show the existence of a filtered environment. Due to the limitations of Coq's fixpoint constructs, we have defined a filtering algorithm as an inductively defined relation. We consider it future work to investigate the use of general recursion in Coq for filtering, which would be useful for applying code extraction to obtain a certified static interpretation implementation. We believe it is a reasonable approach to separate the relational "declarative" definitions from definitions that compute for code extraction. One can then establish a correspondence between the relational and functional representations to show soundness of the implementation.

3.6 Related Work

The concept of static interpretation of modules is not new and has been applied earlier in the context of the MLKit Standard ML compiler [Els99]. In the present work we focus on the formalisation of a higher-order module language in the Coq proof assistant in the style of [Els99]. For that reason, as related work we mention mostly alternative approaches at providing mechanised meta-theories for module languages.

The work on compilation of higher-order modules into F_ω, the higher-order polymorphic lambda calculus [RRD10], comes with a Coq implementation of the work. Compared to our work, which eliminates all module language constructs at compile time, [RRD10] make no distinction between core language and module language constructs in the target code. Moreover, the style of formalisation is different from our Coq development since we use a more direct encoding of the module language semantics in term of semantic objects.

Earlier works include the work on using Twelf to provide a mechanised meta-theory for Standard ML [LCH07], based on Harper-Stone semantics of Standard ML [HS00]. Compared to our work, however, this approach also does not address the problem of eliminating modules at compile time.

Another body of work related to mechanising the meta-theory of ML is the work on CakeML [TMK+16], which, however, supports only non-parameterised modules.

The category of works related to our proof techniques and the approach to the formalisation includes works on reasoning with isomorphic representations of abstract data types in the context of homotopy type theory [Dan12, Dij13]. Nominal techniques are implemented in the Nominal Isabelle package [UT05, UK11] and to a limited extend in Coq (mostly focusing on simply-typed lambda calculus) [ABW07].

3.7 Conclusion and Future Work

We have developed a formalisation in Coq of a higher-order module system along with the static interpretation with the guarantee of termination in the style of [Els99]. Our implementation is one of the first attempts to formalise a module system in this style in the Coq proof assistant. In the course of the implementation we have developed the following techniques.

- Extension of the standard library implementation of sets and environments (finite maps) with extensionality property.
- Isomorphic representations of environments (finite maps) to overcome the issue with the conservative strict positivity check in Coq. The technique developed allowed us to implement semantic objects in Coq with almost no proof obligation overhead despite the fact of using different environment representations.
- Formalisation of nominal sets and applications of generalised nominal techniques allowing to work with sets of variables in binding positions. This is the first implementation in Coq dealing with generalised binders.

- The normalisation proof of the static interpretation using the Tait's method of logical relations [Tai67] in the setting of higher-order modules.

The current version of our Coq development is about 6.5k lines of code, excluding comments and blank lines. It includes definitions from Section 3.3, the proof of Theorem 3.3.1, the module implementing nominal techniques (with examples in the simplified setting discussed in Section 3.4), the module implementing a pair-of-vectors representation of environments, and the proof of strong normalisation for the simply-typed lambda calculus.

Although our implementation makes some simplifying assumptions, we believe it is can be extended to a full setting with no fundamental limitations. Particularly, the nominal techniques gives a uniform structuring principle for dealing with binders.

Moreover, our Coq implementation and the formal specification given in Section 3.3 has been developed hand-in-hand with the Haskell implementation integrated with the Futhark compiler, serving as a guiding line for the development. Using semantic objects allows for the structure of the Haskell implementation to be in a close correspondence with our Coq implementation.

As future work, we would like to extend our implementation of nominal sets with more features and eventually expose it as a library. Regarding the implementation of nominal techniques, we would like to note that a solution making use of type classes instead of modules to structure the library would be beneficial. Ideally, such an implementation would require finite sets to be implemented using type classes as well, which is not the case in the standard library of Coq, where they are implemented using modules.[4] We believe that such an implementation would allow for better proof automation, especially for proving properties such as equivariance.

Another direction of extension of our development in Coq would be an implementation of *algorithms* corresponding to the relational specifications of elaboration, filtering, and eventually, static interpretation. Having such implementations, one could use Coq's code extraction mechanism to obtain certified code in one of the target languages, which could be used as part of a compiler for the Futhark programming language.

[4] An experimental standalone implementation that uses Coq's type classes to implement nominal sets has been developed by the author, and it is available online: `https://github.com/annenkov/stlcnorm`. This implementation, however, still uses the module-based finite sets implementation from the standard library.

Chapter 4

Formalising Two-Level Type Theory

4.1 Introduction

Homotopy Type Theory (HoTT) is a variant of Martin-Löf type theory that pays particular attention to the equality (or identity) type. The equality type in Martin-Löf type theory is defined as an inductive family generated by the single constructor `refl`, called reflexivity. For any two inhabitants a and b of some type A, one can ask if a and b are equal by forming the equality type $a = b$. Since $a = b$ is also a type, one can ask if two proofs of equality are equal, i.e. for $p, q : a = b$, one can form $h : p = q$, and so on. The eliminator of the equality (or identity) types is usually called J (see rule ELIM-$=$ in Section 4.3). As it was observed by Hofmann and Streicher in [HS96], it is not possible from J to show that two proofs of equality are equal.

Therefore, there are two options: one could add axiom K (or equivalently, Uniqueness of Identity Proofs axiom), making any two proofs of equality equal, i.e.

$$\frac{\Gamma \vdash a_1, a_2 : A \qquad \Gamma \vdash p, q : a_1 = a_2}{\Gamma \vdash K(p, q) : p = q} \text{ UIP}$$

The other option is to allow for different ways to identify types by introducing the *univalence axiom* [Uni13]. The univalence axiom says that two types are considered equal when they are equivalent, reflecting the informal principle that is usually used in mathematics. That is, for any two types A and B

$$\mathit{Univalence} : (A = B) \simeq (A \simeq B)$$

More precisely, is says that a function $\mathit{idtoequiv}(A = B) \to (A \simeq B)$, which can be defined by induction on equality, is an equivalence. In other words there is another function, $ua : (A \simeq B) \to (A = B)$ that goes in the opposite direction and allows one to get proofs of equality from proofs of equivalence (see more on definition of equivalences in [Uni13, Chapter 4]).

Types in HoTT are weak ∞-groupoids, which allows for capturing important notions from homotopy theory and for developing it synthetically in type theory.

Homotopy type theory is implemented in a number of proof assistants allowing for development of machine-checkable proofs in such areas as homotopy

theory, category theory and other areas of mathematics. From the functional programming point of view, it also allows for solving abstraction problems in dependently typed programming, since equivalent types can be identified and changing between equivalent representation does not require additional efforts.

If we restrict ourselves to just sets (types, for which two proofs of equality of two elements are equal, also called hSets), the notion of equivalence will be just an isomorphism between types. In this setting, one can benefit from the fact that isomorphic types become equal in the presence of the univalence axiom. That means that all the proofs of properties of some type A are immediately transferred to types isomorphic to A, since we can always substitute equals for equals. This approach allows for better abstraction preservation in proof and program development. Particularly, one could have an abstract representation of some structure, for example environments (or finite maps), as we saw in Chapter 3. One can define several isomorphic representations satisfying the abstract specification and move between these representations using the fact that isomorphic types are considered equal. These ideas are considered in [Dan12, Dij13].

In our development of the Futhark module system formalisation described in Chapter 3 we could have used univalence to switch between the two environment representations easily, while keeping the specification completely abstract. Such a possibility would allow us to take advantage of computational behaviours of concrete representations, since it is impossible to compute with the specification given by abstract type or opaque module.

Another interesting feature of HoTT are higher inductive types (HITs). For instance, HITs allow for avoiding so called "setoid hell" by adding equality constructors to inductive definitions. A setoid is a set equipped with an equivalence relation. Whenever we want to compare two elements of setoid we have to use this custom equivalence relation instead of usual equality. Because of that all the operations on setoids have to respect its equivalence, which require a lot of proofs. Using HITs (to be precise, a specialised version called quotient inductive types, or QITs), one can use the usual propositional equality again when comparing two elements of a setoid.

Homotopy type theory is an active developing field with a number of open questions. Particularly, not being able to talk about strict equality in HoTT sometimes make certain constructions problematic. We will address this problem in the following sections.

The rest of the chapter is structured as follows. In Section 4.2 we discuss motivations for two-level type theory (2LTT). In Section 4.3 we provide a formal specification of 2LTT and discus differences with homotopy type system (HTS). Section 4.4 describes an internalisation of some results on inverse diagrams in 2LTT. The implementation of 2LTT in the Lean proof assistant is discussed in Section 4.5, which outlines the overall idea of the approach to the implementation of 2LTT in a proof assistant and then demonstrates features of our Lean development including the results from Section 4.4. Section 4.5 represents the main contribution of the author to the development of two-level type theory.

4.2 Motivation

The motivation for two-level type theory is twofold.

Many results of homotopy type theory are completely internal to HoTT and can be formalised directly in a proof assistant, and a lot of work has been done using Agda, Coq, and Lean. Some other results are only *partially* internal to HoTT. One example is the constructions of n-restricted semi-simplicial types which we can do only after fixing the number n *externally* (i.e. we have to decide which n we use before we start writing it down in a proof assistant). The reason, why such constructions are problematic to write in HoTT is that the usual definition of n-restricted semi-simplicial type as a *strict* functor from the category of finite non-empty ordinals and strictly monotone maps to the category of types: $S : \Delta_+^{\mathrm{op}} \to \mathcal{U}$ require functor laws to hold strictly (the functor laws correspond to semi-simplicial identities). In HoTT it would require infinite tower of coherencies ensuring that certain proofs of equality are equal, proofs of equality of proofs of equality are equal, and so on.

One can try to avoid writing equalities corresponding to the semi-simplicial identities by using an equivalent representation of n-restricted semi-simplicial types as a nested Σ-type with face maps being projections. For example, if we fix $n = 3$, we can write the following definition (we use Lean [dMKA+15] notation here):

```
definition SST₃ :=
  Σ (X₀ : Type)
    (X₁ : X₀ → X₀ → Type),
      Π (x₀ x₁ x₂ : X₀), X₁ x₀ x₁ → X₁ x₁ x₂ → X₁ x₀ x₁ → Type
```

In the definition for `SST₃` we think of `X₀` as points, `X₁` as lines, and `Π (x₀ x₁ x₂ : X₀), X₁ x₀ x₁ → X₁ x₁ x₂ → X₁ x₀ x₁ → Type` as triangles. To form a triangle, we need for the end of one side and the beginning of another side to be the same point. Instead of using semi-simplicial identities here (formulated using equalities), we just use the same point again for the point that should match the given point. For example `X₁ x₀ x₁` is the line that starts at `x₀` and ends at `x₁`, and `X₁ x₁ x₂` starts at `x₁` and ends at `x₂`.

Although it is possible to write such a definition for any fixed n, it seems to be not possible to generalise it for an arbitrary n internally in HoTT. For the detailed explanation see the introduction section in [ACK16].

Another example of this kind is the work by Shulman on inverse diagrams [Shu15], for which we can do constructions in type theory once we fix a (finite) inverse category *in the meta-theory*. One of the examples of the inverse diagrams is n-restricted semi-simplicial types described above.

In many situations, one would like such constructions to be completely internal (using a variable $n : \mathbb{N}$ or an inverse category expressed internally) and formalisable in a proof assistant, but unfortunately, it is either unknown how this is doable or it is known to be impossible. Two-level type theory gives a way to completely formalise such results. This is the aspect that we explore in the paper [ACK17].

A second motivation of two-level type theory is that it allows for extending homotopy type theory in a "controlled" way. It gives a framework that makes it easy to write down enhancements of the theory, where one can relatively easily

check whether these assumptions hold in some models (models are explored in [Cap16]).

In the present work we are focusing on the first motivation, namely on internalisation of results that can only be partially internalised in HoTT. We also will demonstrate how one can implement two-level type theory in a proof assistant and discuss our experience with developing a formalisation in such an implementation.

