Foundations and Applications of Modal Type Theories

Jason Z. S. Hu
Doctor of Philosophy

School of Computer Science
McGill University
Montreal, Quebec, Canada

November 29, 2024

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Doctor of Philosophy

Abstract

Over the past few decades, type theories as mathematical foundations have been extensively studied and are well understood. Many proof assistants implement type theories and have found important applications to provide critical security guarantees. In these applications, users often write meta-programs, programs that generate other programs, to implement proof search heuristics and improve their work efficiency. However, as opposed to the deep understanding of type theories, it remains unclear what foundation is suitable to support meta-programming in proof assistants. In this thesis, I investigate modal type theories, a specific approach to this problem. In modal type theories, modalities are a way to shallowly embed syntax into the systems, so users can write meta-programs that manipulate syntax through these modalities.

I explore two different styles of modal systems. In the first part, I investigate the Kripke-style systems, which faithfully model the familiar quasi-quoting style of meta-programming. I develop an explicit substitution calculus and scale it to dependent types, introducing Mint. I prove strong normalization of Mint, which implies its logical consistency, using an untyped domain model.

Nevertheless, the Kripke-style systems only support composition and execution of code, and they cannot easily support a general recursion principle on the structure of code. To support such a general recursion principle, I develop the layered style, where a system is divided into nested layers of sub-languages. The layered style scales quite naturally to dependent types, introducing DeLaM. DeLaM allows users to compose, execute and recurse on dependently typed code. I prove that DeLaM is weakly normalizing and its convertibility problem between types and terms is decidable. Hence, DeLaM provides a type-theoretic foundation to support type-safe meta-programming in proof assistants.

Abrégé

Au cours des dernières décennies, les théories des types comme fondements mathématiques ont été étudiées en détails et sont maintenant bien compises. Plusieurs assistants de preuve implémentent les théories des types et ont établi des applications pour fournir d'importantes garanties de sécurité. Dans ces applications, les utilisateur écrivent des méta-programmes, c'est-à-dire des programmes générant d'autres programmes, dans le but d'implémenter des heuristiques de recherche de preuve et ainsi d'améliorer l'efficacité de leur travail. Néanmoins, malgré la grande compréhension des théories des types, les fondements adéquats pour la méta-programmation dans les assistants de preuves demeurent incertains. Dans cette thèse, j'investigue les théories des types modaux, une approche spécifique tentant de résoudre ce problème. Dans les théories des types modaux, les modalités fournissent une encapsulation superficielle de la syntax dans le système, permettant aux utilisateurs d'écrire des méta-programmes qui manipulent la syntaxe à travers ces modalités.

J'explore deux styles distincts de systèmes modaux. D'abord, j'explore les systèmes de style Kripke, qui modélisent fidèlement l'approche familière de méta-programmation appelée quasi-citation (traduit de l'anglais *quasi-quotation*). Je définis un calcul de substitution explicite, puis l'étends aux types dépendents, menant à l'introduction de MINT. Je prouve la normalization forte de MINT, qui implique sa consistence logique, en utilisant un modèle de domaine non-typé.

Cependant, les systèmes de styles Kripke supportent seulement la composition et l'éxécution de code, et permettent difficilement le support d'un principe général de récursion sur la structure du code. Afin de supporter un principe général de récursion, je développe le style stratifié, dans lequel un système est séparé en strates imbriqués de sous-langages. Le style stratifié s'étend de façon naturelle aux types dépendents, menant à l'introduction de DeLaM. DeLaM permet la composition, l'éxécution, et la récursion sur du code à type dépendent. Je prouve la normalization faible de DeLaM et que le problème de convertibilité entre les types et les termes est décidable. Conséquemment, DeLaM offre un fondement dans la théorie des types pour la méta-programmation sécuritaire dans les assistants de preuve.

Contributions

In this thesis, each chapter in Parts I and II is a type theory that I have developed. I am the first and main author of corresponding publications and technical reports. Publications are coauthored with my supervisor Prof. Brigitte Pientka. Junyoung (Clare) Jang also coauthored (Hu et al., 2023). Contents in Chapters 2 and 4 are new; they have not been published.

- Chapter 2 re-examines the system $\lambda^{\Box}$ by Davies and Pfenning (2001). The normalization proof extends the one for simply typed λ calculus by Abel (2013). The mechanization is solely done by myself.

- Chapter 3 introduces a new dependent type theory, Mint. The normalization proof extends the one for Martin-Löf type theory by Abel (2013). The mechanization is about 80% done by myself. The rest is accomplished as a collaboration with Junyoung (Clare) Jang. This chapter is published work (Hu et al., 2023).

- Chapter 4 introduces layered modal type theory, which has certain overlaps with the system by Pfenning and Davies (2001). The weak normalization and the decidability of convertibility is based on Abel et al. (2018).

- Chapter 5 introduces a dependent type theory for intensional analysis, DeLaM. This system extends layered modal type theory from Chapter 4. The weak normalization and the decidability of convertibility is based on Abel et al. (2018). This chapter is published work (Hu and Pientka, 2025).

Publications

- Jason Z. S. Hu and Brigitte Pientka. 2025. A Dependent Type Theory for Meta-programming with Intensional Analysis. *Proc. ACM Program. Lang. 9*, POPL (2025).

- Jason Z. S. Hu and Brigitte Pientka. 2024. Layered Modal Type Theory: Where Meta-programming Meets Intensional Analysis. In *Proceedings of the 33rd European Symposium on Programming on Programming Languages and Systems, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Part I (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, 52–82.

- Jason Z. S. Hu, Junyoung Jang, and Brigitte Pientka. 2023. Normalization by Evaluation for Modal Dependent Type Theory. *J. Funct. Program.* 33 (2023).

- Jason Z. S. Hu and Brigitte Pientka. 2022. A Categorical Normalization Proof for the Modal Lambda-Calculus. In *Proceedings of the 38th Conference on the Mathemati- cal Foundations of Programming Semantics, MFPS XXXXVIII, Cornell University, Ithaca, NY, USA, with a satellite event at IRIF, Denis Diderot University, Paris, France, and online, July 11-13, 2022 (EPTICS, Vol. 1)*, Justin Hsu and Christine Tasson (Eds.). EpiSciences.

Acknowledgements

In the breeze of the sixth fall since my return to Montreal, I finally see the end of my PhD study at McGill University. It was a long, difficult, yet fun period of time, and I was definitely not able to go through it without support from many people. In the past a bit over five years, I owe the most to my supervisor, Prof. Brigitte Pientka. She not only guided me in my research with care, but also assisted me in developing a professional life. Her advice style was critical to our dynamics even during the difficult COVID time. I also enjoyed every part of freedom under her supervision, because of which I had a very good personal life outside of research.

During this long PhD study, I surely did not work alone. Clare Jang and I collaborated on many projects. Interactions and discussions with him were often helpful and inspiring. Members in the CompLogic group had helped me in many ways, including teaching and research. Some off-track chitchats were great ways to relax my nerves. They have my appreciations (not in any particular order): David Thibodeau, Jake Errington, Antoine Gaulin, Hanneli Tavante, Marc-Antoine Ouimet, Johanna Schwartzentruber, Max Kopinsky, Daniel Zackon, Ryan Kavanagh, Chuta Sano, etc..

The Agda programming language/proof assistant has constituted an important part of my PhD study. The Agda community has been very welcoming, and many discussions about type theories inspired me to do a PhD in dependent type theory. A non-exhaustive list of people includes Prof. Matthew Daggitt, G. Allais, Prof. Jacques Carette, Prof. Wolfram Kahl, among many others.

My progress committee members have also been very supportive: Prof. Prakash Panangaden, Prof. Clark Verbrugge and Prof. Stefan Monnier. I would also like to thank my internal examiner Prof. Marcin Sabok, my external examiner Prof. Dominic

Orchard, and my defense committee chair Prof. Mathieu Blanchette.

My journey to the grad school would not have started in the same way without my master's supervisor, Prof. Ondřej Lhoták, who offered great help for me to return to the academia after years in the industry. Prof. Arie Gurfinkel and Prof. Richard Trefler supported my PhD applications and wrote recommendation letters, which were very important to pave my way to a PhD study.

Besides an academic career, I also developed a professional life in the industry at Amazon Web Services (AWS) during the summers. Without Mike Whalen's recognition, it would not have been possible, so I feel very grateful to him. He is the best manager that I could hope for. Nathan Chong is also a fun and great manager. There are many other people whom I received help from and feel also grateful to: Daniel Schwartz-Narbonne, Mark Tuttle, Michael Tautschnig, Dominic Mulligan, Konrad Slind, Bruno Dutertre, Aaron Bradley, Soonho Kong and many others.

Outside of my study, I have developed a very delightful personal life. Among all activities, I cherish badminton the most. Badminton has critical importance in my PhD life, especially during difficult times. I would like to thank my coaches Tyler Patt and Zhifeng Chen for training me and improving my badminton skills. I would also like to thank the numerous badminton mates who I met at McGill's gym, CEPSUM and many other places. They are the ones who let me understand how great sports truly are.

All these beautiful experiences would not have been possible without my mom's support for me to come to Canada. Pierre Grondin and my mom offered the best they could to help me settle a life in Canada. My dad's and other family members' understanding is important for me to take a pause on the industry and start a long grad life. I hope they also feel that the time spent has paid off.

At last, I would like to thank my funding agencies. Without their stable financial supports, I would not have been able to invest time on my research to the full extent. My research was supported by the Natural Sciences and Engineering Research Council (NSERC) through Postgraduate Scholarships-Doctoral program (PGS D), Fonds de recherche du Québec-Nature et technologies (FRQNT) through Doctoral (B2X) Research Scholarship, and McGill University through Trottier Accelerator Award and the Charity Exercise Award.

Contents

List of Figures

List of Tables

1 Introduction

In the past few decades, computer programs have gained more and more significance in society. Today, computer software governs many important areas of everyone's life: finances, national defense systems, aerospace and astronautics industries, etc.. Failures and bugs in computer programs have or could have caused many catastrophes in the history. During the Cold War, both the US (Burr, 2020) and the Soviet (Washington Post, 2007) had false alarms for missiles coming from the other side. In 2012, the Heartbleed bug (Durumeric et al., 2014) was reported and was only patched two years later. It was a security vulnerability in the OpenSSL cryptographic library. The library is widely used in browsers and many secured communication channels. Heartbleed exploited a buffer over-read in the implementation and allowed arbitrary access to memory, including sensitive data. The Y2K problem (Committee on Government Reform and Oversight, 1998), or the millennium bug, was a bug in date representation and handling prior to the year of 2000 and caused a large public panic. This bug was predicted to have catastrophic consequences in the financial industry and the airline companies, but also had unexpected social effects due to the general public's misunderstandings. These examples illustrate that the proper functioning of the modern society critically depends on the correct behaviors of computer programs. But how do we know that a computer program, when being written, is going to execute correctly? One important and frequent solution is testing. We feed some inputs to the program and check whether the outputs match our expectations. However, most programs have a very large number of possible inputs, if not infinite, so testing does not generally provide full correctness guarantees. In practice, testing all corner cases of a program is usually a non-trivial task. The question now is: can we guarantee the correctness of a program?

One alternative to testing is formal software verification. We establish mathematical models for programs and prove some theorems to describe the behaviors of the programs. Unlike testing, this method provides strong mathematical guarantees; we know for sure that a program cannot go wrong as far as the model and the theorems describe.[1] These models are usually very sophisticated for practical software like compilers and operating systems, so it is not sufficient to simply sketch some proofs on a piece of paper and declare that the programs have been verified. To check proofs on a large scale, many researchers (Martin-Löf, 1975; Martin-Löf, 1984; Coquand and Huet, 1988; Pfenning and Paulin-Mohring, 1989; Luo, 1990, etc.) have designed mathematical foundations suitable for computer-based proof checking and implemented some of these foundations as proof assistants, the software that checks correctness of proofs. The foundation that I am particularly interested in this thesis, is type theories. Type theories were originally designed to formalize mathematics but are also successfully used in many software verification projects. These successful applications include CompCert, a certified optimizing C compiler (Leroy et al., 2016), and CertikOS, a preemptive, concurrent operating system kernel (Gu et al., 2011). Both projects are accomplished in the Coq proof assistant (soon to be renamed to Rocq) (The Coq Development Team, 2023). Coq is also widely used in mechanizing mathematics. The famous four-color problem was mechanized in Coq (Gonthier et al., 2008).[2] Nowadays, the Coq community has developed and maintains a large collection of formal mathematical libraries.

[1]If the model or the theorems are too weak, however, it is still possible for a proven correct program to break. This is called a "side channel". What a model and a theorem should be is non-trivial and collectively decided by the research community.

[2]The first proof of this problem was given by Appel and Haken (1977). The proof was controversial at the time because they used a computer program to exhaustively test 1936 different map configurations.

A few notable ones include C-Corn (Cruz-Filipe et al., 2004) for real analysis and algebra, Mathematical Components (Mahboubi and Tassi, 2022) for group theory, and many libraries for category theory (Wiegley, 2019; Huet and Saïbi, 2000; Timany and Jacobs, 2016; Gross et al., 2014; Ahrens et al., 2015). Another popular proof assistant in the mathematical community is Lean (de Moura et al., 2015; de Moura and Ullrich, 2021), which is more recently developed. The Lean community has actively mechanized a significant portion of mathematics (The Mathlib Community, 2020). Lean also has attracted world-renowned mathematicians to mechanize their cutting-edge development (Gowers et al., 2023) and has helped them to catch mistakes in their work (Tao, 2023). My choice of proof assistant for this thesis is Agda (The Agda Team, 2024; Norell, 2007). It is also used to mechanize some mathematics like category theory by various groups (Peebles et al., 2018; Hu and Carette, 2021). Agda is widely used in the programming language community for experiments and research.

A particular strength of type theories, as opposed to more traditional logic, is that type theories are *computational*. Definitions in type theories have computational behaviors and can be treated as programs. The dual reading of type-theoretic definitions as proofs and as programs is called the Curry-Howard Correspondence. Many people might find type theories quite familiar when encountered for the first time, because superficially types theories just look like functional programming languages. In a syntax similar to Haskell and Agda's, multiplication of natural numbers is defined as:

```
mult : Nat → Nat → Nat
mult zero     n = 0
mult (succ m) n = mult m n + n
```

For clarity, I abbreviate `succ ... (succ zero)` as numbers, e.g. `1` is notation for `succ zero`. Readers having some experience in functional programming might already find this definition intuitive. Multiplication takes two natural numbers and returns another natural number. It is defined by recursion on the first natural number. If the first natural number is `zero`, then the result is just `zero`. In the successor case, a recursion occurs and the sum of the recursive result and `n` is returned. In addition to writing functions as in a functional programming language, we can also write proofs in type theories. The computational behaviors often reduce the sizes of proofs, as computation is implicitly carried out during proof checking (or type checking) to relate two terms,[3] whereas in logic, the equality between two terms must be explicitly established. The following proves that `1` is the left identity of `mult`:

```
left-identity-mult : ∀ m → mult 1 m ≡ m
left-identity-mult m = refl -- mult 1 m = mult 0 m + m = 0 + m = m
```

Here `refl` is the reflexivity proof. The type theory *computes* `mult 1 m` to `m` so the proof of this property is just a one-liner.[4] In type theory, the programming language and the proof language are identical, so its learning curve is in fact quite gentle.

Though computation is already quite helpful in reducing the manual effort in a proving activity, for some complicated problems, it is nevertheless time-consuming, tedious and even counterproductive to write down every last detail in a proof. It would be even more convenient, if some part of the proof can be generated by some programs. The art of meta-programming does precisely that. A meta-program is a program that generates other programs. Meta-programming has existed for a long time. Scheme/Lisp (Clinger and Rees, 1991; Kohlbecker et al., 1986; Abelson and Sussman, 1996) includes one of the earliest meta-programming systems. It adopts a *quasi-quoting* style of meta-programming, which has a significant influence on many subsequent systems (Taha and Sheard, 1997, 2000; Culpepper et al., 2019; Sheard and Peyton Jones, 2002; Parreaux et al., 2017; Mainland, 2012, etc.). There are three key operations in a quasi-quoting system: quote, splice and run. We quote a program to obtain its code. We then compose pieces of code by splicing them together, and once we are finished, we run the result code as a program. The multiplication function above can be turned into a meta-program using quasi-quoting as follows:

```
meta-mult : Nat → Code (Nat → Nat)
meta-mult zero     = quote (λ n. 0)
meta-mult (succ m) = quote (λ n. splice (meta-mult m) n + n)
```

The `meta-mult` function takes a natural number and returns a piece of code for a function. The purpose of `meta-mult m` is to generate code that adds the argument of the returned

[3]As an example, computation in type theory is taken advantages of and heavily relied on in Coq's mechanization of the four-color theorem (Gonthier, 2023, Sec. 4).

[4]In Principia Mathematica, Whitehead and Russell (1927) spent over 400 pages to prove $1 + 1 = 2$. This is sometimes used as a counter-argument for formal mathematics. It is clear that our science has gone a long way in the past century.

function code m times. In the zero case, a quote operation constructs Code, which describes a constant function of zero. In the step case, the recursive call is invoked. The recursion meta-mult m returns Code of a function, which is spliced in the λ abstraction. This generated function is applied to n to give the code for the multiplication of m and n. Another addition of n gives the correct result. The code generated by meta-mult runs as if it was manually written:

```
run (meta-mult 2) 5 -- computes to 10
```

In practical programming languages, meta-programming has become a common way to improve productivity, gain modularity and optimize program performance.

In the settings of proof assistants, meta-programming plays an important role in the form of tactic systems and contributes to the successes of type-theory-based proof assistants. Tactics in a broad sense are algorithms that compute proofs. Under the Curry-Howard Correspondence, since proofs are just programs, tactics can also be seen as a special kind of meta-programs. Tactics as a way of proof automation are often used to capture common patterns of proofs to eliminate tedious proof steps, and encode proof heuristics based on domain-specific knowledge. They often further shrink the size of a proof script drastically. Moreover, constructing proofs using tactics helps to make proof scripts more robust. Proofs established by tactics are often resilient to changes to definitions or assumptions, so a moderate adjustment to upstream definitions might require little or even no change to downstream proof scripts at all. Tactics are also used to improve presentations of proof contexts and user experience in the proof assistants (Krebbers et al., 2017, 2018; Cao et al., 2018).

In practice, tactic systems in proof assistants are results of engineering and are usually implemented by instrumenting the type checking kernels. For example, Agda (van der Walt and Swierstra, 2012), Idris (Christiansen and Brady, 2016) and Lean (Ebner et al., 2017) support meta-programming via reflection, which is a mechanism to reflect programs into (usually untyped) abstract syntax trees (ASTs). With reflection, meta-programs are programs that manipulate and output ASTs. In Coq, Ltac (Delahaye, 2000) and Ltac2 (Pédrot, 2019) are frequent choices, which are implemented as a separate language on top of Coq's proof language. Other options in Coq include Mtac (Ziliani et al., 2013) and Mtac2 (Kaiser et al., 2018), which are implemented as an instrumentation of Coq's kernel. They allow users to write meta-programs directly in Coq. MetaCoq (Sozeau et al., 2020; Anand et al., 2018) is originally designed to formalize Coq inside of Coq itself, but it also provides an infrastructure to support meta-programming in Coq using some reflection mechanism. One primary problem with all these tools is that they cannot always guarantee the well-scopedness and well-formedness of generated programs and proofs. Tactics written by these tools could lead to errors, which further require additional error reporting and handling mechanisms. Debugging support is limited so finding out what goes wrong is a painful experience. These tools do not truly leverage the full power of the type theories implemented by the proof assistants. Ultimately, reflection and instrumentation are some external mechanisms and hence have no type-theoretic foundation.

On the theoretical end, more than 20 years ago, Davies and Pfenning discovered that the necessity modality (□) in modal logic S4 can be used to model simply typed meta-programming (Davies and Pfenning, 2001; Pfenning and Davies, 2001). They investigated two different styles of modal type theories, the Kripke style and the dual-context style, and showed that both styles model meta-programming in two different flavors. The Kripke style provides a foundation for the quasi-quoting style of meta-programming introduced above, where the `Code` type constructor is modeled by the □ modality. The dual-context style, on the contrary, models a less common *comonadic* style of meta-programming. In this style, we have access to a meta-language to manipulate code through the □ modality. The multiplication example can be rewritten in the dual-context style as:

```
meta-mult2 : Nat → Code (Nat → Nat)
meta-mult2 zero     = quote (λ n. 0)
meta-mult2 (succ m) = let quote u ← meta-mult2 m
                      in quote (λ n. u n + n)
```

The main difference is in the step case. In this case, the recursive call is invoked outside of `quote`. The result of the call is extracted by `let quote` and is bound to a *meta-variable* `u`. This meta-variable represents a hole in the code, and is used in `quote` where `u n` describes the code for multiplying `m` and `n`. Later in the thesis, I will describe how both styles are substantially different. Davies and Pfenning's work serves as the first step towards type-safe meta-programming in different settings for many subsequent works (Nanevski et al., 2008; Jang et al., 2022; Pientka et al., 2019), including my own work to be discussed in the rest of this thesis.