4.3 Two-Level Type Theory

To address the issues arising from the lack of strict equality, presented in Section 4.2, we introduce two level type theory, which consists of two fragments: a *strict* fragment (a form of MLTT with UIP) and a *fibrant* fragment (essentially HoTT). The fibrant fragment of our type theory has all the basic types and type formers found in HoTT[Uni13, Appendix A.2]:

- Π, the type former of dependent functions;
- Σ, the type former of dependent pairs;
- $+$, the coproduct type former;
- $\mathbf{1}$, the unit type;
- $\mathbf{0}$, the empty type;
- $\mathbb{N}$, the fibrant type of natural numbers;
- $=$, the equality type (in the sense of HoTT);
- a hierarchy $\mathcal{U}_0, \mathcal{U}_1, \ldots$ of universes;
- possibly inductive and higher inductive types.

Furthermore, we have:

- $+^{\mathrm{s}}$, the strict coproduct;
- $\mathbf{0}^{\mathrm{s}}$, the strict empty pretype;
- $\mathbb{N}^{\mathrm{s}}$, the strict pretype of natural numbers;
- $\stackrel{\mathrm{s}}{=}$, the strict equality;
- a hierarchy $\mathcal{U}_0^{\mathrm{s}}, \mathcal{U}_1^{\mathrm{s}}, \ldots$ of strict universes;
- possibly inductive types and quotient types.

We refer to the elements of $\mathcal{U}_i$ as *fibrant types*, while the elements of $\mathcal{U}_i^{\mathrm{s}}$ are *pretypes*. The intuition is that fibrant types are the usual types that are found

in HoTT, whereas pretypes are what one gets if one is allowed to talk about strict equality internally. The rules of $\overset{\mathrm{s}}{=}$:

$$\frac{\Gamma \vdash A : \mathcal{U}_i^{\mathrm{s}} \qquad \Gamma \vdash a, b : A}{\Gamma \vdash a \overset{\mathrm{s}}{=} b : \mathcal{U}_i^{\mathrm{s}}} \ \text{FORM-}\overset{\mathrm{s}}{=} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_a^{\mathrm{s}} : a \overset{\mathrm{s}}{=} a} \ \text{INTRO-}\overset{\mathrm{s}}{=}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma(b : A)(p : a \overset{\mathrm{s}}{=} b) \vdash P : \mathcal{U}_i^{\mathrm{s}} \qquad \Gamma \vdash d : P[a, \mathsf{refl}_a^{\mathrm{s}}]}{\Gamma(b : a)(p : a \overset{\mathrm{s}}{=} b) \vdash J_P^{\mathrm{s}}(d) : P} \ \text{ELIM-}\overset{\mathrm{s}}{=},$$

together with the judgmental computation rule:

$$J_P^{\mathrm{s}}(d)[a, \mathsf{refl}_a^{\mathrm{s}}] \equiv d.$$

Rules for fibrant equality look very similar to those for strict equality:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a_1, a_2 : A}{\Gamma \vdash a_1 = a_2 : \mathcal{U}_i} \ \text{FORM-}= \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_a : a = a} \ \text{INTRO-}=$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma(b : A)(p : a = b) \vdash P : \mathcal{U}_i \qquad \Gamma \vdash d : P[a, \mathsf{refl}_a]}{\Gamma(b : a)(p : a = b) \vdash J_P(d) : P} \ \text{ELIM-}=$$

It is important to note, that the rules FORM-$=$, INTRO-$=$, and ELIM-$=$ only involve fibrant types. For example, we cannot apply the equality type former to two elements of $\mathcal{U}_i^{\mathrm{s}}$. We assume that universes $\mathcal{U}_i$ are univalent, that is for any two types $X, Y : \mathcal{U}_i$, the map $(X = Y) \to (X \simeq Y)$ is an equivalence.

$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_i^{\mathrm{s}}} \ \text{FIB-PRE} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma.A \vdash B : \mathcal{U}_i}{\Gamma \vdash \Pi_A B : \mathcal{U}_i} \ \text{PI-FIB}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma.A \vdash B : \mathcal{U}_i}{\Gamma \vdash \Sigma_A B : \mathcal{U}_i} \ \text{SIGMA-FIB}$$

By the rule FIB-PRE, the type $a_1 = a_2$ is also a pretype, but it is different from the pretype $a_1 \overset{\mathrm{s}}{=} a_2$.

For pretypes $A, B : \mathcal{U}_i^{\mathrm{s}}$, we can form the pretype of strict isomorphisms, written $A \simeq^{\mathrm{s}} B$ (unlike in HoTT, it is enough to have maps in both directions such that both compositions are pointwise strictly equal to the identity). However, we do *not* assume that $\mathcal{U}_i^{\mathrm{s}}$ is univalent. Instead, we add rules corresponding to the principle of uniqueness of identity proofs UIP and function extensionality FUNEXT as follows:

$$\frac{\Gamma \vdash a_1, a_2 : A \qquad \Gamma \vdash p, q : a_1 \overset{\mathrm{s}}{=} a_2}{\Gamma \vdash K^{\mathrm{s}}(p, q) : p \overset{\mathrm{s}}{=} q} \ \text{UIP}$$

$$\frac{\Gamma \vdash f, g : \Pi_{a:A} B(a) \qquad \Gamma(a : A) \vdash p(a) : f(a) \overset{\mathrm{s}}{=} g(a)}{\Gamma \vdash \mathsf{funext}(p) : f \overset{\mathrm{s}}{=} g} \ \text{FUNEXT}$$

That is, strict equality serves as an internalised version of judgmental equality. These ideas of differentiating pretypes and fibrant types are inspired by

Voevodsky's *Homotopy Type System* (HTS) [Voe13]. Although, there are some important differences. In particular, two-level type theory does not assume the reflection rule for strict equality. Instead, we only require that it satisfies UIP. Another important difference is that HTS assumes that $\mathbf{0}$, $\mathbb{N}$, and $+$ from the fibrant fragment eliminate to arbitrary types. We do not assume that in two-level type theory, since all presented results do not depend on these assumptions. Moreover, we leave a possibility to add such assumptions, which makes two-level type theory a "framework" allowing to explore different variations of resulting type theory. For example, if we allowed for coercion from strict natural numbers $\mathbb{N}^{\mathrm{s}}$ to fibrant natural numbers $\mathbb{N}$ then from the construction of type of the n-restricted semi-simplicial types, one would get a type family $\mathbb{N} \to \mathcal{U}$ in the fibrant fragment.

4.4 Applications

One way to look at two-level type theory is to start with ordinary type theory, which correspond to the fibrant fragment, and then add parts of its meta-theory on top of it as an additional type-theoretic layer. This additional layer corresponds to the strict fragment, which can be used to capture meta-theoretic reasoning. This internalisation leads to a uniform treatment of results, which traditionally requires mixing external and internal reasoning. We show applicability of two-level type theory to these kinds of problems internalising some results on Reedy fibrant diagrams [Shu15]. Particularly, we define the notion of Reedy fibration, and show that Reedy fibrant diagrams $I \to \mathcal{U}$ have limits in $\mathcal{U}$, where I is a finite inverse category, and $\mathcal{U}$ is a universe of fibrant types. Let us first define notions required to formulate the theorem about Reedy fibrant diagrams, which is one of the central results of our Lean formalisation (we will discuss implementation details in Section 4.7). We closely follow the style of definitions given in [ACK17].

It is often not necessary to know that a pretype $A : \mathcal{U}^{\mathrm{s}}$ is a fibrant type. Instead, it is usually sufficient to have a fibrant type $B : \mathcal{U}$ and a strict isomorphism $A \simeq^{\mathrm{s}} B$. If this is the case, we say that A is *essentially fibrant*. Clearly, every fibrant type is also an essentially fibrant pretype.

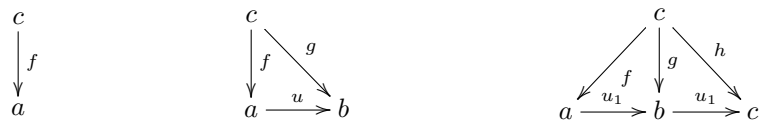
Recall that, in usual type-theoretic terminology, Fin_n is the finite type with n elements. In two-level type theory, for a strict natural number $n : \mathbb{N}^{\mathrm{s}}$, we have the finite type $\mathsf{Fin}^{\mathrm{s}}_n$. If we have inductive types in the strict fragment, we can define it as usual, but we do not need to: we can simply define it as the pretype of strict natural numbers smaller than n. Similarly, we have a fibrant type Fin_n (note that a strict natural number can always be seen as a fibrant natural number, but not vice versa).

We say that a pretype I is *finite* if we have a number $n : \mathbb{N}^{\mathrm{s}}$ and a strict isomorphism $I \simeq^{\mathrm{s}} \mathsf{Fin}^{\mathrm{s}}_n$.

Similar to essential fibrancy, we have the following definition:

Definition 4.4.1 (fibration; see [ACK17, Definition 4])**.** Let $p : E \to B$ be a function (with $E, B : \mathcal{U}^{\mathrm{s}}$). We say that p is a *fibration* for all $b : B$, the fibre of p over b, i.e. the pretype $\Sigma\,(e : E)\,.\,p(e) \stackrel{\mathrm{s}}{=} b$, is essentially fibrant.

We define the notion of a category in the strict fragment with categorical laws formulated using strict equality.



(a) objects (b) morphisms (c) composition

Figure 4.1: Coslice category.

Definition 4.4.2 (category; see [ACK17, Definition 7])**.** A *strict category* (or simply *category*) $\mathcal{C}$ is given by

- a pretype $|\mathcal{C}| : \mathcal{U}^{\mathrm{s}}$ of *objects*;
- for all pairs $x, y : |\mathcal{C}|$, a pretype $\mathcal{C}(x, y) : \mathcal{U}^{\mathrm{s}}$ of *arrows* or *morphisms*;
- an *identity* arrow $\mathsf{id} : \mathcal{C}(x, x)$ for every object x;
- and a *composition* function $\circ : \mathcal{C}(y, z) \to \mathcal{C}(x, y) \to \mathcal{C}(x, z)$ for all objects x, y, z;
- such that the usual categorical laws holds, that is, we have $f \circ \mathsf{id} \overset{\mathrm{s}}{=} f$ and $\mathsf{id} \circ f \overset{\mathrm{s}}{=} f$, as well as $h \circ (g \circ f) \overset{\mathrm{s}}{=} (h \circ g) \circ f$.

We say that a strict category is *fibrant* if the pretype of objects and the family of morphisms are fibrant.

The definition of the strict category corresponds to that of a *precategory* from [Uni13, Chapter 9].

In the following, we will usually drop the attribute "strict" and simply talk about *categories*. A canonical example of a category is the category of pretypes, whose objects are the pretypes in a given universe $\mathcal{U}^{\mathrm{s}}$, and morphisms are functions. By slight abuse of notation, we write $\mathcal{U}^{\mathrm{s}}$ for this category. The usual theory of categories can be reproduced in the context of our categories (at least as long as we stay constructive). In particular, one can define the notions of *functor*, *natural transformation*, *limits*, *adjunctions* in the obvious way, or show that limits (if they exist) are unique up to isomorphism, and so on.

In the context of Reedy fibrations, an important categorical construction is the following one:

Definition 4.4.3 (reduced coslice; see [ACK17, Definition 9])**.** Given a category $\mathcal{C}$ and an object $c : \mathcal{C}$, the *reduced coslice* $c /\!\!/ \mathcal{C}$ is the full subcategory of non-identity arrows in the coslice category $c/\mathcal{C}$. A concrete definition is the following. The objects of $c /\!\!/ \mathcal{C}$ are triples of an $y : |\mathcal{C}|$, a morphism $f : \mathcal{C}(x, y)$, and a proof $\neg \left(p_*(f) \overset{\mathrm{s}}{=} \mathsf{id}\right)$, for all $p : x \overset{\mathrm{s}}{=} y$, where p_* denotes the $\mathsf{transport}$ function $\mathcal{C}(x, y) \to \mathcal{C}(y, y)$. Morphisms between (a, f, s) and (b, g, s') are elements $u : \mathcal{C}(a, b)$ such that $u \circ f \overset{\mathrm{s}}{=} g$ in $\mathcal{C}$ (see Figure 4.1).

Notice that we have a "forgetful functor" $\mathsf{forget} : c /\!\!/ \mathcal{C} \to \mathcal{C}$, given by the first projection on objects as well as on morphisms.

Consider the category $(\mathbb{N}^{\mathrm{s}})^{\mathrm{op}}$ which has $n : \mathbb{N}^{\mathrm{s}}$ as objects, and

$$(\mathbb{N}^{\mathrm{s}})^{\mathrm{op}}(n, m) :\equiv n \geq^{\mathrm{s}} m$$

(the function $>^{\mathrm{s}} \colon \mathbb{N}^{\mathrm{s}} \to \mathbb{N}^{\mathrm{s}} \to \mathsf{Prop}^{\mathrm{s}}$ is defined in the canonical way). Then, we define:

Definition 4.4.4 (inverse category; see [ACK17, Definition 10])**.** We say that a category $\mathcal{C}$ is an *inverse category* if there is a functor $\varphi : \mathcal{C} \to (\mathbb{N}^{\mathrm{s}})^{\mathrm{op}}$ which reflects identities; i.e. if we have $f : \mathcal{C}(x, y)$ and $\varphi_x \stackrel{\mathrm{s}}{=} \varphi_y$, then we also have $p : x \stackrel{\mathrm{s}}{=} y$ and $p_*(f) \stackrel{\mathrm{s}}{=} \mathsf{id}_y$. We call φ the *rank functor*, and say that an object $i : |\mathcal{C}|$ has rank $\varphi(i)$.

Notice that *reflecting identities* usually means that f is an identity whenever $\varphi(f)$ is. In $(\mathbb{N}^{\mathrm{s}})^{\mathrm{op}}$, a morphism is an identity if and only if its domain and codomain coincide. Notice that the expression $f \stackrel{\mathrm{s}}{=} \mathsf{id}_y$ does not type-check, and to remedy this, we have to transport f along a strict equality between x and y, using the notation $p_*(f)$ from [Uni13].

Remark 4.4.1. There are several equivalent ways to define inverse categories. They are often characterised as dual to *direct* categories, which in turn can be described as not having an infinite sequence of non-identity arrows as in $\to\to\to \cdots$. Another way to formulate this following [Shu15] is "An inverse category is a category such that the relation 'x receives a nonidentity arrow from y' on its objects is well-founded." This formulation allows one to use well-founded induction to define diagrams on inverse categories. One important example of an inverse category is Δ_+^{op}, a category of finite non-empty ordinals and strictly monotone maps. That is, a functor $S : \Delta_+^{\mathrm{op}} \to \mathcal{U}$ is an inverse diagram on Δ_+^{op}.

4.4.1 Reedy Fibrant Limits

Recall that our first example of a category was a strict universe $\mathcal{U}^{\mathrm{s}}$ of pretypes and functions. Much of what is known about the category of sets in traditional category theory holds for $\mathcal{U}^{\mathrm{s}}$. For example, the following result translates rather directly:

Lemma 4.4.1 (see [ACK17, Lemma 11])**.** *The category $\mathcal{U}^{\mathrm{s}}$ has all* small *limits, where* small *means that the corresponding diagram has an index category whose objects and morphisms are pretypes in $\mathcal{U}^{\mathrm{s}}$.*

Proof. Let $\mathcal{C}$ be a category with $|\mathcal{C}| : \mathcal{U}^{\mathrm{s}}$ and $\mathcal{C}(x, y) : \mathcal{U}^{\mathrm{s}}$ (for all x, y). Let $X : \mathcal{C} \to \mathcal{U}^{\mathrm{s}}$ be a functor. We define L to be the pretype of natural transformations $1 \to X$, where $\mathbf{1} : \mathcal{C} \to \mathcal{U}^{\mathrm{s}}$ is the constant functor on $\mathbf{1}$. Clearly, $L : \mathcal{U}^{\mathrm{s}}$, and a routine verification shows that L satisfies the universal property of the limit of X. ∎

Unfortunately, the category $\mathcal{U}$ of fibrant types is not as well-behaved. Even pullbacks of fibrant types are not fibrant in general (but see Lemma 4.4.2). Since $\mathcal{U}$ is a subcategory of $\mathcal{U}^{\mathrm{s}}$, a functor $X : \mathcal{C} \to \mathcal{U}$ can always be regarded as a functor $\mathcal{C} \to \mathcal{U}^{\mathrm{s}}$, and we always have a limit in $\mathcal{U}^{\mathrm{s}}$. If this limit happens to be essentially fibrant, we say that X has a *fibrant limit*. Since $\mathcal{U}$ is a full subcategory of $\mathcal{U}^{\mathrm{s}}$, this limit will then be a limit of the original diagram $\mathcal{C} \to \mathcal{U}$.

Lemma 4.4.2 (see [ACK17, Lemma 12]). *The pullback of a fibration $E \to B$ along any function $f : A \to B$ is a fibration.*

Proof. We can assume that E is of the form $\Sigma\,(b : B)\,.\,C(b)$ and p is the first projection. Clearly, the first projection of $\Sigma\,(a : A)\,.\,C(f(a))$ satisfies the universal property of the pullback. ■

Lemma 4.4.2 makes it possible to construct fibrant limits of certain "well-behaved" functors from inverse categories.

In the subsequent definitions we always assume that $\mathcal{C}$ is an inverse category.

Definition 4.4.5 (matching object; see [ACK17, Definition 13] and [Shu15, Chapter. 11]). Let $X : \mathcal{C} \to \mathcal{U}^{\mathrm{s}}$ be a functor. For any $z : \mathcal{C}$, we define the *matching object* M_z^X to be the limit of the composition $z /\!\!/ \mathcal{C} \xrightarrow{\mathsf{forget}} \mathcal{C} \xrightarrow{X} \mathcal{U}^{\mathrm{s}}$.

Definition 4.4.6 (Reedy fibrations; see [ACK17, Definition 14] and [Shu15, Def. 11.3]). Let $X, Y : \mathcal{C} \to \mathcal{U}^{\mathrm{s}}$ be two diagrams (functors). Further, assume $p : X \to Y$ is a natural transformation. We say that p is a *Reedy fibration* if, for all $z : \mathcal{C}$, the canonical map

$$X_z \to M_z^X \times_{M_z^Y} Y_z,$$

induced by the universal property of the pullback, is a fibration.

A diagram X is said to be *Reedy fibrant* if the canonical map $X \to \mathbf{1}$ is a Reedy fibration, where $\mathbf{1}$ is of course the diagram that is constantly the unit type.

The following lemma will be useful for choosing an element with the maximal rank from an inverse category with non-empty finite type of objects.

Lemma 4.4.3. *For some inverse category $\mathcal{C}$ with non-empty finite type of objects, i.e. for any $n : \mathbb{N}^{\mathrm{s}}$, we have $\psi : |\mathcal{C}| \simeq^{\mathrm{s}} \mathsf{Fin}^{\mathrm{s}}_{n+1}$, and the rank functor $\varphi : \mathcal{C} \to (\mathbb{N}^{\mathrm{s}})^{op}$, we can chose an element $z : |\mathcal{C}|$ with the maximal rank, i.e. for any $c : |\mathcal{C}|$, $\varphi_c \leq \varphi_z$.*

Proof. We want to construct an inhabitant of the following type:

$$\Sigma\,(z : |\mathcal{C}|)\,.\,\Pi_{c:|\mathcal{C}|}\,\varphi_c \leq \varphi_z \tag{4.1}$$

We use

$$\begin{aligned} f &: |\mathcal{C}| \to \mathsf{Fin}^{\mathrm{s}}_{n+1} \\ g &: \mathsf{Fin}^{\mathrm{s}}_{n+1} \to |\mathcal{C}| \\ l &: \forall x, g(f\ x) \stackrel{\mathrm{s}}{=} x \\ r &: \forall y, f(g\ y) \stackrel{\mathrm{s}}{=} y \end{aligned}$$

for the components of the isomorphism ψ.
We proceed by induction on n.

Case 1 ($|\mathcal{C}| \simeq^{\mathrm{s}} \mathsf{Fin}^{\mathrm{s}}_1$): We take $z \equiv g\ 0^{\mathrm{s}}$. Thus, we have to show that for any $c : |\mathcal{C}|$ we have $\varphi_c \leq \varphi_{(g\ 0^{\mathrm{s}})}$.

$$\begin{aligned} c &\stackrel{\mathrm{s}}{=} \{\text{by left inverse } l\} \\ &\stackrel{\mathrm{s}}{=} g(f\ c) \\ &\stackrel{\mathrm{s}}{=} \{(f\ c) \stackrel{\mathrm{s}}{=} 0^{\mathrm{s}}, \text{ since } 0^{\mathrm{s}} \text{ is the only inhabitant of } \mathsf{Fin}^{\mathrm{s}}_1\} \\ &\stackrel{\mathrm{s}}{=} g\ 0^{\mathrm{s}} \end{aligned}$$

So, we get $\varphi_{(g\ 0^{\mathrm{s}})} \leq \varphi_{(g\ 0^{\mathrm{s}})}$ as required.

Case 2 ($|\mathcal{C}| \simeq^{\mathrm{s}} \mathsf{Fin}^{\mathrm{s}}_{(n'+1)+1}$): Let us pick some element $z' : |\mathcal{C}|$ (this is possible, since $|\mathcal{C}|$ is finite, for example, we could take $z' \equiv g\ 0^{\mathrm{s}}$, but this proof does not depend on our choice of an element of $\mathsf{Fin}^{\mathrm{s}}_{(n'+1)+1}$, to which we apply the function g). We call C' the category $\mathcal{C}$ with the element z' removed. Also, we have $\psi' : |\mathcal{C}|' \simeq^{\mathrm{s}} \mathsf{Fin}^{\mathrm{s}}_{n'+1}$. We call φ' a function φ restricted to $|\mathcal{C}'|$. By induction hypothesis with ψ' and φ' we have $z'' : |\mathcal{C}'|$ s.t.

$$\Pi_{c:|C'|} \varphi'_c \leq \varphi'_{z''} \tag{4.2}$$

We observe that $\varphi'_c \equiv \varphi_c$ and $\varphi'_{z''} \equiv \varphi_{z''}$, since both c and z'' are in $|\mathcal{C}|'$. We do not know how $\varphi_{z'}$ and $\varphi_{z''}$ are related, but since the order on $\mathbb{N}$ is decidable, we proceed by case analysis on $\varphi_{z'} \leq \varphi_{z''}$.

Subcase 1 $\varphi_{z'} \leq \varphi_{z''}$. We take $z \equiv z''$ in (1) We have to show that for any $c : |\mathcal{C}|$, $\varphi_c \leq \varphi''_z$. By case analysis on decidable equality we again get two cases:

- $c \stackrel{\mathrm{s}}{=} z''$. The claim follows from the assumption $\varphi'_z \leq \varphi''_z$.
- $c \neq^{\mathrm{s}} z''$. Since $\mathcal{C}'$ is a category without z'', we know that $c : |\mathcal{C}'|$. We complete the proof by (2).

Subcase 2 $\varphi'_z > \varphi''_z$. We take $z \equiv z'$ in (1). We have to show that for any $c : |\mathcal{C}|$, $\varphi_c \leq \varphi'_z$. We again proceed by case analysis on decidable equality.

- $c \stackrel{\mathrm{s}}{=} z'$. Follows immediately, since $\varphi'_z \leq \varphi'_z$.
- $c \neq^{\mathrm{s}} z'$. Again, we now that $c : |\mathcal{C}'|$, and complete the proof by (2).

■

Remark 4.4.2. Notice that in Lemma 4.4.3 we could not just pick the maximal element in $\mathsf{Fin}^{\mathrm{s}}_{(n+1)}$ and get an element in $\mathcal{C}$ with maximal rank, since we want the lemma to be independent of particular isomorphism ψ.

Using the definition of Reedy fibrations (Definition 4.4.6), we can make precise the claim that we can construct fibrant limits of certain well-behaved diagrams. The following theorem is (probably) the most involved result of our formalisation:

Theorem 4.4.1 (see [ACK17, Theorem 15] and [Shu15, Lemma 11.8])**.** *Assume that $\mathcal{C}$ is an inverse category with a finite type of objects $|\mathcal{C}|$. Assume further that $X : \mathcal{C} \to \mathcal{U}^{\mathrm{s}}$ is a Reedy fibrant diagram which is pointwise essentially fibrant (which means we may assume that it is given as a diagram $\mathcal{C} \to \mathcal{U}$).*

Then, X has a fibrant limit.

Proof. By induction on the cardinality of $|\mathcal{C}|$. In the case $|\mathcal{C}| \simeq^{\mathrm{s}} \mathsf{Fin}^{\mathrm{s}}_0$, the limit is the unit type.