In this thesis, *I add support for meta-programming to dependent type theory.* This setting is not entirely covered by previous investigations and is particularly interesting to those who want to leverage the full expressive power provided by the dependent type theory implemented by a proof assistant when writing tactics. One particular feature that I would like to achieve is intensional analysis. Intensional analysis allows meta-programs to analyze the syntactic structure of code, so it is especially useful for tactics in proof assistants. However, the correspondence with modal logic S4 observed by Davies and Pfenning does not reveal how intensional analysis can be supported in a type theory. How to enable intensional analysis without resorting to some external mechanism in a dependent type theory while retaining the consistency of the overall system poses a major technical challenge. Metaphorically speaking, intensional analysis using instrumentation or alike is like a surgery on code. What I aim at in this thesis is to ask a surgeon to perform surgery on him- or herself.[5] The results presented in this thesis provide an alternative to the engineering work above, where we are able to make full use of dependent types provided by the proof assistants, and demonstrate how one could support meta-programming within the system itself (or other extensions to a type theory) via systematic thoughts.

This thesis largely extends work by Davies and Pfenning (2001); Pfenning and Davies (2001). In Part I, I first investigate the Kripke-style systems. One main result of Part I is MINT, a Kripke-style dependent type theory, which can be used as a program logic for, e.g., MetaML (Taha and Sheard, 1997, 2000). Nevertheless, one limitation of Kripke-style systems is that it is unclear how the Kripke style and intensional analysis are compatible. Intensional analysis is particularly important in proof assistants, because we often need to analyze the structure of a goal in order to give an algorithm to construct a proof. To support intensional analysis, or more specifically recursion on the structure of code, in Part II, I introduce the layered style, based on the dual-context style. It turns out that the comonadic style of meta-programming is the right step towards enabling intensional analysis. I introduce *the matryoshka principle*, which is a design guideline

[5]Medically, it is called a "self-surgery", which does not occur too often. The first successful self-surgery is believed to be carried out by Dr. Evan O'Neill Kane, who performed an appendectomy on himself in 1921.

to simultaneously enable code composition, code running and code recursion in type theories. At the end of Part II, I define DeLaM, a dependent type theory that builds on the matryoshka principle and supports code running and recursions on the code of types and terms coherently.

In the rest of this introduction,

- I will first give a brief introduction to type theory using simply typed λ-calculus. In this part, I will discuss the recipe for designing a type theory and the two main theorems which I care about in this thesis: normalization and the decidability of convertibility.
- Then I will describe the methodology I used during the investigations.
- At last I will list the contributions I make in this thesis and outline the structure for the rest of this thesis.

1.1 Simply Typed λ-calculus as Type Theory

In this section, I would like to briefly discuss a common framework of type theories, which I follow throughout this thesis. In particular, I focus on two important properties, normalization and the decidability of convertibility, which I will use to justify my own work in later chapters. The simply typed λ-calculus (STLC) with natural numbers is my running example.

Let us begin with the syntax of types and terms of STLC.

$$S, T := \texttt{Nat} \mid S \longrightarrow T \qquad \text{(Types)}$$

$$x, y \qquad \text{(Variables)}$$

$$s, t := x \mid \mathsf{zero} \mid \mathsf{succ}\ t \mid \mathsf{rec}_T\ s\ (x, y.s')\ t \mid \lambda x.t \mid t\ s \qquad \text{(Terms)}$$

$$\Gamma, \Delta := \cdot \mid \Gamma, x : T \qquad \text{(Typing Contexts)}$$

Types are ranged over by S and T, which include $\texttt{Nat}$, the type for natural numbers, and a function space. Terms are ranged over by s and t. Variables are ranged over by x and y. A natural number can be constructed either by zero or $\mathsf{succ}\ t$, which represent zero and the successor, respectively. To eliminate a natural number, $\mathsf{rec}_T\ s\ (x, y.s')\ t$

performs a recursion on a natural number t. Here s is the base case and s' is the step case. For the step case, x stands for the predecessor and y is the result of the recursive call on the predecessor. I call the type which a recursion returns a *motive*, which in this case is T. The term being recursed on is called a *scrutinee*, which is t. Finally, $\lambda x.t$ introduces a function and $t\ s$ applies a function t to an argument s. The typing context Γ is used in a typing judgment to describe the ambient variables of a term.

To describe a type theory, I follow a recipe which includes five kinds of rules:

- formation rules, which define how a type is well-formed,
- variable rules, which define when and how to use a variable,
- introduction rules, which define how a term of a given type is constructed,
- elimination rules, which define how a term of a given type is used, and
- equivalence rules, which define how two terms of the same type are related.

Let us consider what are the rules in STLC for each kind.

Formation rules It turns out that STLC needs no formation rule. That is, all syntactically valid types are well-formed.

$$\frac{}{\texttt{Nat wf}} \qquad\qquad \frac{S\ \texttt{wf} \qquad T\ \texttt{wf}}{S \longrightarrow T\ \texttt{wf}}$$

This however is not always the case in other systems. For example, with dependent types, a type could contain as complex computation as a term and thus formation rules are necessary. We will see in later chapters that my work includes non-trivial formation rules.

Variable rules Variables rules govern when and how variables are used. In STLC, there is only one variable rule:

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

The rule simply allows us to refer to a variable at any time as long as it is bound in the context. However, in later chapters, variable rules become less trivial.

In this thesis, I uniformly assume variables represented by de Bruijn indices (de Bruijn, 1972), so that the α-renaming problem is trivial, though for presentation purposes, I still use x, y to refer to variables.

Introduction rules The introduction rules are typing rules that describe how to construct a term of a given type. In STLC, there are the following introduction rules for `Nat` and functions:

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathtt{Nat}} \qquad \frac{\Gamma \vdash t : \mathtt{Nat}}{\Gamma \vdash \mathsf{succ}\ t : \mathtt{Nat}} \qquad \frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : S \longrightarrow T}$$

The first two rules are the introduction rules for natural numbers. The first rule says that zero always has type `Nat` and the second rule says that in order for succ t to have type `Nat`, t must also have type `Nat`. The third rule introduces a function through a λ. The body t must have type T within a context extended with $x : S$.

Elimination rules The elimination rules describe ways to make use of a type. In STLC, we have

$$\frac{\Gamma \vdash s : T \qquad \Gamma, x : \mathtt{Nat}, y : T \vdash s' : T \qquad \Gamma \vdash t : \mathtt{Nat}}{\Gamma \vdash \mathsf{rec}_T\ s\ (x, y.s')\ t : T} \qquad \frac{\Gamma \vdash t : S \longrightarrow T \qquad \Gamma \vdash s : S}{\Gamma \vdash t\ s : T}$$

The first rule instructs that we can make use of a natural number by recursing on it. To recurse on the scrutinee t, we must specify two branches, each handling the case when t computes to zero or a successor, respectively. In the step case s', x is replaced by the predecessor and y is replaced by the recursive call. The second rule makes use of a function by applying it to an argument.

Equivalence rules So far, the rules given above are relatively simple. There is precisely one rule for each term. Since a type theory computes, these terms should have non-trivial interactions. The computational behaviors are described by the equivalence rules. The equivalence rules are further split into three kinds: the PER[6] rules, the

[6]partial equivalence relation

congruence rules and the computation rules. The PER rules are always fixed. They state that the equivalence judgment $\Gamma \vdash t \approx t' : T$ is symmetric and transitive. The congruence rules are directly derived from the typing rules described above. For example, the congruence rule for λ is derived by replacing typing with equivalence in its typing rule:

$$\frac{\Gamma, x : S \vdash t : T}{\Gamma \vdash \lambda x.t : S \longrightarrow T} \qquad \Longrightarrow \qquad \frac{\Gamma, x : S \vdash t \approx t' : T}{\Gamma \vdash \lambda x.t \approx \lambda x.t' : S \longrightarrow T}$$

This congruence rule allows computation to continue in the body of a λ expression. The same principle applies for all other terms. In this thesis, the congruence rules are *necessarily* derived in this way, which I consider a conceptually clean way to approach type theories. The congruence rules allow us to trigger a computation in an arbitrary place, something that we often want in mathematics.

The computation rules are the final and the most complex ingredient to a type theory. They govern how actually interesting computation occurs. There are a number of β rules for each type, which describe how an elimination form interacts with an introduction form. Sometimes, there are also η rules, which describe how a type can be re-introduced by a vacuous elimination. For functions, the computation rules are

$$\frac{\Gamma, x : S \vdash t : T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x.t)\ s \approx t[s/x] : T}\ \beta \qquad \frac{\Gamma \vdash t : S \longrightarrow T}{\Gamma \vdash t \approx \lambda x.t\ x : S \longrightarrow T}\ \eta$$

The first rule is the β rule. It states that when a λ expression is applied to an argument, we replace s with x everywhere in t. This action is denoted by $[s/x]$, which is a substitution of s for x. The second rule is the η rule. It says that if t has a function type, then we can expand it to a λ expression.

η rules do not always exist. The following β rules are the only available computation rules for $\texttt{Nat}$:

$$\frac{\Gamma \vdash s : T \qquad \Gamma, x : \texttt{Nat}, y : T \vdash s' : T}{\Gamma \vdash \mathsf{rec}_T\ s\ (x, y.s')\ \mathsf{zero} \approx s : T}$$

$$\frac{\Gamma \vdash s : T \qquad \Gamma, x : \texttt{Nat}, y : T \vdash s' : T \qquad \Gamma \vdash t : \texttt{Nat}}{\Gamma \vdash \mathsf{rec}_T\ s\ (x, y.s')\ (\mathsf{succ}\ t) \approx s'[t/x, \mathsf{rec}_T\ s\ (x, y.s')\ t/y] : T}$$

When the recursor hits zero, it simply computes the base case. Otherwise, it computes the step case, with t for x and the recursive call for y. I follow the typical practice in type theory and do not include η rules for inductively defined types, for example, natural numbers.

Compared to a programming language A type theory can also be treated as a programming language due to the computational behaviors. A type theory and a programming language often share typing rules. In a programming language, instead of an equivalence relation, we often fix a reduction strategy and examine that the reduction strategy is compatible with the typing rules. Type safety is often characterized by syntactic properties like progress and preservation (Wright and Felleisen, 1994). In a type theory, besides the same properties as programming languages, we need stronger properties to justify the logical consistency of a type theory due to the dual reading of a type theory as a logic. In this thesis, I focus on two properties, normalization and the decidability of convertibility.

Normalization and Convertibility Normalization states that all well-typed terms in a type theory are equivalent to a subset of terms called normal forms. Normalization immediately implies the termination of all well-typed programs in a type theory. Therefore, a consistent type theory is Turing-incomplete. This, unfortunately, is a tradeoff for the logical consistency of the overall system. In this thesis, all normalization properties accompany normalization algorithms. I will give an explicitly procedure to normalize terms for each type theory.

The other property that I care about is the decidability of convertibility. The conversion checking checks if two terms of the same type are equivalent. Its decidability ensures that this checking procedure always terminates. This procedure is a critical component in a type-checker. As part of the conversion checking, normalization is a necessary step.

A normalization algorithm must have the soundness theorem:

Theorem 1.1 (Soundness)**.** *If* $\Gamma \vdash t : T$*, then* $\Gamma \vdash t \approx \mathsf{normalize}(t) : T$*.*

The theorem says that all well-typed terms are equivalent to their normal forms. This theorem makes sure the correctness of the normalization algorithm. The completeness theorem is optional; it might not hold for some normalization algorithms:

Theorem 1.2 (Completeness)**.** *If* $\Gamma \vdash t \approx t' : T$*, then* $\mathsf{normalize}(t) = \mathsf{normalize}(t')$*.*

The theorem says that if two terms are equivalent, then their normal forms are syntactically equal. If a normalization algorithm has both theorems, then the normalization algorithm is a *strong* one, because the completeness theorem takes into account all possible congruences and still ensures equal normal forms. Otherwise, it is a *weak* one. Having a strong normalization algorithm is convenient, because both soundness and completeness theorems together state that syntactically equivalent terms must have syntactically equal normal forms. Therefore, testing convertibility becomes trivially testing the syntactic equality of normal forms. Thus, when I give strong normalization proofs in Chapters 2 and 3, I elide the discussion about convertibility.

Without completeness, on the other hand, convertibility needs a dedicated, non-trivial algorithm. This method is taken by many (Abel et al., 2018; Pientka et al., 2019; Pujet and Tabareau, 2022, 2023, etc.), including myself in Chapters 4 and 5. Though this method leads to longer proofs, one benefit is that in an implementation, the testing often fails fast if two terms are actually not equivalent.

1.2 Methodology

In this thesis, I investigate two different styles of modal type theory (the Kripke and the layered styles in Parts I and II, resp.). In each part, I always first consider the simply typed version and observe some structures or properties. Then I scale the whole system up to dependent types using the observations from simple types as a guidance.

I find this method particularly enlightening. Though the ultimate goal is to design dependent type theories, the technicality induced by a dependent type theory is often

System	$\lambda^\Box$	MINT	Layered modal type theory	DELAM
Chapter	2	3	4	5
Dependent types	✗	✓	✗	✓
Style	Kripke	Kripke	Layered	Layered
Code composition, running	✓	✓	✓	✓
Intensional analysis	✗	✗	pattern matching	recursion
Normalization	strong	strong	weak	weak

Table 1.1: Comparison of type theories in this thesis

too overwhelming to make concise and crucial observations. In both parts, I always make observations from simple types that scale up to dependent types with little or no change. When handling dependent types, I only need to worry about technicalities incrementally.

The design of type theories in this thesis follows the recipe that I outlined in the previous section.

1.3 Contributions

The contributions made in this thesis are:

- In Chapter 2, I revisit Kripke-style $\lambda^\Box$ introduced by Davies and Pfenning (2001) and prove its *strong normalization* using an untyped domain model, following Abel (2013). I define *K-substitutions* (Kripke-style substitutions), which form a substitution calculus for Kripke-style $\lambda^\Box$, unifying substitutions and modal transformations given separately by Davies and Pfenning (2001). I reformulate $\lambda^\Box$ into an *explicit substitution calculus*. I then use *truncoids* to capture common algebraic structures appearing in both syntax and semantics. This chapter is fully mechanized in Agda.

- In Chapter 3, I scale Kripke-style $\lambda^\Box$ and its *strong normalization* in Chapter 2 to full Martin-Löf type theory, obtaining MINT. MINT models staged computation with dependent types, giving a program logic for similar systems to MetaML (Taha and Sheard, 1997, 2000). I verified that new concepts like K-substitutions and truncoids introduced in Chapter 2 scale nicely to dependent types. The normalization property through an untyped domain model also scales up nicely to dependent types. This chapter is also fully mechanized in Agda, with the help of Junyoung (Clare) Jang. The full mechanization of Part I is available online.[7]

- In Chapter 4, I introduce *the matryoshka principle*, which leads to *layered modal type theory*. This principle adds a layering index to the typing judgment of the (contextual) dual-context-style $\lambda^{\square}$ by Davies and Pfenning (2001); Nanevski et al. (2008). Layering leads to two important properties, static code and lifting, which enable *pattern matching on code* and *code running*, respectively, without jeopardizing normalization. I then give a *weak normalization* algorithm based on reducibility and a *decidable conversion checking* algorithm.

- In Chapter 5, I scale the simply typed layered modal type theory to full Martin-Löf type theory, introducing DeLaM. I continue to use the static code and the lifting properties to guide the design in DeLaM. In DeLaM, we are able to not only compose and run code of Martin-Löf type theory, but also perform *recursions on the code* of types and terms. I establish a layered semantic model by scaling the one from Chapter 4. From this model, I prove the *weak normalization* of DeLaM and its *decidability of convertibility*.

A quick summary can be found in Table 1.1.

Though many of my publications use category theory, I decide to not involve any category theory in this thesis. I hope that this decision will make this thesis more friendly to readers without a strong categorical background. In principle, this thesis should be self-contained, only requiring standard knowledge in programming language theory, i.e. understandings of inference rules, computational reductions and some discrete mathematics. Some experience in type-theory-based proof assistants definitely helps, though I do not rely on that. For this reason, most contents in this thesis are new. Chapter 3 has been fully published (Hu et al., 2023), though the narrative in Chapter 2 overlaps with (Hu et al., 2023). The proof in Chapter 2 is new and different

[7]`https://hustmphrrr.github.io/mech-type-theories/`

from the one in (Hu and Pientka, 2022a). In Chapter 4, I present a slightly more complex version than the one in (Hu and Pientka, 2024b), and a weak normalization proof, for the continuity in Chapter 5. Chapter 5 is also published (Hu and Pientka, 2025).

Technical reports are also available. For Part I, the technical report is given by Hu and Pientka (2022b). For Part II, the technical reports are given by Hu and Pientka (2023, 2024a). This thesis is intentionally kept high-level and only presents proof structures. Proof details can be found in the Agda mechanization and the technical reports.

1.4 Conventions

To develop meta-theories for the type theories, in this thesis I assume an informal *meta-type theory* akin to safe Agda with *induction-recursion* (Dybjer, 2000; Dybjer and Setzer, 2001, 2003) and the axiom of *functional extensionality.* This is the standard meta-type theory also assumed in many previous works (Abel, 2013; Abel et al., 2018; Pientka et al., 2019, etc.).

I use the following notational conventions:

- The scripted letters $\mathcal{C}$, $\mathcal{D}$, etc. denote derivations in the meta-type theory.
- A colon (:) is used to denote a typing relation in a type theory. For example, $t : T$ means that the term t has type T.
- Two colons (::) are used to denote a typing relation in the meta-type theory. I use this notation in the semantics for dependent type theories. For example, $\mathcal{D} :: \Gamma \vdash t : T$ means that $\mathcal{D}$ is the name for the typing derivation $\Gamma \vdash t : T$.
- The equality sign (=) denotes mathematical equality as usual. When used between two terms, types, etc., it means that two terms, types, resp. are syntactically equal.
- The assignment sign (:=) means definitions. The definition $f := g$ defines f as g.

Part I

Kripke-style Modal Type Theories

2

Kripke-style Modal λ-Calculus

In this chapter, I revisit Kripke-style $\lambda^{\Box}$ introduced by Pfenning and Wong (1995); Davies and Pfenning (1996, 2001) and prove its strong normalization. A Kripke-style modal system involves a stack of contexts. Contexts in the stack are pushed and popped as we interact with the $\Box$ modality. In Pfenning and Davies' original presentation, there are two separate operations on terms: ordinary substitutions, which replaces variables with terms, and modal transformations, which maps contexts between two stacks. The distinction between two operations imposes separate characterizations of syntactic properties and adds difficulties to giving a uniform substitution calculus and scaling to dependent types.

In this chapter, I introduce K-substitutions (Kripke-style substitutions), which unifies ordinary substitutions and modal transformations. A unified substitution calculus further leads to a normalization algorithm based on an untyped domain model. This normalization algorithm is both sound and complete. The content in this chapter has been fully mechanized in Agda.[8]

[8] `https://hustmphrrr.github.io/mech-type-theories/Unbox.README.html`

The normalization proof in this chapter is completely new. I have published a different strong normalization proof using a presheaf model (Hu and Pientka, 2022a), which is also based on K-substitutions.