Otherwise, we have $|\mathcal{C}| \simeq^{\mathrm{s}} \mathsf{Fin}^{\mathrm{s}}_{n+1}$. Let us consider the rank functor

$$\varphi : \mathcal{C} \to (\mathbb{N}^{\mathrm{s}})^{\mathrm{op}}.$$

We choose an object $z : \mathcal{C}$ such that φ_z is maximal using Lemma 4.4.3 Let us call $\mathcal{C}'$ the category that we get if we remove z from $\mathcal{C}$; that is, we set $|\mathcal{C}'| :\equiv \Sigma\,(x : |\mathcal{C}|)\,.\,\neg(x \overset{\mathrm{s}}{=} z)$. Clearly, $\mathcal{C}'$ is still inverse, and we have $|\mathcal{C}'| \simeq^{\mathrm{s}} \mathsf{Fin}^{\mathrm{s}}_n$.

Let $X : \mathcal{C} \to \mathcal{U}$ be Reedy fibrant. We can write down the limit of X explicitly as

$$\Sigma\left(c : \Pi_{y:|\mathcal{C}|} X_y\right).\,\Pi_{y,y':|\mathcal{C}|}\Pi_{f:\mathcal{C}(y,y')} Xf(c_y) \overset{\mathrm{s}}{=} c_{y'}. \tag{4.3}$$

Writing the pretype $|\mathcal{C}|$ as a coproduct $1 +^{\mathrm{s}} |\mathcal{C}|'$, we get that the above pretype is strictly isomorphic to

$$\begin{aligned}
&\Sigma\,(c_z : X_z)\,.\,\Sigma\left(c : \Pi_{y:|\mathcal{C}|'} X_y\right).\\
&\quad\left(\Pi_{f:\mathcal{C}(z,z)} Xf(c_z) \overset{\mathrm{s}}{=} c_z\right) \times\\
&\quad\left(\Pi_{y:|\mathcal{C}|'}\Pi_{f:\mathcal{C}(y,z)} Xf(c_y) \overset{\mathrm{s}}{=} c_z\right) \times\\
&\quad\left(\Pi_{y:|\mathcal{C}|'}\Pi_{f:\mathcal{C}(z,y)} Xf(c_z) \overset{\mathrm{s}}{=} c_y\right) \times\\
&\quad\left(\Pi_{y,y':|\mathcal{C}|'}\Pi_{f:\mathcal{C}(y,y')} Xf(c_y) \overset{\mathrm{s}}{=} c_{y'}\right).
\end{aligned} \tag{4.4}$$

Using that z has no incoming non-identity arrows (together with UIP), two of the components of the above type are contractible and can be removed, leaving us with

$$\begin{aligned}
&\Sigma\,(c_z : X_z)\,.\,\Sigma\left(c : \Pi_{y:|\mathcal{C}|'} X_y\right).\\
&\quad\left(\Pi_{y:|\mathcal{C}|'}\Pi_{f:\mathcal{C}(z,y)} Xf(c_z) \overset{\mathrm{s}}{=} c_y\right) \times\\
&\quad\left(\Pi_{y,y':|\mathcal{C}|'}\Pi_{f:\mathcal{C}(y,y')} Xf(c_y) \overset{\mathrm{s}}{=} c_{y'}\right).
\end{aligned} \tag{4.5}$$

Let us write L for the limit of X restricted to $\mathcal{C}'$, and let us further write p for the canonical map $p : L \to M^X_z$. Further, we write q for the map $X_z \to M^X_z$. Then, (4.5) is strictly isomorphic to

$$\Sigma\,(c_z : X_z)\,.\,\Sigma\,(d : L)\,.\,p(d) \overset{\mathrm{s}}{=} q(c_z). \tag{4.6}$$

Swapping sigmas in (4.6) gives us

$$\Sigma\,(d : L)\,.\,\Sigma\,(c_z : X_z)\,.\,p(d) \overset{\mathrm{s}}{=} q(c_z). \tag{4.7}$$

This is the pullback of the span $L \overset{p}{\to} M^X_z \overset{q}{\leftarrow} X_z$:

$$\begin{array}{ccc}
L \times_{M^X_z} X_z & \xrightarrow{\;g\;} & X_z \\
\Big\downarrow{\scriptstyle f} & & \Big\downarrow{\scriptstyle q} \\
L & \xrightarrow{\;p\;} & M^X_z
\end{array} \tag{4.8}$$

By Reedy fibrancy of X, the map q is a fibration. Thus, by Lemma 4.4.2, the map $f : \Sigma\,(c_z : X_z)\,.\,\Sigma\,(d : L)\,.\,p(d) \stackrel{\mathrm{s}}{=} q(c_z) \to L$ on (4.8) is a fibration. By the induction hypothesis, L is essentially fibrant. This implies that (4.6) is essentially fibrant, as it is the domain of a fibration whose codomain is essentially fibrant. ■

4.5 Formalisation in Lean

With a proof assistant that implements our two-level theory, one would thus be able to mechanise the results of the paper rather directly, or at least similarly directly as papers with purely internal results can be implemented in current proof assistants: of course, there is always still some work to do because some low-level steps are omitted in informal presentations. As we do not have such a proof assistant at hand, the task is to implement two level type theory in conventional proof assistants reusing as many features as possible.

An overall idea of an approach to implementation that is suitable for most existing proof assistants is the following. We work in a type theory with universes of "strict" types (i.e. where UIP holds). Pretypes correspond to the ordinary types of the proof assistant and (fibrant) types are represented as pretypes "tagged" with the extra property of being fibrant. The role of our strict equality is played by the ordinary propositional equality of the proof assistant (which, thanks to UIP, is indeed propositional in the sense of HoTT). The fibrant equality type is postulated together with its elimination rule J. We further postulate fibrancy preservation rules for Π and Σ as they are given in Section 4.3. The usual computation rule for J is defined using strict equality.

The proof assistant Lean [dMKA+15], which we have chosen for our formalisation,[1] can operate in two different "modes": one with a built-in UIP, and one which is suitable for HoTT. Our Lean implementation is based on "strict", Lean mode. That means that Lean's `Type` now becomes a pretype in two-level type theory sense.

Remark 4.5.1 (Notation)**.** Before we start describing our Lean development we introduce some notation used in the Lean code snippets:

- `A` $\to$ `B` for the type of functions from `A` to `B`
- `x` $\longrightarrow$ `y` for morphisms (`hom` field in the `category` structure)
- `C` $\Rightarrow$ `D` for functors from a category `C` to a category `D`
- `Nat(F,G)` for natural transformations from a functor `F` to a functor `G`
- `p` $\triangleright$ `a` for transport of `a` along the equality `p`

Fibrant types `Fib` are implemented using Lean dependent records with two fields: a pretype, and the property that it is fibrant:

```
structure Fib : Type := mk ::
  (pretype : Type)
  (fib : is_fibrant pretype)
```

[1] The code is available at `https://github.com/annenkov/two-level`.

The `is_fibrant` property is defined using the type class mechanism provided by the language.

```
constant is_fibrant_internal : Type → Prop

structure is_fibrant [class] (X : Type) := mk ::
  fib_internal : is_fibrant_internal X
```

We declare the `Fib.pretype` field to be a coercion, allowing to project a pretype out of `Fib`. For that purpose, we use the mechanism of *attributes*.

```
attribute Fib.pretype [coercion]
```

Such a declaration implements the FIB-PRE rule, which says that every fibrant type is also a pretype.

For the second component of the `Fib` structure, we define the following attribute:

```
attribute Fib.fib [instance]
```

This definition makes available for every inhabitant of `Fib` an instance of the `is_fibrant` type class.

The rules that Σ- and Π-types preserve fibrancy (rules SIGMA-FIB and PI-FIB) are also postulated and exposed as instances of the `is_fibrant` type class:

```
constant sigma_is_fibrant_internal {X : Type}{Y : X → Type}
  : is_fibrant X
  → (Π (x : X), is_fibrant (Y x))
  → is_fibrant_internal (Σ (x : X), Y x)

definition sigma_is_fibrant [instance] {X : Type}{Y : X → Type}
  [fibX : is_fibrant X] [fibY : Π (x : X), is_fibrant (Y x)] :
  is_fibrant (Σ (x : X), Y x) :=
    is_fibrant.mk (sigma_is_fibrant_internal fibX fibY)
```

The rule for Π-types is implemented in a similar way. In the same way we postulate that the unit type, equality types and `Fib` itself are fibrant.

The presentation of fibrant types using type classes results in a very elegant implementation of the fibrant fragment of the type theory. The class instance resolution mechanism allows us to leave the property of being fibrant implicit in most cases. We use `Fib` in definitions and let Lean insert coercions automatically in places where a pretype is expected, or where a witness that a type is fibrant is required. We will consider several examples showing how Lean's class resolution mechanism helps writing definitions involving fibrant types.

First, we declare some variables, which will be used in our examples:

```
variables {A : Fib} {B : Fib} {C : Fib} (P : A → Fib)
```

Now we can use these declarations in any definition in the same namespace and Lean will automatically add them as arguments to definitions that use them.[2] Another point to note here is that because of the `[instance]` attribute for the

[2] The details about namespaces, variables and other Lean features can be found in the Lean Tutorial `https://leanprover.github.io/tutorial/tutorial.pdf`

`Fib.fib` field, all instances of the `is_fibrant` type class for declared variables of type `Fib` are available for Lean's resolution mechanism.

Our fist example is an equivalence lemma known as associativity of the product type:

```
definition prod_assoc : A × (B × C) ≃ (A × B) × C := sorry
```

Since in this example we care only about the statement itself, and not about the proof, we will use the `sorry` keyword, which allows us to assume a definition. Because we state a fibrant equivalence between fibrant types in `prod_assoc`, both sides of the equivalence must be some fibrant types. All we know from the variable declarations above is that types `A`, `B`, and `C` are fibrant. To show that the product of these types is fibrant as well we would have to apply a special case of the SIGMA-FIB rule (which we call `prod_is_fibrant`) two times on each side. Thanks to the class instance resolution mechanism we can leave this task to Lean. We change some pretty-printing options to be able to see implicit arguments for the `prod_assoc` definition.

```
set_option pp.implicit true
set_option pp.notation false
```

Running the `check @prod_assoc` command gives us the following result:

```
prod_assoc :
Π {A} {B} {C},
  @fib_equiv (prod A (prod B C)) (prod (prod A B) C)
    -- inferred by Lean --
    (@prod_is_fibrant A (prod B C) (Fib.fib A) (@prod_is_fibrant
     B C (Fib.fib B) (Fib.fib C)))
    (@prod_is_fibrant (prod A B) C (@prod_is_fibrant A B (Fib.
    fib A) (Fib.fib B)) (Fib.fib C))
    ----------------------
```

This example shows nested applications of `prod_is_fibrant`, which were resolved automatically by Lean. The same resolution procedure allows for inferring implicit fibrancy conditions using fibrancy-preservation rules for different type formers. In the following example, expressing the universal property of the product type, rules PI-FIB and a special case of SIGMA-FIB are used.

```
definition prod_universal : (C → A × B) ≃ (C → A) × (C → B)
    := sorry
```

We can inspect the inferred implicit arguments again by running the `check @prod_assoc` command:

```
prod_universal :
Π {A} {B} {C},
@fib_equiv (C → prod A B) (prod (C → A) (C → B))
  -- inferred by Lean --
  (@pi_is_fibrant C (λ a, prod A B) (Fib.fib C)
    (λ x, @prod_is_fibrant A B (Fib.fib A) (Fib.fib B)))
  (@prod_is_fibrant (C → A) (C → B)
    (@pi_is_fibrant C (λ a, A) (Fib.fib C) (λ x, Fib.fib A))
      (@pi_is_fibrant C (λ a, B) (Fib.fib C) (λ x, Fib.fib B)))
  ----------------------
```

The following example highlights one of the reasons behind our choice of Lean for implementing two-level type theory.

```
variables  (Q : A → Type) [Π a, is_fibrant (Q a)]

definition pi_eq (f : Π (a :A), Q a) : f ∼ f := refl _
```

An experimental implementation in Agda uses a definition like the above, and the example failed to work in Agda. As it became clear later, the example failed to work because of a small difference in the inference of implicit arguments, That is, changing `Π a, is_fibrant (Q a)` to $\Pi\{a\}$`, is_fibrant (Q a)` (in the corresponding Agda code), would make the example be accepted by Agda (see also Section 4.8.2). In our Lean development `pi_eq` successfully type-checks, and Lean infers the following:

```
pi_eq : Π {A} Q [_inst_1] f,
  @fib_eq (Π a, Q a)
    -- inferred by Lean --
    (@pi_is_fibrant A Q (Fib.fib A) _inst_1)
    ----------------------
    f f
```

Where `_inst_1` is an automatically generated name for the instance of a type class corresponding to `[Π a, is_fibrant (Q a)]` appearing in the variables declaration.

Remark 4.5.2. It is worth pointing out, however, that in our formalisation we do not make a distinction between fibrant and essentially fibrant pretypes, having instead a single predicate `is_fibrant` to express this property. That is, every type which is strictly isomorphic to a fibrant type is also considered fibrant by the axiom we postulate in our implementation. This makes the development more convenient as long as we use essentially fibrant types for most of the results presented in the current formalisation. For instance, Theorem 4.4.1 and a number of auxiliary lemmas for this theorem involve essentially fibrant types.

4.6 Working in the Fibrant Fragment

To show how to work with the fibrant type theory, we have formalised some simple facts from the HoTT library. Our implementation shows that many proofs can be reused almost without change, provided that the same notation is used for basic definitions. For instance, we have ported some theorems about product types with only minor modifications. In particular, induction on fibrant equalities works as expected: we annotate the postulated elimination principle with the `[recursor]` attribute, and the `induction` tactic applies this induction principle automatically.

A point to note is that the computation (or β-) rule for the J eliminator of the fibrant equality type is implemented as a strict equality, using the propositional equality of the proof assistant. This means that the rule does not hold judgmentally. Consequently, this computation does a priori not happen automatically, and explicit rewrites along the propositional β-rules are needed in proof implementations. This and other issues of the same kind are addressed by

using one of Lean's proof automation features. We annotate all the "computational" rules with the attribute `[simp]`. This attribute is used to guide Lean's simplification tactic `simp` which performs reductions in the goal according to the base of available simplification rules. That allows us to use a simple proof pattern: do induction on relevant equalities and then apply the `simp` tactic. However, the `simp` tactic is not a well-documented feature of Lean. Sometimes it fails to simplify goals, and in such cases we apply repeated rewrites using propositional computation rules.

Let us consider the following example from the HoTT Lean library. Notice that we use $=$ for equality here and that it corresponds to the usual HoTT equality, since we are considering an example from the HoTT Lean library, which supports HoTT natively.

```
definition prod_transport (p : a = a') (u : P a × Q a) :
  p ▷ u = (p ▷ u.1, p ▷ u.2) :=
  by induction p; induction u; reflexivity
```

After applying induction on `p` and `u`, we get the following goal:

$$\texttt{refl}_a \triangleright (\texttt{a}_1, \texttt{a}_2) = (\texttt{refl}_a \triangleright (\texttt{a}_1, \texttt{a}_2).1, \texttt{refl}_a \triangleright (\texttt{a}_1, \texttt{a}_2).2)$$

Since we are in the HoTT mode of Lean, computation rule for transport holds judgmentally, and we can simplify the expression above to

$$(\texttt{a}_1, \texttt{a}_2) = ((\texttt{a}_1, \texttt{a}_2).1, (\texttt{a}_1, \texttt{a}_2).2)$$

This goal we can prove by $\text{refl}_{(a_1,a_2)}$, since $((\texttt{a}_1, \texttt{a}_2).1, (\texttt{a}_1, \texttt{a}_2).2)$ reduces to $(\texttt{a}_1, \texttt{a}_2)$. The `reflexivity` tactic performs these reduction steps and proves the goal. We use "$\sim$" in place of "$=$" for the fibrant equality in our two-level type theory.

In the fibrant fragment of type theory we can express this proof in the following form:

```
definition prod_transport (p : a ∼ a') (u : P a  × Q a) :
  p ▷ u ∼ (p ▷ u.1, p ▷ u.2) :=
  by induction p; induction u; repeat rewrite transport_β
```

In this case, after applying induction on `p` and `u`, we get the same goal as in the previous example (up to notation for the equality):

$$\texttt{refl}_a \triangleright (\texttt{a}_1, \texttt{a}_2) \sim (\texttt{refl}_a \triangleright (\texttt{a}_1, \texttt{a}_2).1, \texttt{refl}_a \triangleright (\texttt{a}_1, \texttt{a}_2).2)$$

The difference is that we cannot reduce this goal and apply `reflexivity`, since in the fibrant fragment computation rule for transport is defined using propositional equality. That is, simplification of the goal gives us only that:

$$\texttt{refl}_a \triangleright (\texttt{a}_1, \texttt{a}_2) \sim (\texttt{refl}_a \triangleright \texttt{a}_1, \texttt{refl}_a \triangleright \texttt{a}_2)$$

We can finish the proof with the help of Lean's proof automation by repeatedly applying rewriting with $\texttt{transport}_\beta$. An alternative would be to use the `simp` tactic to simplify the goal; in this case the proof will look very close to the original one:

```
  by induction p; induction u; simp
```

There is another issue which arises, particularly, when defining propositional β-rules for equality-dependent definitions. As an example let us consider an action on paths, which depends of some path p.

```
apd {X : Type} {P : X -> Fib} {x y : X}
    (f : Π x, P x) (p : x ∼ y) : p ▷ f x ∼ f y
```

When defining the computation rule for `apd` we would like to write the following:

$$\texttt{apd f refl}_x \overset{s}{=} \texttt{refl}_{(fx)},$$

Unfortunately, this term is not well-typed, since the left-hand side of the equation has type $\texttt{refl}_x \triangleright (fx) = fx$, while the right-hand side has type $fx = fx$, where $\triangleright$ stand for transport along the fibrant equality. In order to make this definition well-typed we have to apply explicitly the propositional computation rule for transport. This leads to the following equation:

$$\texttt{apd}_\beta\ \texttt{\{P : X} \to \texttt{Fib\} (f : Π x, P x) \{x y : X\} :}$$
$$\texttt{(transport}_\beta\ \texttt{(f x))} \triangleright_s \texttt{(apd f refl}_x\texttt{)} \overset{s}{=} \texttt{refl}_{(fx)}$$

where $\triangleright_s$ denotes transport along the strict equality, i.e. Lean's propositional equality (we could have transported the right-hand side instead). Writing definitions like that is inconvenient, but there is a way to avoid it. We can define propositional β-rules only for some basic cases (like transport) and unfold definitions in proofs to a form for which these basic rules can be applied. We tested this strategy while porting some theorems about Σ-types from Lean's HoTT library. In general, this issue could appear in more complex cases than those we have investigated; it is similar to the problem appearing in axiomatic definitions of higher inductive types in Coq [Bar13], where a proposed solution has been to use private inductive types (see section 4.8.1).

4.7 Internalising the Inverse Diagrams

This section describes the details of the implementation of results from Section 4.4. As we are working in strict Lean, we have decided to use the existing formalisation of category theory from the standard library.[3] Unfortunately, it is not as developed as the formalisation in HoTT Lean. For that reason, additional effort was needed to formalise some concepts from category theory required for the results given in the paper.

The following definitions from the Lean standard library are used in our formalisation:

- categories;
- functors;
- natural transformations.

We had to implement the following notions:

- pullbacks and general limits;
- construction of the limit for Pretype category;
- coslice and reduced coslice;

[3] The standard library is part of the Lean's distribution. The source code in available at `https://github.com/leanprover/lean2`.

- matching object;
- inverse categories

In addition, we have proved some properties of the strict isomorphism and finite sets.

We will outline some definitions from the Lean's standard library and then discuss in details what we have implemented ourselves.

A definition of a category from Lean standard library corresponds to the notion of strict category (Definition 4.4.2). The usual approach to encode algebraic structures as dependent records (or structures) is used in the Lean standard library. Dependent records, being a generalisation of Σ-types, allow for fields in the definition to depend on previously defined fields. In this way, one can combine operations and laws that these operations must satisfy. Usually, laws are expressed in the form of propositions (type `Prop` in Lean).

In particular, a category is defined as follows:

```
structure category [class] (ob : Type) : Type :=
  (hom : ob → ob → Type)
  (comp : Π{|a b c : ob|}, hom b c → hom a b → hom a c)
  (ID : Π (a : ob), hom a a)
  (assoc : Π {|a b c d : ob|} (h : hom c d)
           (g : hom b c) (f : hom a b),
     comp h (comp g f) = comp (comp h g) f)
  (id_left : Π {|a b : ob|} (f : hom a b), comp !ID f = f)
  (id_right : Π {|a b : ob|} (f : hom a b), comp f !ID = f)
```

The definition specifies a category for the pretype of objects `ob` (note that `Type` in our Lean formalisation corresponds to pretypes in two-level type theory). Other fields in the definition follow Definition 4.4.2 quite closely.

The definition of functors follows the same idea of using dependent records. Functors are structures with four fields, defining how the functor acts on objects and morphisms, along with two usual functor laws. Natural transformations could have been represented as structures as well, but library implementors have chosen an equivalent representation as an inductive data type with one constructor taking two arguments: the components and the property, expressing naturality square. Since laws in the definitions of our structures have type `Prop`, and in the strict Lean mode we have proof irrelevance, to prove that two inhabitants of the structure are (strictly) equal it is sufficient to show that only components, for which these laws are defined are equal. Moreover, Lean treats any two propositions of the same type as definitionally equal, allowing for more proofs to be completed by computation. For example, for any two natural transformations `N M : Nat(F,G)`, for some functors `F` and `G`, we have the following:

```
natural_map N = natural_map M → N = M
```

Where `natural_map : Nat(F,G) → Π (a : C), hom (F a) (G a)` projects components from a given natural transformation.

We defined a reduced coslice directly following Definition 4.4.3 as a coslice with an additional property.

```
structure coslice_obs {ob : Type} (C : category ob) (a : ob) :=
```

```
(to  : ob)
(hom_to : hom a to)

open coslice_obs

structure red_coslice_obs {A : Type} (C : category A) (c : A)
    extends coslice_obs C c :=
(RC_non_id_hom : Π (p : c = to), ¬ (p ▷ₛ hom_to = category.id))
```

We use an inheritance mechanism here to add an additional property to the regular coslice definition. The `rc_non_id_hom` property states that a morphism `hom_to` cannot be the identity morphism. We use transport along the strict equality ($\triangleright_s$) to make the definition well-typed.

The reduced coslice forms a category, which is a full subcategory of the coslice category, although, we do not use this fact in the implementation. Instead, we define the reduced coslice category directly.[4] We will show the full definition of the reduced coslice category to demonstrate some Lean's features.

```
definition reduced_coslice {ob : Type}
                           (C : category ob) (c : ob)
  : category (red_coslice_obs C c) :=
{| category,
  hom := λa b, Σ(u : hom (to a) (to b)),
           u ∘ hom_to a = hom_to b,
  comp := λ a b c g f,
  ⟨ (pr1 g ∘ pr1 f),
    (show (pr1 g ∘ pr1 f) ∘ hom_to a = hom_to c,
    proof
      calc
        (pr1 g ∘ pr1 f) ∘ hom_to a = pr1 g ∘ (pr1 f ∘ hom_to a):
    eq.symm !assoc
         ... = pr1 g ∘ hom_to b : {pr2 f}
         ... = hom_to c : {pr2 g}
      qed) ⟩,
  ID := (λ a, ⟨ id, !id_left ⟩),
  assoc := (λ a b c d h g f, sigma.eq !assoc !proof_irrel),
  id_left := (λ a b f, sigma.eq !id_left !proof_irrel),
  id_right := (λ a b f, sigma.eq !id_right !proof_irrel) |}
```

The morphisms in the this category are commutative triangles with a "tip" c (Figure 4.1b), which we define as

```
Σ(u : hom (to a) (to b)), u ∘ hom_to a = hom_to b
```

The first projection is a morphism u between the codomains of morphisms going from c (a morphism $u : a \longrightarrow b$ in Figure 4.1b). The second projection is an equation corresponding to the commutative triangle.

This definition uses one convenient feature of Lean, namely the `calc` environment. The `calc` environment allows one to combine a sequence of equations

[4] this code is a slightly modified definition of the coslice category from Lean's standard library, which was commented out for some reason unknown to the author.

combined using transitivity of equality. Moreover, the same reasoning is possible with any transitive relation, and we are going to use it later for reasoning with isomorphism.