2.1 Syntax of $\lambda^{\square}$

As indicated by its name, Kripke-style $\lambda^{\square}$ is inspired by Kripke semantics (Kripke, 1963) (see (Davies and Pfenning, 1996, Section 3) and (Davies and Pfenning, 2001, Section 4)). In $\lambda^{\square}$, instead of one context as in STLC, a stack of contexts is used in the typing judgment. It is convenient to view the stack as a travel path in a Kripke universe along its accessibility relation. Each context in the stack represents a Kripke world and assumptions in it in the universe. Hence, in this part, I often use "context" and "world" interchangeably. The context stack is always non-empty. Initially, we begin the stack with an empty context.

$$\epsilon; \Gamma_1; \ldots; \Gamma_n \vdash t : A \qquad \text{or} \qquad \vec{\Gamma} \vdash t : A$$

The rightmost context in the stack is the top and represents the current world. In the stack, worlds from left to right go from the past to the future. The $\square$ modality is used to represent facts or truths that are available in future worlds. For instance, if t has type $\square A$, then we can say that A is true in the future, or equivalently $\square A$ is currently true. The introduction form of $\square$ sends us to the next new world (a new empty context pushed into the stack). On the other hand, the elimination form of $\texttt{unbox}_n\ t$ travels n worlds back in the stack and use t of type $\square A$ to establish the truth of A in the current world. Which world A is true in is determined by the possible values of n. Here, n is called a *modal offset*. Controlling the possible values of modal offsets corresponds to controlling reflexivity and transitivity of the accessibility relation among worlds, or equivalently the modal laws in the Kripke semantics.

$$\frac{\vec{\Gamma}; \cdot \vdash t : A}{\vec{\Gamma} \vdash \texttt{box}\ t : \square A} \qquad \frac{\vec{\Gamma} \vdash t : \square A}{\vec{\Gamma}; \Delta_1; \ldots; \Delta_n \vdash \texttt{unbox}_n\ t : A}$$

The choice of the modal offset n corresponds to reflexivity and transitivity of the

Kripke Structure	Law \ System	K	T	$K4$	$S4$
	K: $\square(A \longrightarrow B) \longrightarrow \square A \longrightarrow \square B$	✓	✓	✓	✓
Reflexivity	T: $\square A \longrightarrow A$		✓		✓
Transitivity	4: $\square A \longrightarrow \square\square A$			✓	✓
modal offset n		$\{ 1 \}$	$\{ 0, 1 \}$	$\mathbb{N}^+$	$\mathbb{N}$

Table 2.1: Modal type theories, K, T, $K4$, $S4$

accessibility relation between worlds in the Kripke semantics. The relation between modal offsets and possible modal laws are summarized in Table 2.1. The table shows that $\lambda^{\square}$ is a *parameterized* system. This parametricity is crucial to give one modular normalization proof for all four systems at the same time. By setting possible values of modal offsets to different sets, we obtain different systems. Among the possible laws, the law K necessarily holds, which requires 1 to always be a possible value of modal offsets. In terms of meta-programming, this law permits code composition.

$$K : \square(A \longrightarrow B) \longrightarrow \square A \longrightarrow \square B$$
$$K := \lambda f\ x.(\texttt{unbox}_1\ f)\ (\texttt{unbox}_1\ x)$$

The law T extracts A from $\square A$. In Kripke semantics, it corresponds to reflexivity. It means that the next world includes the current world, so if A is true in the future, then it is also currently true. For A to be true, a modal offset must be able to take 0. In meta-programming, this law corresponds to the capability of code running.

$$T : \square A \longrightarrow A$$
$$T := \lambda x.\texttt{unbox}_0\ x$$

The law 4 duplicates $\square$. It states that truths in one step in the future also appear two steps in the future. Iteratively applying this law concludes that a truth in one step in the future is a truth in all possible futures. In Kripke semantics, it corresponds to transitivity. To model transitivity, modal offsets must be able to take all values ≥ 2.

In meta-programming, this laws allows to write meta-programs for meta-programs.

$$A4 : \Box A \longrightarrow \Box\Box A$$
$$A4 := \lambda x.\texttt{box}\ (\texttt{box}\ (\texttt{unbox}_2\ x))$$

Worlds in which variables reside are cleanly separated by $\Box$ and `unbox` with the right modal offset is necessary to access variables in previous worlds. For example, neither $\texttt{unbox}_1$ nor $\texttt{unbox}_3$ is a valid change in $A4$ and the resulting term is not well-typed because x is not visible.

The four sub-systems of $S4$ are results of choosing the admissibility of the laws T and 4. Among the four systems, $S4$ admits all three laws. Under Curry-Howard Correspondence, $S4$ corresponds to meta-programming in the quasi-quoting style. The laws K and 4 splice code from the same and some previous worlds, respectively. The law T runs the code. Thus, we can use $\lambda^{\Box}$ to model meta-programming when modal offsets take all natural numbers. The multiplication example in Chapter 1 is encoded in $\lambda^{\Box}$ as follows:

```
meta-mult : Nat → □ (Nat → Nat)
meta-mult zero     = box (λ n. zero)
meta-mult (succ m) = box (λ n. unbox₁ (meta-mult m) n + n)
```

The generated code is run by $\texttt{unbox}_0$:

```
unbox₀ (meta-mult 2) 5 -- computes to 10
```

However, looking at the laws, it is not quite clear how it is possible to support intensional analysis in this system. I indeed consider this as a limitation of the Kripke-style systems and this problem is addressed in Part II by layered modal type theories. Nevertheless, $\lambda^{\Box}$ is still a good model for some practical meta-programming systems in the quasi-quoting style, e.g. MetaML (Taha and Sheard, 1997, 2000), where intensional analysis is intentionally avoided.

The full syntax of $\lambda^{\square}$ is given below:

$$
\begin{array}{lll}
A, B & := \ \texttt{Base} \mid \square A \mid A \longrightarrow B & \text{(Types, } \mathsf{Typ}\text{)} \\
k, l, m, n & & \text{(Modal Offsets, } \mathbb{N}\text{)} \\
x, y & & \text{(Variables, } \mathsf{Var}\text{)} \\
s, t, u & := x \mid \texttt{box}\ t \mid \texttt{unbox}_n\ t \mid \lambda x.t \mid s\ t & \text{(Terms, } \mathsf{Exp}\text{)} \\
\Gamma, \Delta & := \cdot \mid \Gamma, x : A & \text{(Contexts, } \mathsf{Ctx}\text{)} \\
\vec{\Gamma}, \vec{\Delta} & := \ \epsilon \mid \vec{\Gamma}; \Gamma & \text{(Context Stack, } \overrightarrow{\mathsf{Ctx}}\text{)} \\
w & := v \mid \texttt{box}\ w \mid \lambda x.w & \text{(Normal Form, } \mathsf{Nf}\text{)} \\
v & := x \mid v\ w \mid \texttt{unbox}_n\ v & \text{(Neutral Form, } \mathsf{Ne}\text{)}
\end{array}
$$

In this minimal system, I only consider some base type `Base`, functions and the modal type $\square A$. The terms are standard. There are variables, `box` and `unbox` to introduce and eliminate a modal type $\square A$, and λ and function applications for functions. The normal and neutral forms are also standard. Neutral forms include variables and elimination forms. Normal forms include neutral forms and introduction forms. Recall that I use de Bruijn indices for variables in this thesis, so that the discussion also aligns with the mechanization closely. Interested readers could find an immediate correspondence between this chapter and the Agda code.[9]

As discussed previously, the typing judgment in $\lambda^{\square}$ uses a (non-empty) context stack, which contains a number of contexts. Each context simply binds a variable to a type. I use $|\vec{\Gamma}|$ to count the number of contexts in the stack.

The typing and equivalence judgments are described in Fig. 2.1. There is no formation rule; all syntactically well-formed types are valid. By the variable rule, only variables in the current world (the topmost context) are visible. The introduction and elimination rules for $\square A$ have been discussed. The rules for functions are standard. λ abstractions only push variables to the current world.

The equivalence rules are given following the recipe given in Chapter 1. The PER rules are fixed and the congruence rules are derived from the typing rules, so I omit them here. The only interesting rules are the computation rules. The β and η rules for functions follow immediately from STLC. For $\square A$, the η rule says that a term t of type

[9] `https://hustmphrrr.github.io/mech-type-theories/Unbox.README.html`

$\boxed{\vec{\Gamma} \vdash t : A}$ Term t has type A in context stack $\vec{\Gamma}$

$$\frac{x : A \in \Gamma}{\vec{\Gamma}; \Gamma \vdash x : A} \qquad \frac{\vec{\Gamma}; \cdot \vdash t : A}{\vec{\Gamma} \vdash \texttt{box}\ t : \Box A} \qquad \frac{\vec{\Gamma} \vdash t : \Box A \qquad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \texttt{unbox}_n\ t : A}$$

$$\frac{\vec{\Gamma}; \Gamma, x : A \vdash t : B}{\vec{\Gamma}; \Gamma \vdash \lambda x.t : A \longrightarrow B} \qquad \frac{\vec{\Gamma} \vdash t : A \longrightarrow B \qquad \vec{\Gamma} \vdash s : A}{\vec{\Gamma} \vdash t\ s : B}$$

$\boxed{\vec{\Gamma} \vdash t \approx t' : A}$ Terms t and t' have type A and are equivalent in context stack $\vec{\Gamma}$

β equivalence:

$$\frac{\vec{\Gamma}; (\Gamma, x : A) \vdash t : B \qquad \vec{\Gamma}; \Gamma \vdash s : A}{\vec{\Gamma}; \Gamma \vdash (\lambda x.t)\ s \approx t[s/x] : B} \qquad \frac{\vec{\Gamma}; \cdot \vdash t : A \qquad |\vec{\Delta}| = n}{\vec{\Gamma}; \vec{\Delta} \vdash \texttt{unbox}_n\ (\texttt{box}\ t) \approx t\{n/0\} : A}$$

η equivalence:

$$\frac{\vec{\Gamma} \vdash t : A \longrightarrow B}{\vec{\Gamma} \vdash t \approx \lambda x.(t\ x) : A \longrightarrow B} \qquad \frac{\vec{\Gamma} \vdash t : \Box A}{\vec{\Gamma} \vdash t \approx \texttt{box}\ (\texttt{unbox}_1\ t) : \Box A}$$

Figure 2.1: Typing judgments and some chosen equivalence judgments

$\Box A$ can be expanded to $\texttt{box}\ (\texttt{unbox}_1\ t)$. Note that in this rule, I directly refer to the modal offset 1. This is fine because in all four sub-systems modal offsets must be able to take 1. The β rule for $\Box$ is also interesting. For a concrete example, what should the right hand side of $\texttt{unbox}_2\ (\texttt{box}\ t)$ be when $n = 2$?

$$\vec{\Gamma}; \Delta_0; \Delta_1 \vdash \texttt{unbox}_2\ (\texttt{box}\ t) : A$$

where

$$\vec{\Gamma}; \cdot \vdash t : A$$

$\texttt{unbox}_2\ (\texttt{box}\ t)$ cannot simply reduce to t because the context stacks disagree. To match up the context stacks, Davies and Pfenning (2001) introduce the *modal transformation* (MoT) operation, written as $\{n/l\}$, which means to insert $n - 1$ contexts to the l'th position in the stack from top, to adjust the context stack of t. In the example, the right hand side should be $t\{2/0\}$. I will explain this operation in more details in the next section. The modal transformation operation is quite difficult to reason about and does not seem compatible with ordinary substitutions. It also poses some troubles in previous attempts to give a unified substitution calculus (Goubault-Larrecq, 1996). Thus, to understand Kripke-style systems better, I shall first give a unified substitution calculus which subsumes both ordinary substitutions and modal transformations.

2.2 Modal Transformations

The modal transformation operation is a structural operation; it characterizes the structural property of a context stack. This operation given by Davies and Pfenning (2001) includes the follow two cases:

$$\begin{aligned}
\texttt{box}\ t\{n/l\} &:= \texttt{box}\ (t\{n/l+1\}) \\
\texttt{unbox}_m\ t\{n/l\} &:= \begin{cases} \texttt{unbox}_m\ (t\{n/l-m\}) & \text{if } m \leq l \\ \texttt{unbox}_{n+m-1}\ t & \text{if } m > l \end{cases}
\end{aligned}$$

Let us not focus too much on the concrete details in this definition. A quick intuition is that this operation is not very simple to reason about. In the `unbox` case, there is a case analysis which involves a subtraction in each case. The purpose of this complex arithmetic is to maintain the following structural property:

Lemma 2.1 (Structural Property of Context Stacks). *If $\overrightarrow{\Gamma}; \Gamma_0; \Delta_0; \cdots; \Delta_l \vdash t : A$, then $\overrightarrow{\Gamma}; \Gamma_0; \cdots; (\Gamma_n, \Delta_0); \cdots; \Delta_l \vdash t\{n/l\} : A$, where Γ_1 to Γ_{n-1} are new and arbitrary contexts.*

This lemma is complex because it simultaneously characterizes *modal fusion* (when $n = 0$) and *modal weakening* (otherwise). To give a few examples:

- When $n = l = 0$, the lemma states that if $\overrightarrow{\Gamma}; \Gamma_0; \Delta_0 \vdash t : A$, then $\overrightarrow{\Gamma}; (\Gamma_0, \Delta_0) \vdash t\{0/0\} : A$. Note that Γ_0 and Δ_0 are fused in the conclusion, hence "modal fusion".

- When $n = 2$ and $l = 0$, the lemma states that if $\overrightarrow{\Gamma}; \Gamma_0; \Delta_0 \vdash t : A$, then $\overrightarrow{\Gamma}; \Gamma_0; \Gamma_1; (\Gamma_2, \Delta_0) \vdash t\{2/0\} : A$. A new context Γ_1 is inserted into the stack and the topmost context is (locally) weakened by Γ_2, hence "modal weakening".

$\boxed{\overrightarrow{\Gamma} \vdash t : A}$ New addition to the typing rules

$$\frac{\overrightarrow{\Delta} \vdash t : T \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash t[\overrightarrow{\sigma}] : T}$$

$\boxed{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}$ K-substitution $\overrightarrow{\sigma}$ maps $\overrightarrow{\Gamma}$ to $\overrightarrow{\Delta}$

$$\frac{}{\overrightarrow{\Gamma} \vdash \overrightarrow{I} : \overrightarrow{\Gamma}} \qquad \frac{}{\overrightarrow{\Gamma}; (\Gamma, x : A) \vdash \mathsf{wk} : \overrightarrow{\Gamma}; \Gamma} \qquad \frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'; \Gamma \qquad \overrightarrow{\Gamma} \vdash t : A}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma}, t : \overrightarrow{\Gamma}'; (\Gamma, x : A)}$$

$$\frac{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'' \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}'}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \circ \overrightarrow{\delta} : \overrightarrow{\Gamma}''} \qquad \frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \qquad |\overrightarrow{\Gamma}'| = n}{\overrightarrow{\Gamma}; \overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma}; \Uparrow^n : \overrightarrow{\Delta}; \cdot}$$

Figure 2.2: Typing judgments for K-substitutions

- In the β rule for $\square$, $\{n/0\}$ is used to transform t in context stack $\overrightarrow{\Gamma}; \cdot$ to $\overrightarrow{\Gamma}; \overrightarrow{\Delta}$ where $|\overrightarrow{\Delta}| = n$.
- When $l > 0$, the leading l contexts are skipped. When $n = 2$ and $l = 1$, the lemma states that if $\overrightarrow{\Gamma}; \Gamma_0; \Delta_0; \Delta_1 \vdash t : A$, then $\overrightarrow{\Gamma}; \Gamma_0; \Gamma_1; (\Gamma_2, \Delta_0); \Delta_1 \vdash t\{2/1\} : A$. Here Δ_1 is kept as is.

In $\lambda^{\square}$, ordinary substitutions are responsible for individual contexts in the stack while modal transformations handle the structure of context stacks. A unified substitution calculus needs to handle both individual contexts and context stacks at the same time. Is such a substitution calculus possible?

2.3 Explicit K-substitutions

The question left at the end of the previous section is positively answered in this section by K-substitutions (or Kripke-style substitutions). Compared to ordinary substitutions, which are typically just lists of terms mapping between two contexts, K-substitutions map between two *context stacks* and add a modal offset whenever the codomain context stack enters a new world. In other words, a K-substitutions not only contains a list of terms, but also a number of modal offsets. Moreover, to prepare for the normalization proof using an untyped domain model, I formulate K-substitutions as an explicit substitution calculus (Abadi et al., 1991; Abel, 2013). A non-explicit substitution formulation is available in my MFPS paper (Hu and Pientka, 2022a). The adjustments to the syntax are given below:

$$\begin{array}{rcll} s, t, u & := & \cdots \mid t[\overrightarrow{\sigma}] & \text{(Application of a K-substitution)} \\ \overrightarrow{\sigma}, \overrightarrow{\delta} & := & \overrightarrow{I} \mid \mathsf{wk} \mid \overrightarrow{\sigma}, t \mid \overrightarrow{\sigma} \circ \overrightarrow{\delta} \mid \overrightarrow{\sigma}; \Uparrow^n & \text{(K-substitutions, } \mathsf{Substs}\text{)} \end{array}$$

The typing rules for K-substitutions are given in Fig. 2.2. The additional syntax for terms $t[\overrightarrow{\sigma}]$ applies a K-substitution to a term. In this thesis, the binding precedence for substitution applications is very low. When I write $t\ s[\overrightarrow{\sigma}]$, I mean $(t\ s)[\overrightarrow{\sigma}]$. Parentheses are used when ambiguities occur. $t\ (s[\overrightarrow{\sigma}])$ unambiguously applies $\overrightarrow{\sigma}$ only to s. Similarly, $\texttt{unbox}_n\ t[\overrightarrow{\sigma}]$ denotes $(\texttt{unbox}_n\ t)[\overrightarrow{\sigma}]$. In an explicit substitution calculus, substitutions have their own syntax. In this case, there are five cases for a K-substitution. $\overrightarrow{I}$ is the identity K-substitution. It maps a context stack to itself. wk is the (local) weakening K-substitution. It weakens the topmost context by dropping the topmost variable. There are other ways to formulate a local weakening (K-)substitution but for minimality, the current formulation suffices. $\overrightarrow{\sigma}, t$ extends $\overrightarrow{\sigma}$ with a term t. The added term t replaces the topmost variable in the codomain context stack. Then there is composition $\overrightarrow{\sigma} \circ \overrightarrow{\delta}$, which composes two K-substitutions. So far, these four cases also appear in an explicit formulation for ordinary substitutions. What is different in K-substitutions, is the last case. $\overrightarrow{\sigma}; \Uparrow^n$ is a modal extension. It modal-extends $\overrightarrow{\sigma}$ with a new world:

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \qquad |\overrightarrow{\Gamma}'| = n}{\overrightarrow{\Gamma}; \overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma}; \Uparrow^n : \overrightarrow{\Delta}; \cdot}$$

After the modal extension, the codomain context stack is necessarily extended with an empty context. The domain context stack, on the other hand, grows from $\overrightarrow{\Gamma}$ to $\overrightarrow{\Gamma}, \overrightarrow{\Gamma}'$, as long as $|\overrightarrow{\Gamma}'| = n$. This operation resembles the modal transformation $\{n/0\}$

described in the previous section, which is the exact adjustment in the β rule for $\square$:

$$\frac{\overrightarrow{\Gamma}; \cdot \vdash t : A \qquad |\overrightarrow{\Delta}| = n}{\overrightarrow{\Gamma}; \overrightarrow{\Delta} \vdash \texttt{unbox}_n\ (\texttt{box}\ t) \approx t[\overrightarrow{I}; \Uparrow^n] : A} \qquad \frac{\overrightarrow{\Gamma}; (\Gamma, x : A) \vdash t : B \qquad \overrightarrow{\Gamma}; \Gamma \vdash s : A}{\overrightarrow{\Gamma}; \Gamma \vdash (\lambda x.t)\ s \approx t[\overrightarrow{I}, s] : B}$$

An explicit K-substitution formulation requires another judgment $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\delta} : \overrightarrow{\Delta}$ to characterize the equivalence between K-substitutions and additional rules to capture how K-substitutions interact with terms and among themselves. These additional rules are available in (Hu and Pientka, 2022b, Sec. 6) and are also placed in Appendix A for completeness.

2.4 Truncation and Truncation Offset

Consider the typing rule for `unbox`, how should a K-substitution interact with it? Concretely, if $\overrightarrow{\Delta} \vdash t : \square A$ and $|\overrightarrow{\Delta}'| = n$, then $\overrightarrow{\Delta}, \overrightarrow{\Delta}' \vdash \texttt{unbox}_n\ t : A$. Further assuming $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}; \overrightarrow{\Delta}'$, what should be the result of $\texttt{unbox}_n\ t[\overrightarrow{\sigma}]$? First, since t lives in $\overrightarrow{\Delta}$, there should be a K-substitution derived from $\overrightarrow{\sigma}$ with $\overrightarrow{\Delta}$ as the codomain context stack. This is easy as the codomain context stack of $\overrightarrow{\sigma}$ is $\overrightarrow{\Delta}; \overrightarrow{\Delta}'$, which is longer than $\overrightarrow{\Delta}$. Also $|\overrightarrow{\Delta}'| = n$. Intuitively, a *truncation* $\overrightarrow{\sigma} \mid n$ truncates $\overrightarrow{\sigma}$ by n and returns the desired K-substitution.

$$? \vdash \overrightarrow{\sigma} \mid n : \overrightarrow{\Delta}$$

Then $\overrightarrow{\sigma} \mid n$ can be applied to t since the codomain context stack matches up. However, what should be the domain context stack? The typing rule of modal extensions potentially adds an arbitrary number of contexts to the domain context stacks, but luckily, this number is precisely identical to the modal offset. Therefore, another operation, *truncation offset* $\mathcal{O}(\overrightarrow{\sigma}, n)$, is needed to add all the modal offsets in the part truncated from $\overrightarrow{\sigma}$. This fills in the question mark above:

$$\overrightarrow{\Gamma} \mid \mathcal{O}(\overrightarrow{\sigma}, n) \vdash \overrightarrow{\sigma} \mid n : \overrightarrow{\Delta}$$

where $\overrightarrow{\Gamma} \mid \mathcal{O}(\overrightarrow{\sigma}, n)$ truncates the context stack $\overrightarrow{\Gamma}$ by simply dropping $\mathcal{O}(\overrightarrow{\sigma}, n)$ topmost contexts from it. The truncation and truncation offset operations are particularly important in the equivalence rule for K-substituting `unbox`:

$$\frac{\overrightarrow{\Delta} \vdash t : \square A \qquad \overrightarrow{\Gamma}; \overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}; \overrightarrow{\Delta}' \qquad |\overrightarrow{\Delta}'| = n \qquad |\overrightarrow{\Gamma}'| = \mathcal{O}(\overrightarrow{\sigma}, n)}{\overrightarrow{\Gamma}; \overrightarrow{\Gamma}' \vdash \texttt{unbox}_n\ t[\overrightarrow{\sigma}] \approx \texttt{unbox}_{\mathcal{O}(\overrightarrow{\sigma}, n)}\ (t[\overrightarrow{\sigma} \mid n]) : A}$$

Note that on the right hand side, the new modal offset of `unbox` is $\mathcal{O}(\overrightarrow{\sigma}, n)$.