To construct a composition of morphisms in the category of (reduced) coslices we have to show that two commutative triangles with one common side form a bigger commutative triangle. The angle brackets notation $\langle a,\ b\rangle$ is used to construct a Σ-type. In the definition of `comp` the first component is just a composition of two morphisms (`pr1 f` and `pr1 g` correspond to morphisms $u_1 : a \longrightarrow b$ and $u_2 : b \longrightarrow c$ in Figure 4.1c), and the second component must be a proof that we get a commutative triangle with a composition `pr1 g` $\circ$ `pr1 f` as a bottom side. This proof is carried out in three steps using reasoning in the `calc` environment. First, we use associativity of the function composition (`assoc`) to rearrange the composition. After that we rewrite using the commutative triangle for f (`r2 fpr2 f`), and then complete the proof by rewriting with the commutative triangle for g (`r2 gpr2 g`).

The proofs of the categorical laws `assoc`, `id_left`, `id_riht` for `reduced_coslice` boils down to respective properties of morphisms in `C`. To "lift" these properties to morphisms in `c//C` the property of path in Σ-type `sigma.eq` is used.

Before we define inverse categories, we have to define a category $(\mathbb{N}^{s})^{op}$:

```
definition nat_cat_op [instance] : category ℕ :=
  {| category,
    hom := λ a b, a ≥ b,
    comp := λ a b c, @nat.le_trans c b a,
    ID := nat.le_refl,
    assoc := λ a b c d h g f, eq.refl _,
    id_left := λ a b f, eq.refl _,
    id_right := λ a b f, eq.refl _ |}

definition ℕop : Category := Mk nat_cat_op
```

In our Lean implementation we use `ℕop` to denote $(\mathbb{N}^{s})^{op}$. A morphism between two objects in `ℕop` is the $\geq$-relation on natural numbers. composition is given by transitivity of $\geq$, identity morphism is reflexivity of $\geq$ (we use properties of $\geq$-relation from the standard library), and proofs of other properties is just `eq.refl` meaning that they hold definitionally in Lean. This is a consequence of what we mentioned before: two inhabitants of the same proposition are definitionally equal in Lean. Let us consider a case for associativity. For any `f : a ≥ b, g : b ≥ c, h : c ≥ d` we have to show that

```
nat.le_trans h (nat.le_trans g f) : a ≥ d =
nat.le_trans (nat.le_trans h g) f : a ≥ d
```

Since both sides have the same type `a ≥ d`, and this is a proposition, from Lean's point of view they represent the same value.

We define the property of a functor that it reflects identities in the following way:

```
definition id_reflect {C D: Category} (φ : C ⇒ D) :=
    Π {|x y : C|} (f : x ⟶ y), (Σ (q : φ x = φ y), q ▹ φ f =
    id) → Σ (p : x = y),  p ▹ f = id
```

The definition of inverse categories uses a specialised version of `id_reflect` for the case of `Nop`, which does not require φ `f` to be an identity.

```
definition id_reflect_Nop {C : Category} (φ : C ⇒ Nop) :=
  Π {|x y : C|} (f : x ⟶ y), φ x = φ y → (Σ (p : x = y), p
  ▷ f = id)
```

We use `id_reflect` in our Lean implementation, and show that these two definitions are logically equivalent. The proof of that uses the fact that the only morphism `f : x ⟶ x` in `Nop` is the identity morphism.

Now we can define inverse categories by equipping a category `C` with a rank functor.

```
structure has_idreflect [class] (C D : Category) :=
    (φ : C ⇒ D)
    (reflecting_id : id_reflect φ)

structure invcat [class] (C : Category) :=
    (reflecting_id_Nop : has_idreflect C Nop)
```

For the definition of the matching object we use the previously constructed limit in the Pretype category (see remark 4.7.1).

```
lemma limit_nat_unit {C : Category.{1 1}}
                     (X : C ⇒ Type_category) (z : C)
      : limit_obj (limit_in_pretype X) = Nat(𝟭,X) := rfl

definition matching_object.{u} {C : Category.{1 1}} [invcat C]
   (X : C ⇒ Type_category.{u}) (z : C) :=
   Nat(𝟭, (X ∘f (forget C z)))
```

Where $\mathbf{1}$: `C` $\Rightarrow$ `Type_category` is a functor, which is constantly the unit type. The functor `forget` is defined exactly as specified in Section 4.4: an action on objects is the projection `to` of the `coslice_obs` structure, and on morphisms the action is a projection of the "bottom" morphism of the commutative triangle (the morphism u in Figure 4.1b), which is a first component of the Σ-type defining morphisms in `reduced_coslice`.

```
definition forget (C : Category) (c : C) : (c // C) ⇒ C :=
  {| functor,
    object := λ a, to a,
    morphism := λ a b f, pr1 f,
    respect_id := λa, eq.refl _,
    respect_comp := λ a b c f g, eq.refl _ |}
```

Remark 4.7.1. [Limits and Pullbacks] The `limit.lean` contains general definition of limits as a terminal object in the category of cones. We also define here an explicit representation of limits (see (3)) along with the proof that this definition is isomorphic to our general definition. We also construct limits in the category of pretypes and show that the limit of the diagram $\mathbf{2} \to \mathcal{U}^s$ is isomorphic to the product pretype. The `pullback.lean` file contains definitions of pullbacks constructed in different ways along with proofs that the definitions are isomorphic.

We chose to specialise Definition 4.4.6, by taking Y to be a constant functor to the unit type, instead of implementing a general definition of a Reedy fibration. That is, we define the canonical map $X_z \to M_z^X$ as the following:

```
definition matching_obj_map {C : Category.{1 1}}
                            [invC : invcat C]
                            (X : C ⇒ Type_category) (z : C)
    : X z → matching_object X z :=
      λ x, natural_transformation.mk (λ a u, X (hom_to a) x)
        begin
        -- proof of naturality is omitted
        end
```

Since matching object is the limit in the Pretype category, it is a natural transformation. We give only the natural map in the definition, omitting the proof of the naturality condition for brevity.

In our implementation we use the following definition of a fibration:

```
definition is_fibration_alt [reducible] {E B : Type} (p : E → B)
  := Π (b : B), is_fibrant (fibre_s p b)
```

Where $\texttt{fibre}_s$ is a "strict" fibre, that is a fibre defined using strict equality:

```
definition fibre_s {X Y : Type} (f : X → Y) (y : Y)
  := Σ (x : X), f x = y
```

Now we have all the ingredients to write a definition of Reedy fibrant diagrams.

```
definition is_reedy_fibrant [class] (X : C ⇒ Type_category) :=
    Π z, is_fibration_alt (matching_obj_map X z)
```

4.7.1 Proof of the Fibrant Limit Theorem

In the current implementation, besides the general two-level framework, we have implemented the machinery required to define Reedy fibrant diagrams and have fully formalised a proof of Theorem 4.4.1. The formalisation of Theorem 4.4.1 closely follows the structure given in the paper [ACK17].

Let us consider some steps of the proof in details. The base case was relatively easy to prove, and we will focus on the inductive step. We use the lemma `max_fun_to_ℕ` to pick an element with the maximal rank from the category $\mathcal{C}$. The way we do it closely corresponds to the proof of Lemma 4.4.3 Surprisingly, proving that after removal of z from $\mathcal{C}$ the resulting $\mathcal{C}'$ is finite, inverse and that diagram $X' : \mathcal{C}' \to \mathcal{U}^{\mathrm{s}}$ is still Reedy fibrant required writing a lot of boilerplate code (see Remark 4.7.2).

The overall idea of the proof of the inductive step is to show that our goal is strictly isomorphic to some fibrant type. Thus, one of the central parts of the proof was a transformation of the limit of a Reedy fibrant diagram through the chain of strict isomorphisms. In our Lean formalisation, it is implemented using the `calc` environment, which gives a very convenient way of chaining transitive steps.

We proved each step of the reasoning using isomorphisms in a separate lemma and then chained them together in the `calc` environment. The most

essential transformations are the `limit_two_piece_limit_equiv` and the `two_piece_limit_pullback_p_q_equiv` lemmas.

The `limit_two_piece_limit_equiv` lemma allows us to represent the limit as a product of two parts:

```
(Σ (c : Π y, X y), Π y y′ f, morphism X f (c y) = c y′)
  ≃ₛ
(Σ (c_z : X z) (c : (Π y : C_without_z z, X y)),
   (Π (y : C_without_z z) (f : z ⟶ obj y ), X f c_z = c y) ×
   (Π (y y′ : C_without_z z) (f : @hom (subcat_obj _ _) _ y y′),
      (Functor_from_C′ z X) f (c y) = c y′))
```

Listing 4.1: Limit isomorphism

Where `C_without_z z` corresponds to the category $\mathcal{C}'$ (a category $\mathcal{C}$ with z removed) in Theorem 4.4.1. We explicitly construct a functor from the category $\mathcal{C}'$ using the functor $X : C \to \mathcal{U}^s$. This lemma also implicitly includes the step (4) (see Theorem 4.4.1). While constructing the isomorphism given in Listing 4.1 we perform a case analysis on the equalities (since they are decidable) $y = z$ and $y' = z$, which gives us four cases corresponding to the product of four parts in 4.4. In case where $y = z$ and $y' = z$, we use the property that the only morphism from z to z in $\mathcal{C}$ is the identity morphism. The case where $y \neq z$ and $y' = z$ is impossible, since we have a morphism $f : y \longrightarrow z$, and z has a maximal rank, the only morphism with z as a codomain is the identity morphism, but we know that $y \neq z$.

For the next transformation step, we use the `two_piece_limit_pullback_p_q_equiv` lemma, which allows us to get (6):

```
Σ (c_z : X z) d, p d = q c_z
```

Where `p` and `q` are defined as the following:

```
  q := matching_obj_map X z
  p := map_L_to_Mz_alt z X
```

In the Lean implementation we use a tactic-level `let` construct to declare these maps. It is important to use `let` here, although we could use the `have` construct instead. The difference is that `let` allows us to keep definitions transparent for the simplification.

The map `p` is a map from the limit of the diagram `X` restricted to the category `C_without_z z` to the matching object. We use explicit an representation of the limit as a natural transformation:

```
definition lim_restricted (X : C ⇒ Type_category) (z : C)
                          [invC : invcat C]
:= Σ (c : Π y, (Functor_from_C′ z X) y),
     Π (y y′ : C_without_z z) (f : @hom (subcat_obj C _) _ y y′),
       ((Functor_from_C′ z X) f) (c y) = c y′

definition map_L_to_Mz_alt (z : C) (X : C ⇒ Type_category.{u})
                           [invC : invcat C]
                           (L : lim_restricted X z)
 : matching_object X z :=
    match L with
```

```
| ⟨η, NatSq⟩ :=
    by refine natural_transformation.mk
       (λ a u, η (mk _ (reduced_coslice_ne z a)))
       (λ a b f, funext (λ u, NatSq _ _ _))
end
```

To show that `lim_restricted X z` is fibrant we would like to use the induction hypothesis, but in order to do that we first have to show that `(Functor_from_C' z X)` is Reedy fibrant and that `C_without_z z` is still a category with a finite object type. We prove this facts in the separate lemmas `Functor_from_C'_reedy_fibrant` and `C_without_z_is_obj_finite`, respectively.

Remark 4.7.2. From the code above, one can see, that removing the element from `C` requires us to show to propagate this change through all the layers, such as definitions of functors and limits, properties if category `C_without_z z` etc. These changes are usually considered "obvious" on paper, but in the formal setting in a proof assistant could require significant efforts. It was important to write the definitions related to these lemmas in such a way that they can be simplified as much as possible using Lean's definitional equalities. Currently, this part of the implementation is the least readable part. Probably, there is a way to generalise this by developing suitable machinery to work with subcategories, but we did not explore that possibility.

By rearranging sigmas using `sigma_swap` lemma, we get the following:

```
(Σ (c_z : X z) (d : lim_restricted X z), p d = q c_z) ≃ₛ
(Σ (d : lim_restricted X z) (c_z : X z), q c_z = p d)
```

And this is exactly the pullback of the span

```
lim_restricted X z ⟶ matching_object X z ⟵ X z
```

By the lemma 4.4.2 (which we call `Pullback'_is_fibrant`), and we know that the map

```
(Σ (d : lim_restricted X z) (c_z : X z), q c_z = p d) ->
  lim_restricted X z}
```

is a fibration.

To complete the proof, we need to show that the domain of a fibration with fibrant codomain is fibrant, i.e.

```
Π (p : E → B), is_fibration_alt p → is_fibrant E
```

In order to do this we make use of the fact that we can "contract" some parts of the type. This is known as a *singleton contraction* (see Lemma 3.11.8 in [Uni13]):

```
definition singleton_contrₛ [instance] {A}
  : Π b, (Σ (a : A), b = a) ≃ poly_unit
```

Here `poly_unit` is a universe-polymorphic unit type. We mark our definition with the `[instance]` attribute to make this definition available for Lean's type class instance resolution mechanism. The full proof of the lemma looks very concise:

```
definition singleton_contr_fiber_s {E B : Type} {p : E → B}
  : (Σ b, fibre_s p b) ≃_s E :=
    calc
    (Σ b x, p x = b) ≃_s (Σ x b, p x = b) : _
                ...  ≃_s (Σ (x : E), poly_unit) : _
                ...  ≃_s E : _
```

This lemma shows another example where type classes are convenient to use in proofs. All the witnesses for the proof steps are inferred automatically on the base of available instances of the strict isomorphism. For the second step, Lean's inference is able to infer that we first need to apply a congruence for Σ-types and then use a singleton contraction lemma. The resulting term, which was constructed by Lean looks as follows (showing all the implicit arguments and replacing notation with textual representation):

```
@sigma_congr_2 E (λ x, @sigma B (@eq B (p x)))
                 (λ x, poly_unit)
                 (λ a, @singleton_contr_s B (p a))
```

Now we can show, that for any fibration `p : E → B` and fibrant `B`, the type `E` is fibrant. First, use `singleton_contr_fiber_s` to show `E` is strictly isomorphic to `(Σ b, fibre_s p b)`, and this type is fibrant, since Σ-type preserve fibrancy, `B` is fibrant and `Π (b : B), is_fibrant (fibre_s p b)` from the fact that `p` is a fibration. Again, in Lean it is sufficient to give a hint, which type `E` is equivalent to, and the rest could be resolved automatically. The resulting proof looks very concise:

```
definition fibration_domain_is_fibrant {E : Type} {B : Fib} :
  Π (p : E → B), is_fibration_alt p → is_fibrant E
    := λp is_fibr_p, @equiv_is_fibrant (Σ b, fibre_s p b) _ _ _
```

Application of this lemma concludes the proof of the Theorem 4.4.1.

4.7.2 Additional facts

In the proof of Theorem 4.4.1 we use some properties of strict isomorphism and finite sets. These lemmas can be found in the files `facts.lean` and `finite.lean` of our implementation. For the strict isomorphism (which is called `equiv` in the Lean's standard library), we implemented congruence lemmas for Π- and Σ-types.

```
pi_congr_1 [instance] {F' : A'  → Type} [φ : A ≃ A']
    : (Π (a : A), F' (φ • a)) ≃ (Π (a : A'), F' a)

pi_congr_2 [instance] {F G : A → Type}  [φ : Π a, F a ≃ G a]
    : (Π (a : A), F a) ≃ (Π (a : A), G a)

pi_congr [instance] {F : A → Type} {F' : A' → Type}
                    [φ : A ≃ A'] [φ' : Π a, F a ≃ F' (φ • a)]
    : (Π a, F a) ≃ Π a, F' a

sigma_congr_1 [instance] {A B: Type} {F : B → Type} [φ : A ≃ B]
    : (Σ a : A, F (φ • a)) ≃ Σ b : B, F b
```

```
sigma_congr₂ [instance] {A : Type} {F G : A → Type}
                        [φ : Π a : A, F a ≃ G a]
    : (Σ a, F a) ≃ Σ a, G a

sigma_congr {A B : Type} {F : A → Type} {G : B → Type}
            [φ : A ≃ B] [φ' : Π a : A, F a ≃ G (φ • a)]
    : (Σ a, F a) ≃ Σ a, G a
```

We used these lemmas in many proofs including the proofs of Theorem 4.4.1, where transformation thorough the sequence of isomorphisms is an important part of the proof. As mentioned before, properties of the isomorphism are instances of a corresponding type class. That allows for proving some goals involving isomorphisms automatically.

Properties of finite sets required for our development are related to the removal of an element from a given finite set.

```
  fin_remove_max {n : ℕ} :
      (Σ i : fin (nat.succ n), i ≠ maxi) ≃ fin n

  fin_remove_equiv {n : ℕ } (z : fin (nat.succ n))
    : (Σ i : fin (nat.succ n), i ≠ z) ≃ fin n
```

The `fin_remove_max` lemma shows that removing the maximal element gives us a finite set of smaller cardinality, and `fin_remove_equiv` generalises this result to the removal of any element. The proof of the last lemma uses several additional lemmas allowing us to manipulate finite sets using transpositions:

```
definition fin_transpose {n} (i j k : fin n) : fin n :=
    match fin.has_decidable_eq _ i k with
    | inl _ := j
    | inr _ := match fin.has_decidable_eq _ j k with
               | inl _ := i
               | inr _ := k
               end
    end
```

4.8 Other Formalisations

4.8.1 The Boulier-Tabareau Coq Development

Boulier and Tabareau [BT16] have implemented a theory with two equalities in the Coq proof assistant [BC10]. It uses an approach that is somewhat similar to our own development of two-level type theory. In particular, the authors use Coq type classes to track fibrant types and exploit the corresponding features of the type class resolution mechanism to derive fibrancy automatically. However, there are some differences in the details of how the fibrant equality type is implemented.

In our Lean development we postulate a fibrant equality type and the equality eliminator, while in [BT16] the authors define it as a *private* inductive type [Ber13]. This feature of the Coq proof assistant allows one to define an

inductive type so that no eliminators are generated and no pattern-matching is allowed outside of the module where this type is defined. Exposing a custom induction principle for such a private inductive type allows one to retain computational behaviour, while restricting the user to explicitly provided eliminators.

Although such an implementation has some advantages, like making more proofs compute, it relies on specific implementation details. In the current version of Coq, private inductive definitions are still an experimental extension. The authors of [BT16] had to use a custom rewrite tactic implemented in OCaml in order to fix an incorrect behaviour of the private definition under the standard Coq rewriting tactic.

Our development in Lean could be seen as more explicit and straightforward approach to the implementation of a two-level type theory, and the simplicity of the encoding of fibrancy constraints makes it potentially more portable to different systems, as long as they are equipped with a powerful enough type class resolution mechanism.

4.8.2 Experience with Agda

Our choice of Lean as the language for the formalisation of this paper has been a consequence of a failed attempt at embedding two-level type theory in the Agda proof assistant [Nor07].

Analogously to the development that has been eventually realised in Lean, our plan was to consider Agda's underlying theory, which includes UIP, as the strict fragment of our two-level type theory, and use *instance arguments*, which are Agda's implementation of type classes, to express fibrancy conditions on pretypes.

Unfortunately, due to the way instance and implicit arguments get resolved in Agda, we were not able to get automatic propagation of fibrancy conditions over type expressions involving families of types, such as Π or Σ types in our initial attempt in Agda.

The self-contained example by Paolo Capriotti shows that certain ways of defining fibrancy condition fail to resolve implicit arguments.

```
module tltt where


postulate

  is-fibrant : ∀ {i} → Set i → Set i
  instance Π-is-fibrant : ∀ {i}{j}{A : Set i}{B : A → Set j}
                   → {| fibA : is-fibrant A |}
                   → {| fibB : (a : A) → is-fibrant (B a) |}
                   → is-fibrant ((a : A) → B a)

module _ {i} {A : Set i} {| fibA : is-fibrant A |} where

  postulate

    _~_ : A → A → Set i
```

```
module test {i}{j}{A : Set i}{B : A → Set j}

  {| fibA : is-fibrant A |}

-- this will work, if we change (a : A) → is-fibrant (B a)
-- to {a : A} → is-fibrant (B a)
  {| fibB : (a : A) → is-fibrant (B a) |} where

  test : (f : (a : A) → B a) → f ~ f

  test = ?
```

As it became clear later, a small change in the definition of `fibB` in the test module from `fibB : (a : A) → is-fibrant (B a)` to `fibB : {a : A} → is-fibrant (B a)` will make Agda's resolution mechanism work. It was not clear from the documentation why the original definition fails to work[5]. Although there is a way to make the example above work, it is still not clear if it is possible to develop two-level type theory in Agda in the same way as we have done in Lean.

We therefore considered alternative approaches, such as postulating a universe of fibrant types and the corresponding type formers. Using a certain trick suggested by Thorsten Altenkirch, one can make sure that the fibrant type formers agree with the primitive ones. The trick is similar to the one that allows higher inductive types with judgemental reduction rules to be implemented in Agda [com12].

However, it appeared that such an approach, although probably feasible, is not as convenient and immediate as the solution based on type classes that we eventually settled with in Lean.

4.9 Conclusion

Two-level type theory is a promising approach to internalisation of results which are currently only partially internal to HoTT, and it is unclear if they can be fully internalised. We have demonstrated that two level type theory can be implemented in an existing proof assistant, outlining the general idea of the implementation. The approach to the implementation is suitable for most of the existing proof assistants based on dependent type theory, since we do not rely on implementation-specific details. Although, for our implementation to be convenient to use, one will require type classes, proof automation, or some way to add new judgmental equalities.

Our Lean development should still be considered a proof of concept, as it does not fully implement all the results presented in the paper [ACK17]. However, we hope that it serves as a compelling demonstration of the feasibility of our formalisation approach. To test how one can work in the fibrant fragment in our Lean development, we have ported some theorems from the Lean HoTT library. We used proof automation to mimic computational behavior of the β-rule for the fibrant equality eliminator. In most cases that we have considered,

[5] Thanks to Nils Danielsson for pointing out what needs to be changed in this example. Also, see `https://github.com/agda/agda/issues/2755`

modifications of proofs were not substantial. Although, it is worth pointing out that in some situations it is inconvenient to write statements where reduction in types involving the β-rule for fibrant equality is required.

As a possible extension of results presented in this work, one could consider to explore the conservativity result from [Cap16]. Having conservativity, one could take, for example, a definition of n-restricted semi-simplicial types in in two-level type theory. Instantiating the definition with particular *strict* natural number (i.e. some $n : \mathbb{N}^{\mathrm{s}}$) 0, 1, 3, etc., and evaluating the term in the strict fragment one could acquire a term, which belongs to the fibrant fragment of our two-level type theory. Since the fibrant fragment represents HoTT, it should be possible to use this term in the proof assistant, where HoTT is supported directly (after converting the term appropriately). In the context of our Lean development it would require only minimal efforts, since Lean (version 2) has a mode supporting HoTT "natively".

Chapter 5

Conclusion

The results of our work show how we have addressed the questions that we stated in the introduction to the thesis.

First, we have developed a payoff intermediate language along with a compilation procedure. The compilation procedure allows us to translate contract specifications in a domain-specific language for financial contracts (CL) to payoff expressions. The template extension to the original contract language and the parameterisation of payoff expressions with the "current time" parameter allow for compiling contact templates (or instruments) once and then reuse the compiled code, instantiating the parameters with different values. Our experience with Haskell code generation shows that payoff expressions are relatively easy to map to a subset of a functional language, and we expect that code generation into the Futhark language [HSE+17] could be implemented in a similar way. Moreover, performance measurements for payoff expressions compiled "by hand" to OpenCL show that for simple contracts the runtime overhead is very small in comparison with the recompilation time. Although, one could potentially find some limitations for more complicated contracts, properties of our implementation do not restrict one from using the contract reduction and recompilation approach. All the development has been carried out in the Coq proof assistant from the beginning and all proofs and definitions related to the payoff intermediate language (including the soundness proofs in Chapter 2) are high-level explanations of the Coq formalisation. We can say that we successfully addressed most of the questions related to this part of the thesis.

Second, we have partially formalised static interpretation of a higher-order module system for Futhark. Particularly, we have defined the core notions required for the formalisation, namely semantic objects, along with required relations. We have proved one part of the normalisation theorem, specifically, that static interpretation terminates with a target language expression. Another part of the theorem related to the typing of target language expressions has not been formalised (we leave this for future work).