One advantage of the current explicit K-substitution calculus is that these two operations are defined simply by a recursion on the structures of the inputs:

Truncation $(\_ \mid \_)$

$$\begin{aligned}
\overrightarrow{\sigma} \mid 0 &:= \overrightarrow{\sigma} \\
\overrightarrow{I} \mid 1+n &:= \overrightarrow{I} \\
(\overrightarrow{\sigma}, t) \mid 1+n &:= \overrightarrow{\sigma} \mid 1+n \\
\mathsf{wk} \mid 1+n &:= \overrightarrow{I} \\
(\overrightarrow{\sigma}; \Uparrow^m) \mid 1+n &:= \overrightarrow{\sigma} \mid n \\
(\overrightarrow{\sigma} \circ \overrightarrow{\delta}) \mid 1+n &:= (\overrightarrow{\sigma} \mid 1+n) \circ (\overrightarrow{\delta} \mid \mathcal{O}(\overrightarrow{\sigma}, 1+n))
\end{aligned}$$

Truncation Offset $(\mathcal{O}(\_, \_))$

$$\begin{aligned}
\mathcal{O}(\overrightarrow{\sigma}, 0) &:= 0 \\
\mathcal{O}(\overrightarrow{I}, 1+n) &:= 1+n \\
\mathcal{O}((\overrightarrow{\sigma}, t), 1+n) &:= \mathcal{O}(\overrightarrow{\sigma}, 1+n) \\
\mathcal{O}(\mathsf{wk}, 1+n) &:= 1+n \\
\mathcal{O}(\overrightarrow{\sigma}; \Uparrow^m, 1+n) &:= m + \mathcal{O}(\overrightarrow{\sigma}, n) \\
\mathcal{O}(\overrightarrow{\sigma} \circ \overrightarrow{\delta}, 1+n) &:= \mathcal{O}(\overrightarrow{\delta}, \mathcal{O}(\overrightarrow{\sigma}, 1+n))
\end{aligned}$$

Both operations satisfy the following coherence conditions:

Lemma 2.2 (Coherence conditions)**.**

- *Coherence of addition: for all $\overrightarrow{\sigma}$, m and n, $\overrightarrow{\sigma} \mid (n+m) = (\overrightarrow{\sigma} \mid n) \mid m$ and $\mathcal{O}(\overrightarrow{\sigma}, n+m) = \mathcal{O}(\overrightarrow{\sigma}, n) + \mathcal{O}(\overrightarrow{\sigma} \mid n, m)$.*
- *Coherence of composition: for all $\overrightarrow{\sigma}$, $\overrightarrow{\delta}$ and m, $(\overrightarrow{\sigma} \circ \overrightarrow{\delta}) \mid n = (\overrightarrow{\sigma} \mid n) \circ (\overrightarrow{\delta} \mid \mathcal{O}(\overrightarrow{\sigma}, n))$ and $\mathcal{O}(\overrightarrow{\sigma} \circ \overrightarrow{\delta}, n) = \mathcal{O}(\overrightarrow{\delta}, \mathcal{O}(\overrightarrow{\sigma}, n))$.*

The truncation and truncation offset operations recur in both syntax and semantics. In fact, truncation and truncation offset form an algebra, which I refer to as a *truncoid*. Essentially, the semantic study of $\lambda^{\square}$ is just a study of various truncoids. Therefore, it is worth writing down the definitions of these truncoids before going into the discussion on semantics. A general truncoid is defined as follows:

Definition 2.1. A truncoid is a triple $(S, \_ \mid \_, \mathcal{O}(\_, \_))$, where

- S is a set;
- the truncation operation $\_ \mid \_$ takes an S and a modal offset and returns an S;
- the truncation offset operation $\mathcal{O}(\_, \_)$ takes an S and a modal offset and returns a modal offset.

where the coherence of addition hold:

$$s \mid 0 = s \text{ and } s \mid (n+m) = (s \mid n) \mid m$$
$$\mathcal{O}(s, 0) = 0 \text{ and } \mathcal{O}(s, n+m) = \mathcal{O}(s, n) + \mathcal{O}(s \mid n, m)$$

Following the common mathematical practice, I directly call S a truncoid. I have already shown that K-substitutions are a truncoid. Note that both operations use modal offsets. In particular, truncation offset even returns a modal offset. Thus truncation offset is required to be closed under possible value sets in Table 2.1. For K-substitutions, let us quickly examine this fact.

- In System K, 1 is the only possible value, so $\mathcal{O}(\overrightarrow{\sigma}, 1)$ is the only possible truncation offset expression, and this necessarily returns 1.
- In System T, a modal offset is either 0 or 1. Then the sum of zero or one number from $\{0, 1\}$ is also 0 or 1.
- In System $K4$, a modal offset takes all positive values. $\mathcal{O}(\overrightarrow{\sigma}, n)$ adds a positive number of positive numbers, which must result in a positive number.
- In System $S4$, the closure condition is trivial.

This closure condition modularizes the normalization proof so that I only need to give one proof for the normalization properties of all four sub-systems.

In later sections, I will describe truncoids other than K-substitutions. In fact, most truncoids have richer structures. *Applicative truncoids* allow a truncoid to be applied to another:

Definition 2.2. An applicative truncoid consists of a triple of truncoids (S_0, S_1, S_2) and an additional apply operation $\_[\_]$ which takes S_0 and S_1 and returns S_2. Moreover, the apply operation satisfies an extra coherence condition:

$$s[s'] \mid n = (s \mid n)[s' \mid \mathcal{O}(s, n)] \text{ and } \mathcal{O}(s[s'], n) = \mathcal{O}(s', \mathcal{O}(s, n))$$

Most semantic models in the semantics are applicative. For example, evaluation environments in Sec. 2.5 are applicative. K-substitutions are also applicative, as the apply operation is just composition. Generalizing this intuition, a specific applicative truncoid which behaves like substitutions is called a *substitutional truncoid*:

Definition 2.3. A substitutional truncoid S is an applicative truncoid (S, S, S) with an identity $\mathsf{id} \in S$. The apply operation is essentially composition, so I just write $\_ \circ \_$. The extra coherence conditions are for identity:

$$\mathsf{id} \mid n = \mathsf{id},\ \mathcal{O}(\mathsf{id}, n) = n,\ \mathsf{id} \circ s = s \text{ and } s \circ \mathsf{id} = s$$

and associativity:

$$(s_0 \circ s_1) \circ s_2 = s_0 \circ (s_1 \circ s_2)$$

K-substitutions are a substitutional truncoid. The semantic counterpart of K-substitutions, the untyped modal transformations (UMoTs), which model the Kripke structure in the semantics, are also substitutional. Another way to enrich an applicative truncoid, is to ask S_1 to be substitutional, which results in a *closed truncoid*:

Definition 2.4. A closed truncoid (S_0, S_1) is an applicative truncoid triple (S_0, S_1, S_0) where S_1 is a substitutional truncoid. id and $\_ \circ \_$ are identity and composition of S_1. The following additional conherence conditions are required:

- coherence of identity: $s[\mathsf{id}] = s$.
- coherence of composition: $s[s_1 \circ s_2] = s[s_1][s_2]$

We say that S_0 is closed under S_1. In the semantics, the evaluation environments are closed under UMoTs. K-substitutions are also closed under themselves. The laws of applicative and closed truncoids cover all the necessary properties for the normalization proof. Not all applicative truncoids are closed, e.g. the evaluation operation of K-substitutions in Sec. 2.7.

2.5 Untyped Domain Model

In this section, I define an untyped domain model (Abel, 2013). The purpose of this model is to establish a normalization by evaluation proof. Normalization by evaluation (NbE) is a specific way to prove normalization. It typically has two steps: first, terms are evaluated into some mathematical structure (in this case, the untyped domain model), and then second, normal forms are extracted from this structure.

In this case, the NbE algorithm has two components:

- In the first component (Sec. 2.7), an evaluation function evaluates well-typed terms into some values in the domain. During the evaluation process, all β redexes are eliminated.
- In the second component (Sec. 2.8), a readback function converts domain values into normal forms. The readback function is type-directed, and during the process, it does η expansion. Therefore, the results of the NbE algorithm are β-η normal forms.

One major advantage of this NbE algorithm is its simplicity to understand. A programmer could easily implement the NbE algorithm given below in the functional programming language of their choice. After finishing describing the NbE algorithm, I prove the completeness and soundness of the algorithm.

- In Sec. 2.9, I establish the completeness theorem, which states that equivalent terms are normalized to syntactically equal normal forms. Completeness is proved by constructing a partial equivalence relation (PER) model, which relates two values from the untyped domain.
- In Sec. 2.10 and 2.11, I establish the soundness theorem, which states that a well-typed term is equivalent to the normal form returned by the algorithm. The model to prove soundness is more sophisticated as it relies on the PER model defined for completeness. The model for soundness must *glue* both values in the untyped domain and the syntactic terms, so this model is a *gluing model*.

The untyped domain model is defined as follows:

$$
\begin{array}{rlr}
z & & \text{(Domain Variables in de Bruijn Levels, } \mathbb{N}) \\
a, b := & \Lambda(t, \overrightarrow{\rho}) \mid \mathtt{box}(a) \mid \uparrow^A (c) & \text{(Domain Values, } D) \\
c := & z \mid c\ d \mid \mathtt{unbox}(k, c) & \text{(Neutral Domain Values, } D^{\mathsf{Ne}}) \\
d := & \downarrow^A (a) & \text{(Normal Domain Values, } D^{\mathsf{Nf}}) \\
\rho \in & \mathsf{Env} := \mathbb{N} \to D & \text{(Local Evaluation Environments)} \\
\overrightarrow{\rho} \in & \mathsf{Envs} := \mathbb{N} \to \mathbb{N} \times \mathsf{Env} & \text{((Global) Evaluation Environments)}
\end{array}
$$

In the untyped domain, variables are represented by de Bruijn *levels*. Consider a topmost context $\Gamma, x : A, \Delta$ in some stack. The de Bruijn *index* of x is $|\Delta|$. Its corresponding de Bruijn level z, on the other hand, equals to $|\Gamma|$. De Bruijn levels assign a *unique absolute name* to each variable in each context to avoid handling local weakening of variables in the semantics. Evidently, these two representations satisfy $z + x + 1 = |\Gamma, x : A, \Delta|$. This equation will be used in the readback functions to correspond syntactic and semantic variables.

In this untyped domain, values are effectively partially β-reduced values and classified into three categories: values (D), neutral values (D^{Ne}) and normal values (D^{Nf}).

In D, $\mathtt{box}(a)$ models the values of `box`'ed term. A neutral value c is embedded into D when annotated with a type A by $\uparrow$. Following Abel (2013), I use *defunctionalization* (Ager et al., 2003; Reynolds, 1998) to capture open terms in the domain together with their surrounding evaluation environments, which enables formalization in Agda. A domain value $\Lambda(t, \overrightarrow{\rho})$ models a λ term and describes a syntactic body t together with an environment $\overrightarrow{\rho}$ which provides values for all the free variables in t except the topmost one bound by λ in the syntax.

A normal domain value is a domain value annotated with a type by $\downarrow$.

The NbE algorithm relies on local and global environments in the evaluation process. Local evaluation environments (Env) are functions mapping de Bruijn indices to domain values. Global evaluation environments (Envs), or just *environments*, are functions mapping modal offsets to tuples of modal offsets and Env's. An environment can be viewed as a *stream* of local environments paired with modal offsets. In the NbE algorithm, when evaluating a well-typed term, only a finite prefix of an environment is used, which is ensured by soundness.

$\mathsf{emp} :: \mathsf{Env}$ defines the empty local environment and $\mathsf{empty} :: \mathsf{Envs}$ defines the empty global environment.

$$\begin{aligned} \mathsf{emp} &:: \mathsf{Env} \\ \mathsf{emp}(\_) &:= \mathtt{ze} \end{aligned}$$

$$\begin{aligned} \mathsf{empty} &:: \mathsf{Envs} \\ \mathsf{empty}(\_) &:= (1, \mathsf{emp}) \end{aligned}$$

Their definitions do not matter here because they only provide default values which are guaranteed to be never used by soundness. There are two ways to extend environments. The `ext` function extends an environment with a modal offset n:

$$\begin{aligned} \mathtt{ext} &:: \mathsf{Envs} \to \mathbb{N} \to \mathsf{Envs} \\ \mathtt{ext}(\overrightarrow{\rho}, n)(0) &:= (n, \mathsf{emp}) \\ \mathtt{ext}(\overrightarrow{\rho}, n)(1+m) &:= \overrightarrow{\rho}(m) \end{aligned}$$

Furthermore, $\mathtt{ext}(\overrightarrow{\rho})$ is an abbreviation for $\mathtt{ext}(\overrightarrow{\rho}, 1)$. The `ext` function models modal extensions of a K-substitution $(\overrightarrow{\sigma}; \Uparrow^n)$, so n is not associated with any local environment, hence emp.

The `lext` function locally extends an environment with a value by inserting it into the topmost local environment. The `lext'` function is a helper which conses a value to a local environment and is used in `lext`:

$$\begin{aligned} \mathtt{lext'} &:: \mathsf{Env} \to D \to \mathsf{Env} \\ \mathtt{lext'}(\rho, a)(0) &:= a \\ \mathtt{lext'}(\rho, a)(1+m) &:= \rho(m) \end{aligned}$$

$$\begin{aligned} \mathtt{lext} &:: \mathsf{Envs} \to D \to \mathsf{Envs} \\ \mathtt{lext}(\overrightarrow{\rho}, a)(0) &:= (n, \mathtt{lext'}(\rho, a)) \qquad (\text{where } (n, \rho) := \overrightarrow{\rho}(0)) \\ \mathtt{lext}(\overrightarrow{\rho}, a)(1+m) &:= \overrightarrow{\rho}(1+m) \end{aligned}$$

The `drop` function drops the zeroth mapping from the topmost Env. It is needed for the interpretation of wk:

$$\begin{aligned}
\mathtt{drop} &:: \mathsf{Envs} \to \mathsf{Envs} \\
\mathtt{drop}(\overrightarrow{\rho})(0) &:= (n, m \mapsto \rho(1+m)) \qquad (\text{where } (n, \rho) := \overrightarrow{\rho}(0)) \\
\mathtt{drop}(\overrightarrow{\rho})(1+m) &:= \overrightarrow{\rho}(1+m)
\end{aligned}$$

Last, environments form a truncoid. The truncation and truncation offset operations on $\overrightarrow{\rho}$ are defined as:

$$\begin{aligned}
\_ \mid \_ &:: \mathsf{Envs} \to \mathbb{N} \to \mathsf{Envs} \\
(\overrightarrow{\rho} \mid n)(m) &:= \overrightarrow{\rho}(n+m) \\
\mathcal{O}(\_, \_) &:: \mathsf{Envs} \to \mathbb{N} \to \mathbb{N} \\
\mathcal{O}(\overrightarrow{\rho}, 0) &:= 0 \\
\mathcal{O}(\overrightarrow{\rho}, 1+n) &:= m + \mathcal{O}(\overrightarrow{\rho} \mid 1, n) \qquad (\text{where } (m, \_) := \overrightarrow{\rho}(0))
\end{aligned}$$

2.6 Untyped Modal Transformations (UMoTs)

To model $\square$, the semantics must handle the Kripke structure of $\lambda^{\square}$. In the untyped domain model, I employ an *internal* approach, where the Kripke structure is *internalized* by untyped modal transformations (UMoTs), ranged over by κ. Formally, a UMoT is just a function of type $\mathbb{N} \to \mathbb{N}$, modeling a *stream* of modal offsets. Given a domain value $a \in D$ and a UMoT κ, the UMoT application operation $a[\kappa]$ applies κ to a and denotes sending a to another world according to κ. The internalization further requires subsequent models to satisfy *monotonicity* w.r.t. UMoTs (see Lemmas 2.4 and 2.8), which is the root of other properties involving UMoTs. The internalization of UMoTs provides certain modularity. UMoTs have sufficiently captured the Kripke structures of all Systems K, T, $K4$ and $S4$, so as a bonus, their normalization is established in one single proof.

This approach is in contrast to the *external* approach by Gratzer et al. (2019), where the model is parameterized by an extra layer of poset[10] to capture the Kripke structure of $\square$. Subsequent proofs thus must explicitly quantify over this poset, making their proof more difficult to adapt to other modalities (Gratzer et al., 2020).

[10]partial order set

UMoTs include the following basic definitions:

$$\begin{array}{lrl}
\text{Truncation of UMoTs} & \_ \mid \_ & :: \text{UMoT} \to \mathbb{N} \to \text{UMoT} \\
 & (\kappa \mid n)\ (m) & := \kappa(n+m) \\
\text{Identity UMoT} & \overrightarrow{1} & :: \text{UMoT} \\
 & \overrightarrow{1}\ (\_) & := 1 \\
\text{Truncation Offset of UMoTs} & \mathcal{O}(\_,\_) & :: \text{UMoT} \to \mathbb{N} \to \mathbb{N} \\
 & \mathcal{O}(\kappa, 0) & := 0 \\
 & \mathcal{O}(\kappa, 1+n) & := \kappa(0) + \mathcal{O}(\kappa \mid 1, n) \\
\text{Cons of UMoTs} & \_;\Uparrow^{\_} & :: \text{UMoT} \to \mathbb{N} \to \text{UMoT} \\
 & (\kappa;\Uparrow^n)\ (0) & := n \\
 & (\kappa;\Uparrow^n)\ (m) & := \kappa(1+m)
\end{array}$$

$$\begin{array}{lrl}
\text{Composition of UMoTs} & \_ \circ \_ & :: \text{UMoT} \to \text{UMoT} \to \text{UMoT} \\
 & (\kappa \circ \kappa')\ (0) & := \mathcal{O}(\kappa', \kappa(0)) \\
 & (\kappa \circ \kappa')\ (1+n) & := ((\kappa \mid 1) \circ (\kappa' \mid \kappa(0)))(n)
\end{array}$$

Cons of UMoTs is defined in a way similar to environments. UMoTs have composition, and we use $\overrightarrow{1}$ to represent the identity UMoT in our setting. As previously mentioned, UMoTs form a substitutional truncoid. A quick intuition is that UMoTs behave like substitutions in the semantics, except that they only bring values from one world to another without touching the variables.

Then I give the definitions for applying a UMoT to domain values and environments:

$$\begin{aligned}
\texttt{box}(a)[\kappa] &:= \texttt{box}(a[\kappa; \Uparrow^1]) \\
\Lambda(t, \overrightarrow{\rho})[\kappa] &:= \Lambda(t, \overrightarrow{\rho}[\kappa]) \\
\uparrow^A (c)[\kappa] &:= \uparrow^A (c[\kappa]) \\
z[\kappa] &:= z \\
c\ d[\kappa] &:= (c[\kappa])\ (d[\kappa]) \\
\texttt{unbox}(k, c)[\kappa] &:= \texttt{unbox}(\mathcal{O}(\kappa, k), c[\kappa \mid k]) \\
\downarrow^A (a)[\kappa] &:= \downarrow^A (a[\kappa]) \\
\overrightarrow{\rho}[\kappa](0) &:= (\mathcal{O}(\kappa, k), \rho[\kappa]) \qquad (\text{where } (k, \rho) := \overrightarrow{\rho}(0)) \\
\overrightarrow{\rho}[\kappa](1 + n) &:= \overrightarrow{\rho} \mid 1[\kappa \mid \mathcal{O}(\overrightarrow{\rho}, 1)](n)
\end{aligned}$$

Similar to the syntax, $a[\kappa]$ takes a very low parsing precedence. Most cases just recursively push κ inwards, except the `box` and `unbox` cases. In the `box` case, $\kappa; \Uparrow^1$ instructs the recursion to enter a new world, indicated by cons'ing 1 to κ. The `unbox` case is similar to the rule in Sec. 2.4. The recursion is $c[\kappa \mid n]$ because c is in the n-th previous world. The `unbox` level is adjusted to $\mathcal{O}(\kappa, n)$ for coherence with the Kripke structure. The apply operation for a local environment ρ is just defined by applying κ to all values within pointwise. The apply operation for $\overrightarrow{\rho}$ can be motivated by making the triple $(\mathsf{Envs}, \text{UMoT}, \mathsf{Envs})$ an applicative truncoid. Indeed, this is the unique definition (up to isomorphism) to prove $\overrightarrow{\rho}[\kappa] \mid n = (\overrightarrow{\rho} \mid n)[\kappa \mid \mathcal{O}(\overrightarrow{\rho}, n)]$. Here, the formulation of truncoids does pay off; it provides guidance to quickly derive the right definition of operations. Moreover, since UMoTs are substitutional, Envs intuitively should be closed under UMoTs. Indeed, this fact can be examined by checking the necessary laws imposed by a closed truncoid.

2.7 Evaluation

Next I define the evaluation functions ($\llbracket\_\rrbracket$), which, given $\overrightarrow{\rho}$, evaluates a syntactic term to a domain value or evaluates a K-substitution to another environment. The procedure only terminates for well-formed terms and substitutions, and might loop for ill-typed inputs. In the following sections, I will define a number of partial functions (denoted by $\rightharpoonup$), which cannot be directly formalized in Agda. Instead, I define them as *functional relations* between inputs and outputs, and prove that equal inputs produce equal outputs. When a conclusion refers to a result of a partial function, this result is implicitly existentially quantified.

$$\begin{aligned}
[\![\_]\!] &: \mathsf{Exp} \rightharpoonup \mathsf{Envs} \to D \\
[\![x]\!](\overrightarrow{\rho}) &:= \rho(x) \quad \text{(where } (\_, \rho) := \overrightarrow{\rho}(0)) \\
[\![\mathtt{box}\ t]\!](\overrightarrow{\rho}) &:= \mathtt{box}([\![t]\!](\mathtt{ext}(\overrightarrow{\rho}))) \\
[\![\mathtt{unbox}_n\ t]\!](\overrightarrow{\rho}) &:= \mathtt{unbox}\cdot(\mathcal{O}(\overrightarrow{\rho}, n), [\![t]\!](\overrightarrow{\rho} \mid n)) \\
[\![\lambda x.t]\!](\overrightarrow{\rho}) &:= \Lambda(t, \overrightarrow{\rho}) \\
[\![t\ s]\!](\overrightarrow{\rho}) &:= [\![t]\!](\overrightarrow{\rho}) \cdot [\![s]\!](\overrightarrow{\rho}) \\
[\![t[\overrightarrow{\sigma}]]\!](\overrightarrow{\rho}) &:= [\![t]\!]([\![\overrightarrow{\sigma}]\!](\overrightarrow{\rho}))
\end{aligned}$$

$$\begin{aligned}
[\![\_]\!] &:: \mathsf{Substs} \rightharpoonup \mathsf{Envs} \to \mathsf{Envs} \\
[\![\overrightarrow{I}]\!](\overrightarrow{\rho}) &:= \overrightarrow{\rho} \\
[\![\mathsf{wk}]\!](\overrightarrow{\rho}) &:= \mathtt{drop}(\overrightarrow{\rho}) \\
[\![\overrightarrow{\sigma}, t]\!](\overrightarrow{\rho}) &:= \mathtt{lext}([\![\overrightarrow{\sigma}]\!](\overrightarrow{\rho}), [\![t]\!](\overrightarrow{\rho})) \\
[\![\overrightarrow{\sigma}; \Uparrow^n]\!](\overrightarrow{\rho}) &:= \mathtt{ext}([\![\overrightarrow{\sigma}]\!](\overrightarrow{\rho} \mid n), \mathcal{O}(\overrightarrow{\rho}, n)) \\
[\![\overrightarrow{\sigma} \circ \overrightarrow{\delta}]\!](\overrightarrow{\rho}) &:= [\![\overrightarrow{\sigma}]\!]([\![\overrightarrow{\delta}]\!](\overrightarrow{\rho}))
\end{aligned}$$

Note that here $(\mathsf{Substs}, \mathsf{Envs}, \mathsf{Envs})$ forms an applicative truncoid when the evaluation terminates. To evaluate function applications and `unbox`, the following partial functions either perform β reductions or form neutral values:

$$\begin{aligned}
\mathtt{unbox}\cdot &: \mathbb{N} \rightharpoonup D \rightharpoonup D \\
\mathtt{unbox}\cdot(k, \mathtt{box}(a)) &:= a[\overrightarrow{1}; \Uparrow^k] \\
\mathtt{unbox}\cdot(k, \uparrow^{\square A}(c)) &:= \uparrow^A(\mathtt{unbox}(k, c))
\end{aligned}$$

$$\begin{aligned}
\_\cdot\_ &: D \rightharpoonup D \rightharpoonup D \\
(\Lambda(t, \overrightarrow{\rho})) \cdot a &:= [\![t]\!](\mathtt{lext}(\overrightarrow{\rho}, a)) \\
(\uparrow^{A \longrightarrow B}(c)) \cdot a &:= \uparrow^B(c\ \downarrow^A(a))
\end{aligned}$$

The reduction case for `unbox` matches the syntactic β rule by applying a UMoT $\overrightarrow{1}; \Uparrow^k$ to a. This UMoT sends a to k worlds in the future.

2.8 Readback Functions

After evaluating a term to D, all β redexes are eliminated. The last step is the readback functions, which read from D back to normal forms and do the η expansion at the same time to obtain β-η normal forms:

$$\begin{aligned}
\mathsf{R}^{\mathsf{Nf}} &: \overrightarrow{\mathbb{N}} \rightharpoonup D^{\mathsf{Nf}} \rightharpoonup \mathsf{Nf} \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\Box A}(a)) &:= \texttt{box}\ \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z};0}(\downarrow^{A}(\texttt{unbox}\cdot(1,a))) \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z};z}(\downarrow^{A\longrightarrow B}(a)) &:= \lambda x.\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z};z+1}(\downarrow^{B}(a\cdot\uparrow^{A}(z))) \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\mathsf{Base}}(\uparrow^{\mathsf{Base}}(c))) &:= \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c) \\
\mathsf{R}^{\mathsf{Ne}} &: \overrightarrow{\mathbb{N}} \rightharpoonup D^{\mathsf{Ne}} \rightharpoonup \mathsf{Ne} \\
\mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z};z'}(z) &:= x \\
&\quad\text{(where the de Bruijn index of } x \text{ is computed by } \max(z'-z-1,0)\text{)} \\
\mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c\ d) &:= \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c)\ \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(d) \\
\mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(\texttt{unbox}(k,c)) &:= \texttt{unbox}_k\ \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}|k}(c)
\end{aligned}$$

The readback process consists of two functions: R^{Nf} reads back a normal form and R^{Ne} reads back a neutral form. R^{Nf} is type-directed, so η expansion is performed. A readback function takes as an argument $\overrightarrow{z} :: \overrightarrow{\mathbb{N}}$, which is a nonempty list of natural numbers. Each number in this list records the length of the context in that position of the context stack. This list supplies new de Bruijn levels (i.e. new absolute and fresh names) as the readback process continues. In the $\Box A$ case, 0 is pushed to the list because `box` enters a new world with no assumptions. In the function case, the topmost context is extended by one due to the argument of λ, so the topmost de Bruijn level is also incremented by one. In the `unbox` case, $\overrightarrow{z}$ is truncated by n in order to correctly keep track of the lengths of contexts in the stack as the context stack is also truncated. In the variable case in R^{Ne}, the de Bruijn index of x is computed by the aforementioned formula $z' - z - 1$. If readback is applied to the result of evaluating a well typed term, then this formula is always non-negative, so the cut off at 0 is neglected most of the time.

Given evaluation and readback, the NbE algorithm is defined as first evaluating a term to the domain and then reading back as a normal form:

Definition 2.5. For $\overrightarrow{\Gamma} \vdash t : A$, the NbE algorithm is

$$\mathsf{nbe}^{A}_{\overrightarrow{\Gamma}}(t) := \mathsf{R}^{\mathsf{Nf}}_{\mathtt{map}(\Gamma \mapsto |\Gamma|, \overrightarrow{\Gamma})}(\downarrow^{A} (\llbracket t \rrbracket(\uparrow^{\overrightarrow{\Gamma}})))$$

where the initial environment $\uparrow^{\overrightarrow{\Gamma}}$ is defined by the structure of $\overrightarrow{\Gamma}$:

$$\begin{aligned}
\uparrow^{\overrightarrow{\Gamma}} &:: \mathsf{Envs} \\
\uparrow^{\epsilon} &:= \mathsf{empty} \\
\uparrow^{\overrightarrow{\Gamma};\Gamma} (0) &:= (1, x \mapsto \uparrow^{A} (z) \text{ if } x : A \in \Gamma) \\
\uparrow^{\overrightarrow{\Gamma};\Gamma} (1 + n) &:= \uparrow^{\overrightarrow{\Gamma}} (n)
\end{aligned}$$

where the de Bruijn level of z is computed by $|\Gamma| - x - 1$ as described in Sec. 2.5.

2.9 PER Model And Completeness

The full NbE algorithm has been given in the previous sections. In this section, I follow Abel (2013) and define a partial equivalence relation (PER) model for the untyped domain. The idea of the PER model is to relate two "equivalent" domain values once they are read back as normal forms. The PER model is defined by recursion on the structure of types, which determines how terms are equivalent. Then I prove the completeness theorem, which states that equivalent terms evaluate to an equal normal form. There are two steps to establish completeness:

- the fundamental theorems which prove soundness of the PER model, and
- the realizability theorem, which states that values related by the PER model have an equal normal form.

The following two inference rules define two special PERs between two domain values:

$$\frac{\forall\ \overrightarrow{z}, \kappa\ .\ \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(d[\kappa]) = \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(d'[\kappa])}{d \approx d' \in \mathit{Nf}} \qquad \frac{\forall\ \overrightarrow{z}, \kappa\ .\ \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c[\kappa]) = \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c'[\kappa])}{c \approx c' \in \mathit{Ne}}$$

where $\mathit{Nf} \subseteq D^{\mathsf{Nf}} \times D^{\mathsf{Nf}}$ and $\mathit{Ne} \subseteq D^{\mathsf{Ne}} \times D^{\mathsf{Ne}}$. *Nf* relates two normal domain values iff their readbacks are equal given any context stack and UMoT. *Ne* is defined similarly.

Later, the realizability theorem (Theorem 2.3) shows that the PER model to be given below is subsumed by *Nf*, so any two related values have equal readbacks as normal forms, through which completeness is established.

The PER model is universally quantified over UMoTs. This universal quantification is crucial in the PER model. Due to the substitutional truncoid structure of UMoTs, universally quantifying UMoTs *internalizes* the Kripke structure of $\lambda^{\square}$ and delegates the handling of the said structure to UMoTs, so that the whole normalization proof is completely oblivious to the exact structure $\square$ possesses, making the proof applicable to all four systems without any modification as described in Sec. 2.6.

The PER model is defined recursively on a type and relates two domain values:

$$\begin{aligned}
[\![A]\!] &\subseteq D \times D \\
[\![\texttt{Base}]\!] &:= \{\ (\uparrow^{\texttt{Base}}(c), \uparrow^{\texttt{Base}}(c')) \mid c \approx c' \in Ne\ \} \\
[\![\square A]\!] &:= \{\ (a, b) \mid \forall\ k, \kappa\ .\ \texttt{unbox} \cdot (k, a[\kappa]) \approx \texttt{unbox} \cdot (k, b[\kappa]) \in [\![A]\!]\ \} \\
[\![A \longrightarrow B]\!] &:= \{\ (a, b) \mid \forall\ \kappa, a' \approx b' \in [\![A]\!]\ .\ a[\kappa] \cdot a' \approx b[\kappa] \cdot b' \in [\![B]\!]\ \}
\end{aligned}$$

In the $\square$ case, two values are related if the results of `unbox` are related under any offset k and any UMoT κ. Functions are related in a similar way. Two values are related if the results of function applications remain related under any UMoT κ and given any related arguments.

The realizability theorem below states that related values are read back equal. Realizability of the PER model is essential to establish the completeness and the soundness theorems.

Theorem 2.3 (Realizability). *For all type A,*

- *if $c \approx c' \in Ne$, then $\uparrow^A(c) \approx \uparrow^A(c') \in [\![A]\!]$;*
- *if $a \approx b \in [\![A]\!]$, then $\downarrow^A(a) \approx \downarrow^A(b) \in Nf$.*

Proof. Induction on A. $\square$

The PER model between values is scaled to a PER between local evaluation environments. The PER model between global environments are defined by also requiring modal offsets to be equal pointwise.

$$
\begin{aligned}
[\![\Gamma]\!] &\subseteq \mathsf{Env} \times \mathsf{Env} \\
[\![\Gamma]\!] &:= \{\ (\rho, \rho') \mid \rho(x) \approx \rho'(x) \in [\![A]\!] \text{ if } x : A \in \Gamma\ \} \\
[\![\overrightarrow{\Gamma}]\!] &\subseteq \mathsf{Envs} \times \mathsf{Envs} \\
[\![\epsilon]\!] &:= \{\ (\overrightarrow{\rho}, \overrightarrow{\rho}')\ \} \\
[\![\overrightarrow{\Gamma}; \Gamma]\!] &:= \{\ (\overrightarrow{\rho}, \overrightarrow{\rho}') \mid k = k' \text{ and } \rho \approx \rho' \in \Gamma \text{ and } \overrightarrow{\rho} \mid 1 \approx \overrightarrow{\rho}' \mid 1 \in [\![\overrightarrow{\Gamma}]\!] \\
&\qquad\qquad \text{where } (k, \rho) := \overrightarrow{\rho}(0) \text{ and } (k', \rho') := \overrightarrow{\rho}'(0)\ \}
\end{aligned}
$$

In the base case, two environments are related unconditionally. This is fine because context stacks are always non-empty, so this case is never hit.

The monotonicity lemma ensures that the Kripke structure is successfully internalized, so subsequent proofs are unaware of the exact structure of $\square$, making the NbE proof structure very general. Due to the composition of UMoTs, all properties describing the Kripke structure eventually boil down to monotonicity.

Lemma 2.4 (Monotonicity)**.**

- *If* $a \approx b \in [\![T]\!]$, *then* $a[\kappa] \approx b[\kappa] \in [\![T]\!]$.
- *If* $\overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\overrightarrow{\Gamma}]\!]$, *then* $\overrightarrow{\rho}[\kappa] \approx \overrightarrow{\rho}'[\kappa] \in [\![\overrightarrow{\Gamma}]\!]$.

The lemma says that the PER models are invariant under any UMoT.

The semantic judgments for completeness are defined as:

$$
\begin{aligned}
\overrightarrow{\Gamma} \vDash t \approx t' : A &:= \forall\ \overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\overrightarrow{\Gamma}]\!]\ .\ [\![t]\!](\overrightarrow{\rho}) \approx [\![t']\!](\overrightarrow{\rho}') \in [\![A]\!] \\
\overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta} &:= \forall\ \overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\overrightarrow{\Gamma}]\!]\ .\ [\![\overrightarrow{\sigma}]\!](\overrightarrow{\rho}) \approx [\![\overrightarrow{\sigma}']\!](\overrightarrow{\rho}') \in [\![\overrightarrow{\Delta}]\!] \\
\overrightarrow{\Gamma} \vDash t : A &:= \overrightarrow{\Gamma} \vDash t \approx t : A \\
\overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} : \overrightarrow{\Delta} &:= \overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} \approx \overrightarrow{\sigma} : \overrightarrow{\Delta}
\end{aligned}
$$

Soundness of the semantic judgments are proved by the following fundamental theorem.

Theorem 2.5 (Fundamental)**.**

- *If* $\overrightarrow{\Gamma} \vdash t : T$, *then* $\overrightarrow{\Gamma} \vDash t : T$.
- *If* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$, *then* $\overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} : \overrightarrow{\Delta}$.

- *If* $\overrightarrow{\Gamma} \vdash t \approx t' : T$*, then* $\overrightarrow{\Gamma} \vDash t \approx t' : T$*.*
- *If* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}$*, then* $\overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}$*.*

Proof. Proceed by a mutual induction on the syntactic judgments. □

The fundamental theorem states that all syntactic judgments are semantically sound w.r.t. the semantic judgments. More specifically, the completeness theorem of the NbE algorithm is an instantiation of the fundamental theorem.

Theorem 2.6 (Completeness)**.** *If* $\overrightarrow{\Gamma} \vdash t \approx t' : T$*, then* $\mathsf{nbe}^{T}_{\overrightarrow{\Gamma}}(t) = \mathsf{nbe}^{T}_{\overrightarrow{\Gamma}}(t')$*.*

Proof. $\overrightarrow{\Gamma} \vdash t \approx t' : T$ implies $\overrightarrow{\Gamma} \vDash t \approx t' : T$ by the fundamental theorem. Combining realizability, the goal is concluded. □

2.10 Restricted Weakenings

In the previous section, I have established the completeness theorem. In the next step, I will establish the soundness proof, central to which is a *Kripke* gluing model. The gluing model is Kripke because it is monotonic w.r.t. a special class of K-substitutions, restricted weakenings. Similar to completeness, the proof of soundness also has two steps:

- the fundamental theorems, and
- the realizability theorem stating that a term is equivalent to the readback of its related value.

In this section, I first give the definition of restricted weakenings.

Restricted weakenings are a special class of K-substitutions which characterize the possible changes in context stacks *during evaluation*. Therefore, if a term and a value are related, their relation must be stable under restricted weakenings, hence introducing a Kripke structure to the gluing model.

Definition 2.6. A K-substitution $\overrightarrow{\sigma}$ is a restricted weakening if it inductively satisfies

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{I} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Delta}} \qquad \frac{\overrightarrow{\Gamma} \vdash_r \overrightarrow{\sigma}' : \overrightarrow{\Delta}; \Delta, x : A \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \mathsf{wk} \circ \overrightarrow{\sigma}' : \overrightarrow{\Delta}; \Delta}{\overrightarrow{\Gamma} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Delta}; \Delta}$$

$$\frac{\overrightarrow{\Gamma} \vdash_r \overrightarrow{\sigma}' : \overrightarrow{\Delta} \qquad |\overrightarrow{\Gamma}'| = n \qquad \overrightarrow{\Gamma}; \Gamma' \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}'; \Uparrow^n : \overrightarrow{\Delta}; \cdot}{\overrightarrow{\Gamma}; \Gamma' \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Delta}; \cdot}$$

Effectively, a restricted weakening can only do either local weakenings (wk) or modal extensions ($\_; \Uparrow^n$), because only these two cases are possible during evaluation. Since restricted weakenings also have identity and composition, we require the gluing model to respect them.

2.11 Gluing Model And Soundness

In this section, I will define the gluing model $t \sim a \in (\!|T|\!)_{\overrightarrow{\Gamma}}$ to relate a well-typed term with a domain value a. As previously said, the gluing model is *Kripke*, in the sense that it should respect restricted weakenings. Though it is quite clear that a restricted weakening is directly applicable to a term t, how does a domain value a respect a K-substitution? This is achieved by UMoT extraction ($\mathtt{mt}(\_)$)[11], which extracts a UMoT from a K-substitution (not just a restricted weakening).

$$\begin{aligned}
\mathtt{mt}(\_) &:: \mathsf{Substs} \to \text{UMoT} \\
\mathtt{mt}(\overrightarrow{I}) &:= \overrightarrow{1} \\
\mathtt{mt}(\mathsf{wk}) &:= \overrightarrow{1} \\
\mathtt{mt}(\overrightarrow{\sigma}, t) &:= \mathtt{mt}(\overrightarrow{\sigma}) \\
\mathtt{mt}(\overrightarrow{\sigma}; \Uparrow^n) &:= \mathtt{mt}(\overrightarrow{\sigma}); \Uparrow^n \\
\mathtt{mt}(\overrightarrow{\sigma} \circ \overrightarrow{\delta}) &:= \mathtt{mt}(\overrightarrow{\sigma}) \circ \mathtt{mt}(\overrightarrow{\delta})
\end{aligned}$$

Moreover, for conciseness, given a K-substitution $\overrightarrow{\sigma}$ and a domain value a, I write $a[\overrightarrow{\sigma}]$ for $a[\mathtt{mt}(\overrightarrow{\sigma})]$ unless the distinction is worth emphasizing.

The following are two gluing relations. They state that a term and a value are related if they are syntactically equivalent after respective readback under any restricted

[11]$\mathtt{mt}()$ stands for "we modal transform a K-substitution into a UMoT."

weakening. In particular, the soundness theorem simply requires $\overline{A}_{\overrightarrow{\Gamma}}$ to relate any well-typed term and its evaluation.

$$
\begin{aligned}
&\underline{A}_{\overrightarrow{\Gamma}} \subseteq \mathsf{Exp} \times \mathcal{D} \\
&\underline{A}_{\overrightarrow{\Gamma}} := \{\ (t, \uparrow^{A} (c)) \mid \overrightarrow{\Gamma} \vdash t : A \text{ and } \forall\ \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}\ .\ \overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c[\overrightarrow{\sigma}]) : A\ \} \\
&\overline{A}_{\overrightarrow{\Gamma}} \subseteq \mathsf{Exp} \times \mathcal{D} \\
&\overline{A}_{\overrightarrow{\Gamma}} := \{\ (t, a) \mid \overrightarrow{\Gamma} \vdash t : A \text{ and } \forall \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}\ .\ \overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{A} (a[\overrightarrow{\sigma}])) : A\ \}
\end{aligned}
$$

where $\overrightarrow{z} := \mathtt{map}(\Delta \mapsto |\Delta|, \overrightarrow{\Delta})$. In other words, $\overrightarrow{z}$ measures the lengths of all contexts in the stack. It is used to generate new de Bruijn levels as we proceed reading back. The goal of these two relations is to show that the gluing model $(\!|A|\!)_{\overrightarrow{\Gamma}}$ is subsumed by $\overline{A}_{\overrightarrow{\Gamma}}$ via realizability (Theorem 2.7), which is then used to prove the soundness theorem.