The implementation of semantic objects turned out to be surprisingly tricky due to the conservative strict positivity check in Coq. We have developed a technique allowing for another representation of finite maps to be used in the definition of semantic objects. We have proved that this representation is isomorphic to the one from the standard library of Coq and used the implicit coercion mechanism to reuse operations on finite maps from the standard li-

brary. We have built most of the required machinery to deal with bindings in semantic objects using nominal techniques. For a simplified notion of semantic objects, we have developed a corresponding nominal set and have defined α-equivalence. In the full setting we have sketched the approach and have developed examples showing the feasibility of our approach. However these changes are not yet incorporated into the proof of Theorem 3.3.1.

Our experience shows that some restrictions on definitions in Coq do not allow one to use abstractions properly. For example, in the definition of semantic objects, it is impossible to use an abstract type of finite maps, otherwise Coq will not be able to check the definition for strict positivity. When using the `Fixpoint` construct one has to be very explicit about recursive calls and in most cases it is not possible to call another function, implementing for example a nested fixpoint. Instead, one has to inline the definition (see Remark 3.5.6).

We have developed most of the machinery required for the full formalisation of static interpretation normalisation, and we believe it is possible to finish the formalisation given enough time. Having the full result we can then approach the problem of extracting a certified implementation from our formalisation.

Third, we have implemented a formalisation of two-level type theory in the Lean proof assistant using only features available for the users and not by modifying Lean's code. We showed that in order to make such an encoding usable in an existing proof assistant, support for type classes, or support for a similar mechanism is required. Our example application to internalisation of inverse diagrams shows that our implementation is suitable for this purpose.

We believe that there are no conceptual difficulties in extending our Lean development with other results from [ACK17] because most of the reasoning happens in the strict fragment. However, such an extension requires a well-developed standard library to avoid formalising notions not directly related to the results. We would like to mention that working in the fibrant fragment is somewhat less convenient in comparison with the proof assistant with "native" support for homotopy type theory. This is mainly due to the lack of computational behavior of the fibrant equality eliminator. We have found ways to overcome this limitation, but it could still cause inconveniences for more complicated proofs in the fibrant fragment.

Bibliography

[ABB+16] Christian Andreetta, Vivien Bégot, Jost Berthold, Martin Elsman, Fritz Henglein, Troels Henriksen, Maj-Britt Nordfang, and Cosmin E. Oancea. FinPar: A parallel financial benchmark. *ACM Trans. Archit. Code Optim.*, 13(2):18:1–18:27, June 2016.

[ABW07] Brian Aydemir, Aaron Bohannon, and Stephanie Weirich. Nominal reasoning techniques in Coq. *Electron. Notes Theor. Comput. Sci.*, 174(5):69–77, June 2007.

[ACK16] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending Homotopy Type Theory with Strict Equality. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62, pages 21:1–21:17, Dagstuhl, Germany, 2016.

[ACK17] D. Annenkov, P. Capriotti, and N. Kraus. Two-Level Type Theory and Applications. *ArXiv e-prints*, May 2017.

[AEH+06] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer*, 8(6):485–516, 2006.

[AVR95] B.R.T Arnold, A. Van Deursen, and M. Res. An algebraic specification of a language for describing financial products. In *ICSE-17 Workshop on Formal Methods Application in Software Engineering*, pages 6–13, 1995.

[Bar13] Bruno Barras. Native implementation of Higher Inductive Types (HITs) in Coq, September 2013. Available at `http://www.crm.cat/en/Activities/Documents/barras-crm-2013.pdf`.

[BBE15] Patrick Bahr, Jost Berthold, and Martin Elsman. Certified symbolic management of financial multi-party contracts. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP'2015, pages 315–327, September 2015.

[BC10] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer-Verlag, 2010.

[Ber13] Yves Bertot. Private Inductive Types: Proposing a language extension, 2013. Available at `http://coq.inria.fr/files/coq5_submission_3.pdf`.

[BHKM12] Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in Coq. *J. Autom. Reasoning*, 49(2):141–159, 2012.

[Bre15] Matthew Brecknell. Pattern-matching dependent types in Coq. Available at `https://matthew.brecknell.net/post/pattern-matching-dependent-types-in-coq/`, 2015. Talk.

[BT16] Simon Boulier and Nicolas Tabareau. Formalization of model structures in Homotopy Type System (in Coq), 2016. `https://github.com/SimonBoulier/ModelStructure-HTS`.

[Cap16] Paolo Capriotti. *Models of Type Theory with Strict Equality*. PhD thesis, School of Computer Science, University of Nottingham, Nottingham, UK, 2016.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[Cha12] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3):363–408, Oct 2012.

[Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 143–156, New York, NY, USA, 2008. ACM.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[Cho15] Pritam Choudhury. Constructive representation of nominal sets in agda. MPhil Thesis, University of Cambridge, 2015.

[Clo13] Ranald Clouston. Generalised name abstraction for nominal sets. In Frank Pfenning, editor, *Foundations of Software Science and Computation Structures: 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 434–449. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[com12] The HoTT and UF community. Homotopy Type Theory, Since 2012. Agda library, available online at `https://github.com/HoTT/HoTT-Agda`.

[CP90] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88: International Conference on Computer Logic Tallinn, USSR, December 12–16, 1988 Proceedings*, pages 50–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.

[Cur34] H. B. Curry. Functionality in Combinatory Logic. In *Proceedings of the National Academy of Sciences of the United States of America*, volume 20, pages 584–590, November 1934.

[Dan12] Daniel Licata. Abstract Types with Isomorphic Types. `https://homotopytypetheory.org/2012/11/12/abstract-types-with-isomorphic-types/`, 2012. Blog post.

[dB72] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[Dij13] Gabe Dijkstra. Programming in homotopy type theory and erasing propositions. Master's thesis, Utrecht University, 2013.

[dMKA+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover. In *Automated Deduction - CADE-25, 25th International Conference on Automated Deduction*, 2015.

[Els99] Martin Elsman. Static interpretation of modules. In *Procedings of Fourth International Conference on Functional Programming*, ICFP '99, pages 208–219. ACM Press, September 1999.

[FSNB09] Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(1):27–45, 2009.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A Machine-Checked Proof of the Odd Order Theorem. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 163–179. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[GKN10] Adam Grabowski, Artur Kornilowicz, and Adam Naumowicz. Mizar in a Nutshell. *Journal of Formalized Reasoning*, 3(2):153–245, 2010.

[Gon08] Georges Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In Deepak Kapur, editor, *Computer Mathematics: 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, pages 333–333, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Gor94] Andrew D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications: 6th International Workshop, HUG '93 Vancouver, B. C.,*

Canada, August 11–13, 1993 Proceedings, pages 413–425. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[GP02] Murdoch J. Gabbay and Andrew M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13(3):341–363, Jul 2002.

[Hed98] Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *J. Funct. Program.*, 8(4):413–436, 1998.

[HEO14] Troels Henriksen, Martin Elsman, and Cosmin E Oancea. Size slicing: a hybrid approach to size inference in Futhark. In *Proceedings of the 3rd ACM SIGPLAN workshop on Functional high-performance computing*, pages 31–42. ACM, 2014.

[HKZ12] Tom Hvitved, Felix Klaedtke, and Eugen Zalinescu. A trace-based model for multiparty contracts. *The Journal of Logic and Algebraic Programming*, 81(2):72–98, 2012.

[How80] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*, pages 480–490. Academic Press, London-New York, 1980.

[HS96] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[HS00] Robert Harper and Christopher Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction*, pages 341–387. MIT Press, Cambridge, MA, USA, 2000.

[HSE+17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[KS15] Pepijn Kokke and Wouter Swierstra. Auto in Agda. In Ralf Hinze and Janis Voigtländer, editors, *Mathematics of Program Construction: 12th International Conference, MPC 2015, Königswinter, Germany, June 29–July 1, 2015. Proceedings*, pages 276–301. Springer International Publishing, Cham, 2015.

[Lam80] J. Lambek. From λ-calculus to Cartesian closed categories. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 375–402. Academic Press, London, 1980.

[LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. Towards a mechanized metatheory of Standard ML. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '07, pages 173–184, New York, NY, USA, 2007. ACM.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.

[Ler09] Xavier Leroy. Programming with dependent types: passing fad or useful tool? Available at `http://www.cs.ox.ac.uk/ralf.hinze/WG2.8/26/slides/xavier.pdf`, 2009. Talk.

[Let08] Pierre Letouzey. Extraction in Coq: An overview. In *Computability in Europe*, volume 5028 of *LNCS*, pages 359–369, 2008.

[Lex] LexiFi. Contract description language (MLFi). Available at `http://www.lexifi.com/technology/contract-description-language`.

[Lex08] LexiFi. Structuring, Pricing, and Processing Complex Financial Products with MLFi. `http://www.lexifi.com/files/resources/MLFiWhitePaper.pdf`, 2008. White paper.

[LS13] Daniel R. Licata and Michael Shulman. Calculating the fundamental group of the circle in homotopy type theory. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '13, pages 223–232, Washington, DC, USA, 2013. IEEE Computer Society.

[ML84] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

[MP93] James McKinna and Robert Pollack. Pure type systems formalized. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications: International Conference on Typed Lambda Calculi and Applications TLCA '93 March, 16–18, 1993, Utrech, The Netherlands Proceedings*, pages 289–305. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Available at `http://bitcoin.org/bitcoin.pdf`, 2008.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology and Göteborg University, Göteborg, Sweden, 2007.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[PdAC+16] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2016. Version 4.0. `http://www.cis.upenn.edu/~bcpierce/sf`.

[PES00] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering (functional pearl). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, ICFP'2000, September 2000.

[PI17] Grant Olney Passmore and Denis Ignatovich. Formal verification of financial algorithms. In Leonardo de Moura, editor, *Automated Deduction – CADE 26: 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6–11, 2017, Proceedings*, pages 26–41. Springer International Publishing, Cham, 2017.

[Pit06] A. M. Pitts. Alpha-structural recursion and induction. *Journal of the ACM*, 53:459–506, 2006.

[Pit13] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, New York, NY, USA, 2013.

[Pit16] A. M. Pitts. Nominal techniques. *ACM SIGLOG News*, 3(1):57–72, January 2016.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *Proceedings of the 16th International Conference on Automated Deduction: Automated Deduction*, CADE-16, pages 202–206, London, UK, UK, 1999. Springer-Verlag.

[RRD10] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. F-ing modules. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '10, pages 89–102, New York, NY, USA, 2010. ACM.

[Shu15] Michael Shulman. Univalence for Inverse Diagrams and Homotopy Canonicity. *Mathematical Structures in Computer Science*, pages 1–75, Jan 2015.

[Sim09] SimCorp A/S. XpressInstruments solutions. Company whitepaper. Available at `http://simcorp.com`, 2009.

[Swa17] A. Swan. Some Brouwerian Counterexamples Regarding Nominal Sets in Constructive Set Theory. *ArXiv e-prints*, February 2017.

[Tai67] William W. Tait. Intensional interpretations of functionals of finite type. *Journal of symbolic logic*, 32:198–212, 1967.

[TMK+16] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 60–73, New York, NY, USA, 2016. ACM.

[UBN07] Christian Urban, Stefan Berghofer, and Michael Norrish. Barendregt's variable convention in rule inductions. In Frank Pfenning, editor, *Automated Deduction – CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings*, pages 35–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[UK11] Christian Urban and Cezary Kaliszyk. General bindings and alpha-equivalence in Nominal Isabelle. In Gilles Barthe, editor, *Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings*, pages 480–500. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book/`, Institute for Advanced Study, 2013.

[UT05] Christian Urban and Christine Tasson. Nominal techniques in Isabelle/HOL. In Robert Nieuwenhuis, editor, *Automated Deduction – CADE-20: 20th International Conference on Automated Deduction, Tallinn, Estonia, July 22-27, 2005. Proceedings*, pages 38–53. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[VAG+] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath*: Univalent Mathematics. Available at `https://github.com/UniMath`.

[Voe13] Vladimir Voevodsky. A simple type system with two identity types. Available at `https://ncatlab.org/homotopytypetheory/files/HTS.pdf`, 2013. Unpublished note.

[Wad87] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM.

[Wad15] Philip Wadler. Propositions as types. *Commun. ACM*, 58(12):75–84, November 2015.

[Woo15] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2015. Homestead revision, Founder, Ethereum & Ethcore, `gavin@ethcore.io`.