The gluing model is defined as

$$
\begin{aligned}
(\!|A|\!)_{\overrightarrow{\Gamma}} &\subseteq \mathsf{Exp} \times \mathcal{D} \\
(\!|\mathtt{Base}|\!)_{\overrightarrow{\Gamma}} &:= \underline{\mathtt{Base}}_{\overrightarrow{\Gamma}} \\
(\!|\square A|\!)_{\overrightarrow{\Gamma}} &:= \{\ (t, a) \mid \overrightarrow{\Gamma} \vdash t : \square A \text{ and} \\
&\qquad \forall\ \overrightarrow{\Delta}', \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}\ .\ \mathtt{unbox}_{|\overrightarrow{\Delta}'|}\ (t[\overrightarrow{\sigma}]) \sim \mathtt{unbox} \cdot (|\overrightarrow{\Delta}'|, a[\overrightarrow{\sigma}]) \in (\!|A|\!)_{\overrightarrow{\Delta};\overrightarrow{\Delta}'}\ \} \\
(\!|A \longrightarrow B|\!)_{\overrightarrow{\Gamma}} &:= \{\ (t, a) \mid \overrightarrow{\Gamma} \vdash t : A \longrightarrow B \text{ and} \\
&\qquad \forall\ \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}, s \sim b \in (\!|A|\!)_{\overrightarrow{\Delta}}\ .\ t[\overrightarrow{\sigma}]\ s \sim a[\overrightarrow{\sigma}] \cdot b \in (\!|B|\!)_{\overrightarrow{\Delta}}\ \}
\end{aligned}
$$

In the $\square A$ case, t and a are related if the results of `unbox` from any future work are related under any restricted weakening. In the function case, t and a are related if the results of function applications are related under any restricted weakening given any related arguments.

The gluing model also has a realizability theorem:

Theorem 2.7 (Realizability). $\underline{A}_{\overrightarrow{\Gamma}} \subseteq (\!|A|\!)_{\overrightarrow{\Gamma}} \subseteq \overline{A}_{\overrightarrow{\Gamma}}$, *i.e.*

- *If* $t \sim a \in \underline{A}_{\overrightarrow{\Gamma}}$, *then* $t \sim a \in (\!|A|\!)_{\overrightarrow{\Gamma}}$.
- *If* $t \sim a \in (\!|A|\!)_{\overrightarrow{\Gamma}}$, *then* $t \sim a \in \overline{A}_{\overrightarrow{\Gamma}}$.

Proof. Induction on A. $\square$

The gluing model for types is then generalized to the gluing model for context stacks.

$$
\begin{aligned}
(\!|\overrightarrow{\Gamma}|\!)_{\overrightarrow{\Delta}} &\subseteq \mathsf{Substs} \times \mathsf{Envs} \\
(\!|\epsilon|\!)_{\overrightarrow{\Delta}} &:= \{\ (\overrightarrow{\sigma}, \overrightarrow{\rho})\ \} \\
(\!|\overrightarrow{\Gamma};\Gamma|\!)_{\overrightarrow{\Delta}} &:= \{\ (\overrightarrow{\sigma}, \overrightarrow{\rho}) \mid \overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma};\Gamma \text{ and } \mathcal{O}(\overrightarrow{\sigma}, 1) = k \text{ and } \overrightarrow{\sigma} \mid 1 \sim \overrightarrow{\rho} \mid 1 \in (\!|\overrightarrow{\Gamma}|\!)_{\overrightarrow{\Delta}|k} \\
&\qquad \text{and } \forall\ x : A \in \Gamma\ .\ x[\overrightarrow{\sigma}] \sim \rho(x) \in (\!|A|\!)_{\overrightarrow{\Delta}} \text{ where } (k, \rho) := \overrightarrow{\rho}(0)\ \}
\end{aligned}
$$

Again, the base case for ϵ will not be hit for well-formed judgments so it is simply set as true. Otherwise, to relate $\overrightarrow{\sigma}$ and $\overrightarrow{\rho}$, their truncation and truncation offset must agree. Also terms in the K-substitution and values in the environment must also be related pointwise.

The monotonicity of the gluing model is characterized as

Lemma 2.8 (Monotonicity).

- *If* $t \sim a \in [\![A]\!]_{\overrightarrow{\Delta}}$ *and* $\overrightarrow{\Gamma} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Delta}$, *then* $t[\overrightarrow{\sigma}] \sim a[\overrightarrow{\sigma}] \in (\!|A|\!)_{\overrightarrow{\Gamma}}$.
- *If* $\overrightarrow{\sigma} \sim \overrightarrow{\rho} \in (\!|\overrightarrow{\Gamma}|\!)_{\overrightarrow{\Delta}}$, *given* $\overrightarrow{\Delta}' \vdash_r \overrightarrow{\delta} : \overrightarrow{\Delta}$, *then* $\overrightarrow{\sigma} \circ \overrightarrow{\delta} \sim \overrightarrow{\rho}[\overrightarrow{\delta}] \in (\!|\overrightarrow{\Gamma}|\!)_{\overrightarrow{\Delta}'}$.

Proof. Induction on A. □

Monotonicity ensures that the gluing model is stable under restricted weakenings. Restricted weakenings apply to both syntax and semantics because they contain modal extensions to instruct both sides to travel among Kripke worlds.

Semantic judgments for the soundness theorem are defined as follows:

$$
\begin{aligned}
\overrightarrow{\Gamma} \Vdash t : A &:= \forall\ \overrightarrow{\sigma} \sim \overrightarrow{\rho} \in (\!|\overrightarrow{\Gamma}|\!)_{\overrightarrow{\Delta}}\ .\ t[\overrightarrow{\sigma}] \sim [\![t]\!](\overrightarrow{\rho}) \in (\!|A|\!)_{\overrightarrow{\Delta}} \\
\overrightarrow{\Gamma} \Vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}' &:= \forall\ \overrightarrow{\sigma} \sim \overrightarrow{\rho} \in (\!|\overrightarrow{\Gamma}|\!)_{\overrightarrow{\Delta}}\ .\ \overrightarrow{\delta} \circ \overrightarrow{\sigma} \sim [\![\overrightarrow{\delta}]\!](\overrightarrow{\rho}) \in (\!|\overrightarrow{\Gamma}'|\!)_{\overrightarrow{\Delta}}
\end{aligned}
$$

Soundness of the semantic judgments for the gluing model is proved by the fundamental theorem.

Theorem 2.9 (Fundamental).

- *If* $\overrightarrow{\Gamma} \vdash t : T$, *then* $\overrightarrow{\Gamma} \Vdash t : T$.
- *If* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$, *then* $\overrightarrow{\Gamma} \Vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$.

Proof. Proceed by a mutual induction on the syntactic judgments. $\square$

Theorem 2.10 (Soundness)**.** *If* $\overrightarrow{\Gamma} \vdash t : T$*, then* $\overrightarrow{\Gamma} \vdash t \approx \mathsf{nbe}^{T}_{\overrightarrow{\Gamma}}(t) : T$*.*

Proof. Applying the fundamental theorem gives $t[\overrightarrow{I}] \sim [\![t]\!]_{\overrightarrow{\Gamma}}(\uparrow^{\overrightarrow{\Gamma}}) \in (\!|T|\!)_{\overrightarrow{\Gamma}}$. The goal is concluded by realizability. $\square$

2.12 Summary

At this point, I have finished the strong normalization proof of $\lambda^{\square}$. The convertibility problem is trivially decidable, because I can simply compare equality between normal forms after normalization. In this chapter, I develop K-substitutions, a uniform representation of a substitution calculus of $\lambda^{\square}$, and an NbE algorithm. This setup continues to scale naturally in dependent types as we will see very soon in the next chapter. This also shows the effectiveness of my methodology discussed in Sec. 1.2.

3

Mint: A Kripke-style Modal Dependent Type Theory

In the previous chapter, I have proved the strong normalization of $\lambda^{\Box}$ and given a normalization algorithm explicitly. In this chapter, I scale $\lambda^{\Box}$ to Martin-Löf type theory (MLTT) (Martin-Löf, 1984), introducing MINT, a **M**odal **IN**tuitionistic **T**ype theory. I will first introduce the judgments for MINT and then establish its strong normalization by scaling the normalization proof in the previous chapter.

Despite the complexity induced by dependent types, many critical observations in fact have already been made in the previous chapter. They can be carried over to MINT with virtually no change, so I just need to focus on adapting these observations for the Kripke style to dependent types. As a result, the proof of the overall system becomes more manageable than working on MINT from scratch. These important observations include:

- the syntax of the explicit K-substitutions is unchanged;

- the exact definitions for truncoids (i.e. truncation and truncation offset) remain unchanged;
- the definitions for evaluation environments and UMoTs and their operations in the untyped domain model are unchanged;
- the semantic models for dependent types also need universal quantifications over UMoTs and respects UMoTs (i.e. monotonicity).

These observations have set a basis for dependent types. In this chapter, I incrementally build MINT on top of $\lambda^{\square}$.

The content in this chapter has been published (Hu et al., 2023) and mechanized.[12]

3.1 Introducing Mint by Examples

Before introducing MINT and its normalization by evaluation proof, let us consider some examples that exploit the □ modality in dependent types. In particular, I will use these programs to highlight different design decisions.

3.1.1 Laws in $S4$

Similar to $\lambda^{\square}$, MINT captures dependently typed variants of four different modal systems: K, T, $K4$ and $S4$. These systems are distinguished by the laws that they admit, as described in Table 2.1. With dependent types, for example, it is tempting to give the law `T` the following type in MINT:

```
T : {A : Ty} → □ A → A
```

Here `Ty` denotes universes to avoid clashing with Agda's terminology.[13] Unfortunately, this type does not type-check, because the type `A` is in the current world, but □ requires the type `A` to be meaningful in the next world. The correct implementation of `T` states that `A` is a type that is accessible in the next world by giving it the kind □ `Ty`. This ensures that `A` remains accessible. When using `A` in the definition of `T`, I now need to first `unbox` it with a proper level.

```
T : {A : □ Ty} → □ (unbox₁ A) → unbox₀ A
T x = unbox₀ x
```

[12] `https://hustmphrrr.github.io/mech-type-theories/README.html`

[13] Universe levels are an orthogonal issue, which is not discussed in this section.

It might appear counter-intuitive at first glance, why should MINT distinguish between a type of kind `□ Ty` and a type of kind `Ty`? This distinction is necessary, as MINT does not support cross-stage persistence (Taha and Sheard, 1997), i.e. the law $R : A \to \Box A$ or more specifically `Ty` $\to$ □`Ty`. In particular, there is no way to implement a function that would lift any type of kind `Ty` such that it would have kind □`Ty`. As a consequence, to ensure that all the types, in particular types such as $\Box A$ are well kinded, `A` needs to be meaningful in all Kripke worlds. A similar design decision is also taken by Jang et al. (2022) in their work on developing a polymorphic modal type system that supports the generation of polymorphic code.

The other two laws are implemented similarly:

```
K : {A B : □ Ty} → □ (unbox₁ A → unbox₁ B) →
    □ (unbox₁ A) → □ (unbox₁ B)
K f x = box ((unbox₁ f) (unbox₁ x))

A4 : {A : □ Ty} → □ (unbox₁ A) → □ □ (unbox₂ A)
A4 x = box (box (unbox₂ x))
```

3.1.2 Lifting of Natural Numbers

As previously discussed, MINT does not support cross-stage persistence and the law $A \to \Box A$ is not admissible for all A. Nevertheless, there are types which can be explicitly lifted from A to $\Box A$. The type for natural numbers is one such example. In MINT, such lifting functions needs to be implemented when required to ensure cross-stage persistence. Since MINT supports inductive types just as MLTT does, natural numbers, `Nat`, are defined in the usual way, with `zero` and `succ` as the constructors. Then the `lift` function shows that natural numbers do admit the law $A \to \Box A$:

```
lift : Nat → □ Nat
lift zero     = box zero
lift (succ n) = box (succ (unbox₁ (lift n)))
```

Note that this function is implemented by recursion on the input number. If the input is just `zero`, then the solution is easy: it is just `box zero`. The constructor `zero` can be referred inside of a `box`, which requires a term in the next world, because `Nat` is a closed definition, so `Nat` and its constructors can be lifted to any world. In the `succ` case, `box` enters a new world and the recursion `lift n` needs to be invoked somehow. Luckily, unbox₁ brings us back to the current world, where n is accessible and is precisely needed for recursion.

Just like MLTT, Mint can be used to prove properties about a definition. For example, the following definition proves that unbox₀ is an inverse of lift:

```
unbox-lift : (n : Nat) → unbox₀ (lift n) ≡ n
unbox-lift zero     = refl -- zero ≡ zero
unbox-lift (succ n) = cong succ (unbox-lift n)
```

In the base case, the left hand side evaluates to unbox₀ (box zero) which is just equivalent to zero. Therefore reflexivity (refl) suffices to prove this goal. In the step case, the proof obligation is

```
unbox₀ (box (succ (unbox₁ (lift n)))) ≡ succ n
```

The left hand side reduces to succ (unbox₀ (lift n)) based on the equivalence rules to be described in the next section. The recursive call gives unbox-lift n : unbox₀ (lift n) ≡ n. Therefore the goal is concluded by the recursive call modulo an extra congruence of succ.

3.1.3 Generating N-ary Sum

According to Davies and Pfenning (2001), the modal logic $S4$ corresponds to staged computation under Curry-Howard correspondence, where $\square A$ denotes the type of a computation of type A, the result of which is only available in some future stages of computation. Effectively, $\square$ segments different computational stages, so that variables in past stages cannot be directly referred to in the current stage. The $\square$ modality provides a logical foundation for multi-staged programming systems like MetaML (Taha and Sheard, 1997; Taha, 2000) and MetaOCaml (Kiselyov, 2014). By integrating $\square$ into MLTT, Mint can be viewed as a program logic to model dependently typed staged computations and use Mint's *equational theory* to prove that (meta-)programs satisfy certain specifications. In this section, I show how the $S4$ variant of Mint models staged programming and in the next, I prove that this meta-program is correctly implemented. Proving the correctness of a staged or meta-program in MetaML or a similar system has not been previously considered, but with Mint, this capability comes very naturally. For more practicality, certain extraction mechanisms can be implemented and employed here to extract the code to a mature staged programming system such as MetaML

(Taha and Sheard, 1997) or MetaOCaml (Kiselyov, 2014) with proper type-level magic to erase dependent types as commonly practiced in Coq and Agda.

The task in this example is to model a meta-program `nary-sum` that generates code for an n-ary sum function that sums up n numbers. If n is zero, then the result is `zero`; if it is one, then the result is the identity function; if it is two, then the result is a function that sums up two arguments, i.e. `box λ x y → x + y`. Writing such an n-ary sum function in a type-safe manner can be achieved by exploiting large elimination in MLTT.

In dependent type theory, large elimination allows recursions to compute types. It is an elimination into a "large" type,[14] hence the name. In this example, the type-level function `nary n` computes the type of an `n`-ary function via a large elimination:

```
nary : Nat → Ty
nary zero     = Nat
nary (succ n) = Nat → nary n
```

The type of `nary-sum` should intuitively take a natural number `n : Nat` and return code of type `nary n`. This, however, does not quite work, as `□ (nary n)` is ill-typed. Note that `n` is defined in the current world, but it is used inside `□` (i.e. in the next world). Hence, the `lift` function defined previously is used to first lift the `n : Nat` to `□ Nat` to be able to splice it in with $\texttt{unbox}_1$. The need to lift types such as natural numbers to have access to it in both current and future worlds is a common theme when writing staged programs. In a dependently typed setting, this pattern of lifting also occurs on the type level to support a form of cross-stage persistence of values. The correct type of `nary-sum` is hence `(n : Nat) → □ (nary (unbox₁ (lift n)))`. The implementation itself is in fact rather intuitive:

```
nary-sum : (n : Nat) → □ (nary (unbox₁ (lift n)))
nary-sum zero               = box zero
nary-sum (succ zero)        = box λ x → x
nary-sum (succ (succ n)) =
    box λ x y → (unbox₁ (nary-sum (succ n))) (x + y)
```

Note that in the base case of `zero`, the return type is `□ Nat`; in the case of `succ zero`, the result is a `box`ed identity function because the return type is `□ (Nat → Nat)`; in the

[14] as opposed to ordinary recursions which compute some data, i.e. "small" types

case of `succ (succ n)`, `nary-sum (succ (succ n))` returns a term of type

$$\square\ \texttt{(nary (unbox}_1\texttt{ (lift (succ (succ n)))))}$$

The recursive call `nary-sum (succ n)` has type $\square$ `(nary (unbox₁ (lift (succ n))))`. Further by computation

```
  nary (unbox₁ (lift (succ n)))
≈ Nat → nary (unbox₁ (lift n))
  nary (unbox₁ (lift (succ (succ n))))
≈ nary (succ (succ (unbox₁ (lift n))))
≈ Nat → Nat → nary (unbox₁ (lift n))
```

To compute the final result of `nary-sum (succ (succ n))`, I `unbox₁` the code generated by `nary-sum (succ n)`, which has type `Nat → nary (unbox₁ (lift n))`, and then apply this function to the sum of the first two arguments. To illustrate, let us normalize `nary-sum 3`. For convenience, numeric literals `0`, `1`, etc. and for natural numbers `zero`, `succ zero`, etc. are interchangeable.

```
nary-sum 1 ≈ box λ x1 → x1
nary-sum 2 ≈ box λ x2 x1 → (unbox₁ (nary-sum 1)) (x2 + x1)
           ≈ box λ x2 x1 → (λ x1 → x1) (x2 + x1)
           ≈ box λ x2 x1 → x2 + x1
nary-sum 3 ≈ box λ x3 x2 → (unbox₁ (nary-sum 2)) (x3 + x2)
           ≈ box λ x3 x2 → (λ x2 x1 → x2 + x1) (x3 + x2)
           ≈ box λ x3 x2 x1 → (x3 + x2) + x1
```

The last equation shows in Mint that `nary-sum 3` and the code of `λ x y z → (x + y) + z` are definitionally equal due to the congruence of `box`:

```
  nary-sum-3 : nary-sum 3 ≡ box λ x y z → (x + y) + z
  nary-sum-3 = refl
```

Mint admits the congruence of `box` and as a result, reductions occur freely even inside of a `box`. This behavior models the code optimization phases in meta-programming systems as in MetaML (Taha, 2000), so Mint serves as a program logic to reason about the behaviors of meta-programs. The congruence of `box` is essential to model MetaML and particularly helpful when using Mint as a program logic, for the same reason as having the congruence of λ. Moreover, the congruence of `box` allows a significantly simpler semantic model.

3.1.4 Soundness of N-ary Sum

Previously, `nary-sum-3` gives a specific soundness proof for the ternary sum. Mint can take one step further: it can be used to prove a general soundness theorem for `nary-sum`. Specifically, I prove that given a list `xs` of natural numbers which has length `n`, adding up all numbers in `xs` returns the same result as using the code generated by `nary-sum n` to add them up. To make this theorem precise, the following are the function `sum`, which sums up all the numbers in a list `xs` of length `n`, and the function `ap-list`, which applies a function `f : nary n` to all the numbers in `xs`:

```
sum : (n : Nat) (xs : List Nat) → length xs ≡ n → Nat
sum zero              []            refl = zero
sum (succ zero)       (x :: [])     refl = x
sum (succ (succ n)) (x :: y :: xs) eq   =
    sum (succ n) ((x + y) :: xs) omitted-eq

ap-list : (n : Nat) (xs : List Nat) → length xs ≡ n → nary n → Nat
ap-list zero      []        refl x = x
ap-list (succ n) (x :: xs) eq   f = ap-list n xs omitted-eq (f x)
```

where `omitted-eq` has type `length xs ≡ n` when `eq` has type `succ (length xs) ≡ succ n`. Equational reasoning is omitted in the example to avoid distraction. The slightly unorthodox definition of `sum` is defined by recursion on `n` just as `ap-list`, so that auxiliary lemmas such as the associativity of addition are avoided in the subsequent soundness theorem. Proving it equal to the standard definition is an easy exercise in MLTT, which is also omitted here. Then the final soundness theorem is stated as follows:

```
nary-sum-sound : (n : Nat) (xs : List N)
  (eq : length xs ≡ n) (eq' : length xs ≡ unbox₀ (lift n)) →
  ap-list (unbox₀ (lift n)) xs eq' (unbox₀ (nary-sum n)) ≡ sum n xs eq
nary-sum-sound zero               []             refl refl
    = refl -- zero ≡ zero
nary-sum-sound (succ zero)        (x :: [])      refl refl
    = refl -- x ≡ x
nary-sum-sound (succ (succ n)) (x :: y :: xs) eq   eq'
    = nary-sum-sound (succ n) ((x + y) :: xs) omitted-eq omitted-eq'
```

`nary-sum-sound` takes two equality proofs to simplify the formulation of this lemma. When using `nary-sum-sound`, `eq'` can be derived from `eq` and `unbox-lift` defined above.

The first two base cases are easy. In the last case, a recursive call suffices. The expected return type is

```
ap-list (succ (succ (unbox₀ (lift n)))) (x :: y :: xs) eq'
        (unbox₀ (nary-sum (succ (succ n))))
 ≡ sum (succ (succ n)) (x :: y :: xs) eq
```

Simplify the left hand side:

```
  ap-list (succ (succ (unbox₀ (lift n)))) (x :: y :: xs) eq'
          (unbox₀ (nary-sum (succ (succ n))))
≈ ap-list (unbox₀ (lift n)) xs omitted-eq'
          ((λ x y → unbox₀ (nary-sum (succ n)) (x + y)) x y)
≈ ap-list (unbox₀ (lift n)) xs omitted-eq'
          ((unbox₀ (nary-sum (succ n))) (x + y))
```

On the other hand, the recursive call gives:

```
ap-list (succ (unbox₀ (lift n))) ((x + y) :: xs) eq'
        (unbox₀ (nary-sum (succ n)))
≡  sum (succ n) ((x + y) :: xs) eq
```

Again simplify the left hand side:

```
  ap-list (succ (unbox₀ (lift n))) ((x + y) :: xs) omitted-eq'
          (unbox₀ (nary-sum (succ n)))
≈ ap-list (unbox₀ (lift n)) xs omitted-eq'
          ((unbox₀ (nary-sum (succ n))) (x + y))
```

Therefore by definitional equality, `nary-sum-sound` is a valid proof.

3.2 Definition of Mint

In the previous section, I have given a few example (meta-)programs to illustrate MINT informally. Starting this section, I give a formal account for MINT and assign semantics to it.

Mint has the following syntax:

$$\begin{array}{rcl}
i & & \text{(Universe Levels, } \mathbb{N}) \\
s, t, M, S, T & := & x \mid \mathtt{Nat} \mid \square T \mid \Pi(x : S).T \mid \mathtt{Ty}_i \mid \mathsf{zero} \mid \mathsf{succ}\ t \\
& & \mid \mathsf{elim}_{\mathtt{Nat}}\ (x.M)\ s\ (x, y.s')\ t \mid \mathtt{box}\ t \mid \mathtt{unbox}_n\ t \mid \lambda x.t \mid s\ t \mid t[\overrightarrow{\sigma}] \\
& & \text{(Terms, } \mathsf{Trm}) \\
w, W & := & u \mid \mathtt{Nat} \mid \mathtt{Ty}_i \mid \square W \mid \Pi(x : W).W' \mid \mathsf{zero} \mid \mathsf{succ}\ w \mid \mathtt{box}\ w \mid \lambda x.w \\
& & \text{(Normal forms, } \mathsf{Nf}) \\
u, V & := & x \mid \mathsf{elim}_{\mathtt{Nat}}\ (x.W)\ w\ (x, y.w')\ u \mid \mathtt{unbox}_n\ u \mid u\ w \\
& & \text{(Neutral forms, } \mathsf{Ne})
\end{array}$$

Mint combines $\lambda^{\square}$ introduced in Chapter 2 and Martin-Löf type theory. The same as $\lambda^{\square}$, Mint models the Kripke semantics and uses context stacks to keep track of assumptions in all accessed worlds. Mint has natural numbers ($\mathtt{Nat}$, zero, $\mathsf{succ}\ t$), Π types, cumulative universes, written as $\mathtt{Ty}_i$, and explicit K-substitutions. Here, the cumulativity of universes means if a type is in the universe of level i, then it is also in the universe of level $1 + i$, not a stronger notion of cumulativity based on universe subtyping. The elimination of natural numbers is performed by a recursive principle ($\mathsf{elim}_{\mathtt{Nat}}\ (x.M)\ s\ (x, y.s')\ t$). In this expression, t is the scrutinee describing a natural number, and s and s' are referring to the two possible cases where t is zero and the successor respectively; M is the motive describing essentially the overall type skeleton of the recursor. As the overall type of the recursor depends on t, the motive M depends on an open variable x of type $\mathtt{Nat}$. The handling of natural numbers is unsurprising and is almost identical to (Abel, 2013) so I choose to omit most of its discussion. Finally, λ abstractions are used to construct a term of Π type, which can then be used via function applications as usual. The role of $\square T$, its constructor `box` and its eliminator `unbox` is the same as Chapter 2, so I do not reiterate here again. The context and context stack structures are identical to $\lambda^{\square}$ defined in Sec. 2.1. Mint is also formulated as an explicit K-substitution calculus and the exact syntax is pleasantly unchanged (Sec. 2.3). All other important definitions including truncoids and the truncation and truncation offset operations of K-substitutions are carried over to Mint (see Sec. 2.4). This is very advantageous because their properties have been thoroughly studied in Chapter 2, so I will only need to focus on adapting these definitions to Mint.

Notationally, I consistently use upper cases for types and lower cases otherwise.

Some selected typing rules are given in Fig. 3.1 and 3.2. The full set of rules can be found in Appendix B. There are six different judgments in Mint:

- $\vdash \overrightarrow{\Gamma}$ denotes that the context stack $\overrightarrow{\Gamma}$ is well formed;
- $\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}$ denotes that $\overrightarrow{\Gamma}$ and $\overrightarrow{\Delta}$ are equivalent context stacks;
- $\overrightarrow{\Gamma} \vdash t : T$ denotes that term t has type T in context stack $\overrightarrow{\Gamma}$;
- $\overrightarrow{\Gamma} \vdash t \approx s : T$ denotes that terms t and s of type T are equivalent in context stack $\overrightarrow{\Gamma}$;
- $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$ denotes that $\overrightarrow{\sigma}$ is a K-substitution susbtituting terms in $\overrightarrow{\Delta}$ into ones in $\overrightarrow{\Gamma}$;
- $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\delta} : \overrightarrow{\Delta}$ denotes that $\overrightarrow{\sigma}$ and $\overrightarrow{\delta}$ are equivalent in K-substituting terms in $\overrightarrow{\Delta}$ into ones in $\overrightarrow{\Gamma}$.

Though the syntax of K-substitutions is identical, due to dependent types, the typing rules for K-substitutions must also keep track of the well-formedness of context stacks.

A basic syntactic validity check is done by proving two syntactic properties, *context stack conversion* and *presupposition*. Context stack conversion states that all syntactic judgments respect context stack equivalence $\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}$ to the left of the turnstile. Note that for judgments for K-substitutions like $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'$ and $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Gamma}'$, $\overrightarrow{\Gamma}'$ also respect context stack equivalence, but this property is built in to the rules of the judgments. Presupposition (or syntactic validity) includes for example facts such as if $\overrightarrow{\Gamma} \vdash t : T$ then $\vdash \overrightarrow{\Gamma}$ and $\overrightarrow{\Gamma} \vdash T : \texttt{Ty}_i$ for some i.

Theorem 3.1 (Context stack conversion)**.** *Given* $\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}$*,*

- *if* $\overrightarrow{\Gamma} \vdash t : T$*, then* $\overrightarrow{\Delta} \vdash t : T$*;*
- *if* $\overrightarrow{\Gamma} \vdash t \approx s : T$*, then* $\overrightarrow{\Delta} \vdash t \approx s : T$*;*
- *if* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'$*, then* $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'$*;*

$\boxed{\vdash \overrightarrow{\Gamma}}$ and $\boxed{\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}}$ well formedness of $\overrightarrow{\Gamma}$ and equivalence between $\overrightarrow{\Gamma}$ and $\overrightarrow{\Delta}$

$$\frac{}{\vdash \epsilon; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma}}{\vdash \overrightarrow{\Gamma}; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma}; \Gamma \qquad \overrightarrow{\Gamma}; \Gamma \vdash T : \mathtt{Ty}_i}{\vdash \overrightarrow{\Gamma}; \Gamma, x : T} \qquad \frac{}{\vdash \epsilon; \cdot \approx \epsilon; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}}{\vdash \overrightarrow{\Gamma}; \cdot \approx \overrightarrow{\Delta}; \cdot}$$

$$\frac{\begin{array}{c}\vdash \overrightarrow{\Gamma}; \Gamma \approx \overrightarrow{\Delta}; \Delta \qquad \overrightarrow{\Gamma}; \Gamma \vdash T \approx T' : \mathtt{Ty}_i \\ \overrightarrow{\Delta}; \Delta \vdash T \approx T' : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma}; \Gamma \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Delta}; \Delta \vdash T' : \mathtt{Ty}_i\end{array}}{\vdash \overrightarrow{\Gamma}; \Gamma, x : T \approx \overrightarrow{\Delta}; \Delta, x : T'}$$

$\boxed{\overrightarrow{\Gamma} \vdash t : T}$ t has type T in $\overrightarrow{\Gamma}$

$$\frac{\overrightarrow{\Gamma} \vdash T : \mathtt{Ty}_i}{\overrightarrow{\Gamma} \vdash T : \mathtt{Ty}_{1+i}} \qquad \frac{\overrightarrow{\Gamma}; \Gamma \vdash S : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma}; \Gamma, x : S \vdash T : \mathtt{Ty}_i}{\overrightarrow{\Gamma}; \Gamma \vdash \Pi(x : S).T : \mathtt{Ty}_i}$$

$$\frac{\overrightarrow{\Gamma}; \Gamma \vdash S : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma}; \Gamma, x : S \vdash t : T}{\overrightarrow{\Gamma}; \Gamma \vdash \lambda x : t : \Pi(x : S).T} \qquad \frac{\begin{array}{c}\overrightarrow{\Gamma}; \Gamma \vdash S : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma}; \Gamma, x : S \vdash T : \mathtt{Ty}_i \\ \overrightarrow{\Gamma}; \Gamma \vdash t : \Pi(x : S).T \qquad \overrightarrow{\Gamma}; \Gamma \vdash s : S\end{array}}{\overrightarrow{\Gamma}; \Gamma \vdash t\ s : T[\overrightarrow{I}, s]}$$

$$\frac{\overrightarrow{\Gamma}; \cdot \vdash T : \mathtt{Ty}_i}{\overrightarrow{\Gamma} \vdash \square T : \mathtt{Ty}_i} \qquad \frac{\overrightarrow{\Gamma}; \cdot \vdash t : T}{\overrightarrow{\Gamma} \vdash \mathtt{box}\ t : \square T} \qquad \frac{\begin{array}{c}\overrightarrow{\Gamma}; \cdot \vdash T : \mathtt{Ty}_i \\ \overrightarrow{\Gamma} \vdash t : \square T \qquad \vdash \overrightarrow{\Gamma}; \overrightarrow{\Delta} \qquad |\overrightarrow{\Delta}| = n\end{array}}{\overrightarrow{\Gamma}; \overrightarrow{\Delta} \vdash \mathtt{unbox}_n\ t : T[\overrightarrow{I}; \Uparrow^n]}$$

$$\frac{\overrightarrow{\Gamma} \vdash t : T \qquad \overrightarrow{\Gamma} \vdash T \approx T' : \mathtt{Ty}_i}{\overrightarrow{\Gamma} \vdash t : T'}$$

$\boxed{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}$ $\overrightarrow{\sigma}$ is a well formed K-substitution from $\overrightarrow{\Delta}$ to $\overrightarrow{\Gamma}$

$$\frac{\vdash \overrightarrow{\Gamma}}{\overrightarrow{\Gamma} \vdash \overrightarrow{I} : \overrightarrow{\Gamma}} \qquad \frac{\vdash \overrightarrow{\Gamma}; \Gamma, x : T}{\overrightarrow{\Gamma}; \Gamma, x : T \vdash \mathsf{wk} : \overrightarrow{\Gamma}; \Gamma} \qquad \frac{\begin{array}{c}\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'; \Gamma \\ \overrightarrow{\Gamma}'; \Gamma \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t : T[\overrightarrow{\sigma}]\end{array}}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma}, t : \overrightarrow{\Gamma}'; \Gamma, x : T}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \qquad \vdash \overrightarrow{\Gamma}; \overrightarrow{\Gamma}' \qquad |\overrightarrow{\Gamma}'| = n}{\overrightarrow{\Gamma}; \overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma}; \Uparrow^n : \overrightarrow{\Delta}; \cdot} \qquad \frac{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'' \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}'}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \circ \overrightarrow{\delta} : \overrightarrow{\Gamma}''}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \qquad \vdash \overrightarrow{\Delta} \approx \overrightarrow{\Delta}'}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}'}$$

Figure 3.1: Selected rules for Mint

$\boxed{\overrightarrow{\Gamma} \vdash t \approx t' : T}$ t and t' of type T are equivalent in $\overrightarrow{\Gamma}$

$$\frac{\overrightarrow{\Gamma}; \Gamma \vdash S : \texttt{Ty}_i \quad \overrightarrow{\Gamma}; \Gamma, x : S \vdash T : \texttt{Ty}_i \quad \overrightarrow{\Gamma}; \Gamma, x : S \vdash t : T \quad \overrightarrow{\Gamma}; \Gamma \vdash s : S}{\overrightarrow{\Gamma}; \Gamma \vdash (\lambda x.t)\ s \approx t[\overrightarrow{I}, s] : T[\overrightarrow{I}, s]}$$

$$\frac{\overrightarrow{\Gamma}; \Gamma \vdash S : \texttt{Ty}_i \quad \overrightarrow{\Gamma}; \Gamma, x : S \vdash T : \texttt{Ty}_i \quad \overrightarrow{\Gamma}; \Gamma \vdash t : \Pi(x : S).T}{\overrightarrow{\Gamma}; \Gamma \vdash t \approx \lambda x.(t[\mathsf{wk}]\ x) : \Pi(x : S).T}$$

$$\frac{\overrightarrow{\Gamma}; \cdot \vdash T : \texttt{Ty}_i \quad \overrightarrow{\Gamma}; \cdot \vdash t : T \quad \vdash \overrightarrow{\Gamma}; \overrightarrow{\Delta} \quad |\overrightarrow{\Delta}| = n}{\overrightarrow{\Gamma}; \overrightarrow{\Delta} \vdash \texttt{unbox}_n\ (\texttt{box}\ t) \approx t[\overrightarrow{I}; \Uparrow^n] : T[\overrightarrow{I}; \Uparrow^n]}$$

$$\frac{\overrightarrow{\Gamma}; \cdot \vdash T : \texttt{Ty}_i \quad \overrightarrow{\Gamma} \vdash t : \square T}{\overrightarrow{\Gamma} \vdash t \approx \texttt{box}\ (\texttt{unbox}_1\ t) : \square T}$$

Figure 3.2: Selected rules for Mint (Cont'd)

- *if* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Gamma}'$, *then* $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Gamma}'$.

Theorem 3.2 (Presupposition)**.**

- *If* $\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}$, *then* $\vdash \overrightarrow{\Gamma}$ *and* $\vdash \overrightarrow{\Delta}$.
- *If* $\overrightarrow{\Gamma} \vdash t : T$, *then* $\vdash \overrightarrow{\Gamma}$ *and* $\overrightarrow{\Gamma} \vdash T : \texttt{Ty}_i$ *for some* i.
- *If* $\overrightarrow{\Gamma} \vdash t \approx t' : T$, *then* $\vdash \overrightarrow{\Gamma}$, $\overrightarrow{\Gamma} \vdash t : T$, $\overrightarrow{\Gamma} \vdash t' : T$ *and* $\overrightarrow{\Gamma} \vdash T : \texttt{Ty}_i$ *for some* i.
- *If* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$, *then* $\vdash \overrightarrow{\Gamma}$ *and* $\vdash \overrightarrow{\Delta}$.
- *If* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}$, *then* $\vdash \overrightarrow{\Gamma}$, $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$, $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma}' : \overrightarrow{\Delta}$ *and* $\vdash \overrightarrow{\Delta}$.

3.3 Scaling Untyped Domain Model

In this section, I scale the untyped domain model described in Sec. 2.5 to Mint. The extension to the untyped domain model is moderate. There are still three kinds of values (D, D^{Ne} and D^{Nf}) and local and global evaluation environments. The definitions of domain values are adapted to Mint, but the definitions of environments and UMoTs remain the same. In fact, global environments and UMoTs possess identical truncoid structures to those described in Sec. 2.4. This is very convenient as most definitions are simply carried over from $\lambda^{\square}$ to Mint.

What is substantially changed in Mint is the PER model (see Sec. 3.5). Due to dependent types, the PER model must take type-level computation into account and therefore can no longer be defined by recursion on the structure of types. Instead, the semantics of types must be defined simultaneously with the semantics of terms, using *induction-recursion* (Dybjer, 2000; Dybjer and Setzer, 2001, 2003). Induction-recursion allows a mutual definition of multiple predicates, where one predicate is defined inductively and others are defined by recursions on this inductive predicate. In the PER model, the semantics of types are defined inductively and the semantics of terms are defined recursively. Once the PER model is established, I will prove completeness of NbE (Sec. 3.7).

Another complication due to dependent types, is the soundness proof (Sec. 3.10). Unlike $\lambda^{\square}$, the gluing model (Sec. 3.8) in the soundness proof relies on the PER model. The gluing model is defined by recursion on the PER model.

Now let us move on to the renewed definition of the untyped domain model:

$$\begin{aligned}
a, b, A &:= \ \mathsf{N} \mid \blacksquare A \mid Pi(A, T, \overrightarrow{\rho}) \mid \mathsf{U}_i \mid \texttt{ze} \mid \texttt{su}(a) \mid \Lambda(t, \overrightarrow{\rho}) \mid \texttt{box}(a) \mid \ \uparrow^A(c) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(Domain Values, } D\text{)} \\
c, C &:= \ z \mid \mathsf{rec}(M, a, t, c, \overrightarrow{\rho}) \mid c\ d \mid \texttt{unbox}(n, c) \quad \text{(Neutral Domain Values, } D^{\mathsf{Ne}}\text{)} \\
d, D &:= \ \downarrow^A(a) \qquad\qquad\qquad\qquad\qquad\qquad \text{(Normal Domain Values, } D^{\mathsf{Nf}}\text{)}
\end{aligned}$$

Same as before, I consistently use upper cases for semantic types and lower cases otherwise. In D, N models natural numbers, $\blacksquare$ models $\square$ semantically and U_i models a universe at level i. $\texttt{ze}$ models zero and $\texttt{su}(a)$ models successors. A neutral value c is embedded into D when annotated with a type A. The domain type $Pi(A, T, \overrightarrow{\rho})$ models a Π type and consists of a domain type A as the input type and a syntactic type T as the output type together with its ambient environment $\overrightarrow{\rho}$ which provides instantiations for all the free variables in T except the topmost one bound by Π in the syntax. This is another instance of defunctionalization first discussed in Sec. 2.5. Similarly, the domain value $\Lambda(t, \overrightarrow{\rho})$ models a λ term and describes a syntactic body t together with an environment $\overrightarrow{\rho}$ which provides values for all the free variables in t except the topmost one bound by λ in the syntax. For a neutral domain value for the recursor of natural numbers $\mathsf{rec}(M, a, t, c, \overrightarrow{\rho})$, the neutral domain value c describes the scrutinee, which is intended to be a natural number, while a describes the domain value for the base case. Next, t is an open syntactic term because it describes the term for the step case. Last, M describes the motive in the syntax with one free variable expressing the dependency on the natural number c. Due to defunctionalization, $\overrightarrow{\rho}$, the surrounding environment of M and t, is captured and stored.

The definition and the operations of evaluation environments are unchanged (see Sec. 2.5).

The updated untyped domain model also relies on UMoTs. The definition and the operations of UMoTs remain unchanged (see Sec. 2.6), so I do not repeat them here again. The application of a UMoT to domain values and environments is defined as follows:

$$\begin{array}{rcl}
\mathsf{N}[\kappa] & := & \mathsf{N} \\
\blacksquare A[\kappa] & := & \blacksquare(A[\kappa;\Uparrow^1]) \\
Pi(A, T, \overrightarrow{\rho})[\kappa] & := & Pi(A[\kappa], T, \overrightarrow{\rho}[\kappa]) \\
\mathsf{U}_i[\kappa] & := & \mathsf{U}_i \\
\mathtt{ze}[\kappa] & := & \mathtt{ze} \\
\mathtt{su}(a)[\kappa] & := & \mathtt{su}(a[\kappa]) \\
\Lambda(t, \overrightarrow{\rho})[\kappa] & := & \Lambda(t, \overrightarrow{\rho}[\kappa]) \\
\mathtt{box}(a)[\kappa] & := & \mathtt{box}(a[\kappa;\Uparrow^1]) \\
\uparrow^{A}(c)[\kappa] & := & \uparrow^{A[\kappa]}(c[\kappa]) \\
\\
z[\kappa] & := & z \\
\mathsf{rec}(M, a, t, c, \overrightarrow{\rho})[\kappa] & := & \mathsf{rec}(M, a[\kappa], t, c[\kappa], \overrightarrow{\rho}[\kappa]) \\
c\ d[\kappa] & := & (c[\kappa])\ (d[\kappa]) \\
\mathtt{unbox}(n, c)[\kappa] & := & \mathtt{unbox}(\mathcal{O}(\kappa, n), c[\kappa \mid n]) \\
\\
\downarrow^{A}(a)[\kappa] & := & \downarrow^{A[\kappa]}(a[\kappa]) \\
\\
\overrightarrow{\rho}[\kappa](0) & := & (\mathcal{O}(\kappa, n), \rho[\kappa]) \qquad (\text{where } (n, \rho) := \overrightarrow{\rho}(0)) \\
\overrightarrow{\rho}[\kappa](1+n) & := & \overrightarrow{\rho} \mid 1[\kappa \mid \mathcal{O}(\overrightarrow{\rho}, 1)](n)
\end{array}$$

The $\blacksquare$ case is similar to the `box` case, where A is applied to $\kappa;\Uparrow^1$, meaning that A is in a new world. Due to dependent types, κ is also applied to the domain type A in the cases of $\uparrow^A (c)$ and $\downarrow^A (a)$. The cases for evaluation environments are the same as those for $\lambda^{\square}$, which implies that the triple $(\mathsf{Envs}, \text{UMoT}, \mathsf{Envs})$ still forms an applicative truncoid. Other cases are identical to $\lambda^{\square}$ so I do not elaborate further here.

3.4 Evaluation and Readback

The change of the syntax and the domain also leads to an update to the evaluation and readback functions. Nevertheless, the update is moderate and takes dependent types into consideration. The same as $\lambda^{\square}$, the evaluation operation is responsible for eliminating all β redexes, so two auxiliary partial functions are defined to handle function applications and `unbox`'es. The new evaluation function is:

$$
\begin{aligned}
[\![\_]\!] &:: \mathsf{Trm} \rightharpoonup \mathsf{Envs} \to D \\
[\![\texttt{Nat}]\!](\overrightarrow{\rho}) &:= \mathsf{N} \\
[\![\texttt{Ty}_i]\!](\overrightarrow{\rho}) &:= \mathsf{U}_i \\
[\![\square T]\!](\overrightarrow{\rho}) &:= \blacksquare([\![T]\!](\texttt{ext}(\overrightarrow{\rho}))) \\
[\![\Pi(x : S).T]\!](\overrightarrow{\rho}) &:= Pi([\![S]\!](\overrightarrow{\rho}), T, \overrightarrow{\rho}) \\
[\![x]\!](\overrightarrow{\rho}) &:= \rho(x) \qquad \text{(where } (\_, \rho) := \overrightarrow{\rho}(0)) \\
[\![\mathsf{zero}]\!](\overrightarrow{\rho}) &:= \texttt{ze} \\
[\![\mathsf{succ}\ t]\!](\overrightarrow{\rho}) &:= \texttt{su}([\![t]\!](\overrightarrow{\rho})) \\
[\![\texttt{box}\ t]\!](\overrightarrow{\rho}) &:= \texttt{box}([\![t]\!](\texttt{ext}(\overrightarrow{\rho}))) \\
[\![\texttt{unbox}_n\ t]\!](\overrightarrow{\rho}) &:= \texttt{unbox} \cdot (\mathcal{O}(\overrightarrow{\rho}, n), [\![t]\!](\overrightarrow{\rho} \mid n)) \\
[\![\lambda x.t]\!](\overrightarrow{\rho}) &:= \Lambda(t, \overrightarrow{\rho}) \\
[\![t\ s]\!](\overrightarrow{\rho}) &:= [\![t]\!](\overrightarrow{\rho}) \cdot [\![s]\!](\overrightarrow{\rho}) \\
[\![t[\overrightarrow{\sigma}]]\!](\overrightarrow{\rho}) &:= [\![t]\!]([\![\overrightarrow{\sigma}]\!](\overrightarrow{\rho}))
\end{aligned}
$$

Only the evaluation function for terms is updated; the evaluation for K-substitutions remains identical to the one in Sec. 2.7. This again shows how the framework for a normalization proof has been robustly set up in $\lambda^{\square}$. The two auxiliary partial functions are:

$$
\begin{aligned}
\texttt{unbox}\cdot &:: \mathbb{N} \to D \rightharpoonup D \\
\texttt{unbox} \cdot (n, \texttt{box}(a)) &:= a[\overrightarrow{1}; \Uparrow^n] \\
\texttt{unbox} \cdot (n, \uparrow^{\blacksquare A} (c)) &:= \uparrow^{A[\overrightarrow{1}; \Uparrow^n]} (\texttt{unbox}(n, c))
\end{aligned}
$$

$$
\begin{aligned}
\_ \cdot \_ &:: D \rightharpoonup D \rightharpoonup D \\
(\Lambda(t, \overrightarrow{\rho})) \cdot a &:= [\![t]\!](\texttt{lext}(\overrightarrow{\rho}, a)) \\
(\uparrow^{Pi(A,T,\overrightarrow{\rho})} (c)) \cdot a &:= \uparrow^{[\![T]\!](\texttt{lext}(\overrightarrow{\rho}, a))} (c \ \downarrow^{A} (a))
\end{aligned}
$$

To take dependent types into account, the neutral cases in these two partial functions must also change the type annotations. For $\texttt{unbox}\cdot$, the type A also needs to be transformed into $A[\overrightarrow{1};\Uparrow^n]$ to match up with the β rule in Fig. 3.2. Similarly, when applying a neutral function, the result type is computed by $[\![T]\!](\texttt{lext}(\overrightarrow{\rho}, a))$. There is also an auxiliary function to handle recursions on natural numbers, which is omitted here for concision.

The readback functions should also be updated. The purpose of the readback functions is to convert the domain values to normal or neutral forms, and perform η expansions during the conversion. The readback process consists of three functions: R^{Nf} reads back a normal form; R^{Ne} reads back a neutral form; R^{Ty} reads back a normal type.

$$
\begin{aligned}
\mathsf{R}^{\mathsf{Nf}} &:: \overrightarrow{\mathbb{N}} \rightharpoonup D^{\mathsf{Nf}} \rightharpoonup \mathsf{Nf} \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\mathsf{U}_i}(A)) &:= \mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(A) \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\uparrow^{A}(c)}(\uparrow^{A'}(c'))) &:= \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c') \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\blacksquare A}(a)) &:= \texttt{box}\ \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z};0}(\downarrow^{A}(\texttt{unbox}\cdot(1,a))) \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z};z}(\downarrow^{Pi(A,T,\overrightarrow{\rho})}(a)) &:= \lambda x.\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z};1+z}(\downarrow^{[\![T]\!](\texttt{lext}(\overrightarrow{\rho},\uparrow^{A}(z)))}(a\cdot\uparrow^{A}(z))) \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\mathsf{N}}(\texttt{ze})) &:= \mathsf{zero} \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\mathsf{N}}(\texttt{su}(a))) &:= \mathsf{succ}\ (\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\mathsf{N}}(a))) \\
\mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(\downarrow^{\mathsf{N}}(\uparrow^{\mathsf{N}}(c))) &:= \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c) \\
\mathsf{R}^{\mathsf{Ty}} &:: \overrightarrow{\mathbb{N}} \rightharpoonup D \rightharpoonup \mathsf{Nf} \\
\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(\mathsf{U}_i) &:= \texttt{Ty}_i \\
\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(\blacksquare A) &:= \square \mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z};0}(A) \\
\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z};z}(Pi(A,T,\overrightarrow{\rho})) &:= \Pi(x : \mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z};z}(A)).\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z};1+z}([\![T]\!](\texttt{lext}(\overrightarrow{\rho},\uparrow^{A}(z)))) \\
\mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(\uparrow^{A}(c)) &:= \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c) \\
\mathsf{R}^{\mathsf{Ne}} &:: \overrightarrow{\mathbb{N}} \rightharpoonup D^{\mathsf{Ne}} \rightharpoonup \mathsf{Ne} \\
\mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z};z'}(z) &:= x \quad \text{(the de Bruijn index of } x \text{ is } \max(z'-z-1,0)) \\
\mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c\ d) &:= \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c)\ \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(d) \\
\mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(\texttt{unbox}(n,c)) &:= \texttt{unbox}_n\ \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}|n}(c)
\end{aligned}
$$

With the updated definitions of evaluation and readback, the definition of the NbE algorithm is also slighted updated due to dependent types. The difference is in the type annotation; the type T must also be evaluated.

Definition 3.1. For $\overrightarrow{\Gamma} \vdash t : T$, the NbE algorithm is

$$\mathsf{nbe}^{T}_{\overrightarrow{\Gamma}}(t) := \mathsf{R}^{\mathsf{Nf}}_{\mathtt{map}(\Gamma \mapsto |\Gamma|, \overrightarrow{\Gamma})}(\downarrow^{[\![T]\!](\uparrow^{\overrightarrow{\Gamma}})}([\![t]\!](\uparrow^{\overrightarrow{\Gamma}})))$$

where the initial environment $\uparrow^{\overrightarrow{\Gamma}}$ is defined by the structure of $\overrightarrow{\Gamma}$:

$$\begin{aligned}
\uparrow &:: \overrightarrow{\mathsf{Ctx}} \rightharpoonup \mathsf{Envs} \\
\uparrow^{\epsilon;\cdot} &:= \mathsf{empty} \\
\uparrow^{\overrightarrow{\Gamma};\cdot} &:= \mathtt{ext}(\uparrow^{\overrightarrow{\Gamma}}) \\
\uparrow^{\overrightarrow{\Gamma};(\Gamma.T)} &:= \mathtt{lext}(\overrightarrow{\rho}, \uparrow^{[\![T]\!](\overrightarrow{\rho})}(l_{|\Gamma|})) \qquad \text{(where } \overrightarrow{\rho} := \uparrow^{\overrightarrow{\Gamma};\Gamma}\text{)}
\end{aligned}$$

3.5 PER Model

In this section, I define the PER model for the untyped domain model. The change to the PER model is more substantial. In fact, it is completely redefined due to dependent types, because the semantics of terms can no longer be defined by recursion on the structure of types. Instead, induction-recursion (Dybjer, 2000; Dybjer and Setzer, 2001, 2003) is used to give the semantics of types and terms simultaneously. Similar to other NbE proofs for dependent types (Abel, 2013; Abel et al., 2017; Gratzer et al., 2019; Altenkirch and Kaposi, 2017; Wieczorek and Biernacki, 2018), the soundness proof of NbE relies on the fundamental theorem of the PER model, so the PER model is a prerequisite for Sec. 3.8.

Following Abel (2013), I first introduce the following relations.

$$\frac{\forall\ \overrightarrow{z}, \kappa\ .\ \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(d[\kappa]) = \mathsf{R}^{\mathsf{Nf}}_{\overrightarrow{z}}(d'[\kappa])}{d \approx d' \in \mathit{Nf}} \qquad \frac{\forall\ \overrightarrow{z}, \kappa\ .\ \mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(A[\kappa]) = \mathsf{R}^{\mathsf{Ty}}_{\overrightarrow{z}}(A'[\kappa])}{A \approx A' \in \mathit{Ty}}$$

$$\frac{\forall\ \overrightarrow{z}, \kappa\ .\ \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c[\kappa]) = \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c'[\kappa])}{c \approx c' \in \mathit{Ne}}$$

where $\mathit{Nf} \subseteq D^{\mathsf{Nf}} \times D^{\mathsf{Nf}}$, $\mathit{Ty} \subseteq D \times D$ and $\mathit{Ne} \subseteq D^{\mathsf{Ne}} \times D^{\mathsf{Ne}}$. *Nf* relates two normal domain values iff their readbacks are equal given any context stack and UMoT. *Ty* and *Ne* are defined similarly.

Now I define the actual PERs that relate domain values. The PER model resembles Tarski-style universes (Palmgren, 1998), in which universes contain "codes" and **El** converts these codes into actual types which contain values. The PER model consists of two PERs: $\mathcal{U}_i$, which denotes a relation for types and relates two domain types at level i, and $\mathbf{El}_i(\mathcal{D})$, which given $\mathcal{D} :: A \approx B \in \mathcal{U}_i$ relates two domain values of domain types A and B. Following Abel (2013); Abel et al. (2018), $\mathcal{U}_i$ and $\mathbf{El}_i$ are defined inductive-recursively. Moreover, due to cumulative universes, they must in addition be defined with the well-foundedness of the universe levels.

Definition 3.2. The equivalence for domain types $\mathcal{D} :: A \approx B \in \mathcal{U}_i$ is defined inductively and the equivalence for domain values $a \approx b \in \mathbf{El}_i(\mathcal{D})$ is defined by recursion on $\mathcal{D}$ as follows:

- Neutral types and neutral values:

$$\mathcal{D} := \frac{C \approx C' \in Ne}{\uparrow^{A} (C) \approx \uparrow^{A'} (C') \in \mathcal{U}_i}$$

  Then $a \approx b \in \mathbf{El}_i(\mathcal{D})$ iff $a \approx b \in Neu$, where Neu relates two values only when they are actually neutral:

$$\frac{c \approx c' \in Ne}{\uparrow^{A_1} (c) \approx \uparrow^{A_2} (c') \in Neu}$$

  Note that the annotating domain types A_1 and A_2 do not matter as long as c and c' are related by Ne.

- Natural numbers:

$$\mathcal{D} := \frac{}{\mathsf{N} \approx \mathsf{N} \in \mathcal{U}_i}$$

  Then $a \approx b \in \mathbf{El}_i(\mathcal{D})$ iff $a \approx b \in Nat$, where Nat inductively relates two domain values that are natural numbers:

$$\frac{}{\texttt{ze} \approx \texttt{ze} \in \mathit{Nat}} \qquad \frac{a \approx b \in \mathit{Nat}}{\texttt{su}(a) \approx \texttt{su}(b) \in \mathit{Nat}} \qquad \frac{c \approx c' \in \mathit{Ne}}{\uparrow^{\mathsf{N}} (c) \approx \uparrow^{\mathsf{N}} (c') \in \mathit{Nat}}$$

- Universes:

$$\mathcal{D} := \frac{j < i}{\mathsf{U}_j \approx \mathsf{U}_j \in \mathcal{U}_i}$$

Then $a \approx b \in \mathbf{El}_i(\mathcal{D})$ iff $a \approx b \in \mathcal{U}_j$. Note that here $\mathbf{El}_i(\mathcal{D})$ is defined in terms of $\mathcal{U}_j$. This is fine because of $j < i$ and the well-foundedness of universe levels.

- Semantic $\blacksquare$ types:

$$\mathcal{D} := \frac{\mathcal{J} :: \forall\ \kappa\ .\ A[\kappa] \approx A'[\kappa] \in \mathcal{U}_i}{\blacksquare A \approx \blacksquare A' \in \mathcal{U}_i}$$

Then $a \approx b \in \mathbf{El}_i(\mathcal{D})$ iff for any UMoT κ and `unbox` level n, the `unbox`ing's of $a[\kappa]$ and $b[\kappa]$ remain related: $\texttt{unbox} \cdot (n, a[\kappa]) \approx \texttt{unbox} \cdot (n, b[\kappa]) \in \mathbf{El}_i(\mathcal{J}(\kappa; \Uparrow^n))$. In other words, if a and b are still related no matter how they travel in Kripke worlds and then are `unbox`'ed, then they are related by $\mathbf{El}_i(\mathcal{D})$.

- Semantic Pi types:

$$\mathcal{D} := \frac{\begin{array}{c}\mathcal{J}_1 :: \forall\ \kappa\ .\ A[\kappa] \approx A'[\kappa] \in \mathcal{U}_i \\ \mathcal{J}_2 :: \forall\ \kappa, a \approx a' \in \mathbf{El}_i(\mathcal{J}_1[\kappa])\ .\ [\![T]\!](\texttt{lext}(\overrightarrow{\rho}[\kappa], a)) \approx [\![T']\!](\texttt{lext}(\overrightarrow{\rho}'[\kappa], a')) \in \mathcal{U}_i\end{array}}{\mathit{Pi}(A, T, \overrightarrow{\rho}) \approx \mathit{Pi}(A', T', \overrightarrow{\rho}') \in \mathcal{U}_i}$$

Then $a \approx b \in \mathbf{El}_i(\mathcal{D})$ iff for any UMoT κ and related a' and b', i.e. $\mathcal{E} :: a' \approx b' \in \mathbf{El}_i(\mathcal{J}_1(\kappa))$, the results of applying $a[\kappa]$ and $b[\kappa]$ remain related: $a[\kappa] \cdot a' \approx b[\kappa] \cdot b' \in \mathbf{El}_i(\mathcal{J}_2(\kappa, \mathcal{E}))$. That is, a and b are related if all results of applying them in other worlds to related values are still related.

3.6 Properties for PERs

Due to the complication of the inductive-recursive definition of the PER model, the statements and the proofs of necessary properties are no longer as straightforward as in $\lambda^{\square}$. In this section, I discuss a number of properties which are made precise and adjusted to a type-theoretic flavor during mechanization. Contrasts are drawn between this mechanized work of Mint and other on-paper, set-theoretic NbE proofs (Abel, 2013; Abel et al., 2017; Gratzer et al., 2019).

3.6.1 $\mathcal{U}$ Irrelevance

While it comes for free in paper proofs, in type theory, it requires a proof that $\mathbf{El}_i$ only relies on A and B in $A \approx B \in \mathcal{U}_i$, not how exactly they are related by $\mathcal{U}_i$. Effectively, $\mathcal{U}$ is proof-irrelevant:

Lemma 3.3 ($\mathcal{U}$ irrelevance). *Given $\mathcal{D} :: A \approx B \in \mathcal{U}_i$ and $a \approx b \in \boldsymbol{El}_i(\mathcal{D})$,*

- *if $\mathcal{E}_1 :: A \approx B' \in \mathcal{U}_i$, then $a \approx b \in \boldsymbol{El}_i(\mathcal{E}_1)$;*
- *if $\mathcal{E}_2 :: A' \approx B \in \mathcal{U}_i$, then $a \approx b \in \boldsymbol{El}_i(\mathcal{E}_2)$.*

Proof. Do an induction on $\mathcal{D}$ and invert $\mathcal{E}_1$ and $\mathcal{E}_2$. □

This lemma matches the set-theoretic intuition, which states that a and b are related as long as we know they are related by one "representative" domain type.

3.6.2 $\mathcal{U}$ and El are PERs

Though the PER model has been called a "PER", this fact requires a proof. During the proof, however, the strong scrutiny from Agda's termination checker requires adjustments to the statements so that they are more type-theoretic. For example, the statement of symmetry includes an extra premise $\mathcal{D}_2$. This extra assumption $\mathcal{D}_2$ exposes a clearer termination measure and allows Agda to admit the proof just by recognizing decreasing structures.

Lemma 3.4 (Symmetry). *Given $\mathcal{D}_1 :: A \approx B \in \mathcal{U}_i$,*

- *$B \approx A \in \mathcal{U}_i$;*

- *if* $\mathcal{D}_2 :: B \approx A \in \mathcal{U}_i$*, and* $a \approx b \in \textbf{\textit{El}}_i(\mathcal{D}_1)$*, then* $b \approx a \in \textbf{\textit{El}}_i(\mathcal{D}_2)$*.*

Proof. Induction on i and $\mathcal{D}_1$ and inversion on $\mathcal{D}_2$ in the second statement. □

The symmetry of **El** requires two $\mathcal{U}_i$ derivations: $\mathcal{D}_1$ relating A and B, and $\mathcal{D}_2$ relating B and A. They are used in the premise and the conclusion respectively. $\mathcal{D}_2$ is important to handle the contravariance of the input and the output in the function case. $\mathcal{D}_2$ can be eventually discharged by combining the symmetry of $\mathcal{U}_i$ and irrelevance, but it is necessary to prove the lemma in a type-theoretic flavor.

Transitivity also requires a similar treatment but more complex:

Lemma 3.5 (Transitivity)**.** *Given* $\mathcal{D}_1 :: A_1 \approx A_2 \in \mathcal{U}_i$ *and* $\mathcal{D}_2 :: A_2 \approx A_3 \in \mathcal{U}_i$*,*

- $A_1 \approx A_3 \in \mathcal{U}_i$*;*
- *if* $\mathcal{D}_3 :: A_1 \approx A_3 \in \mathcal{U}_i$*,* $A_1 \approx A_1 \in \mathcal{U}_i$*, and* $a_1 \approx a_2 \in \textbf{\textit{El}}_i(\mathcal{D}_1)$ *and* $a_2 \approx a_3 \in \textbf{\textit{El}}_i(\mathcal{D}_2)$*, then* $a_1 \approx a_3 \in \textbf{\textit{El}}_i(\mathcal{D}_3)$*.*

In addition to $\mathcal{D}_3$ which is used in $\mathbf{El}_i(\mathcal{D}_3)$ in conclusion, reflexivity of A_1, $A_1 \approx A_1 \in \mathcal{U}_i$, is also a required assumption. This is again due to the function case. Proving reflexivity requires transitivity, so during the proof of transitivity, it is easier to make $A_1 \approx A_1 \in \mathcal{U}_i$ an extra assumption.

3.6.3 Monotonicity

The same as $\lambda^{\square}$, the PER model must respect UMoTs, i.e. is monotonic. Monotonicity ensures that the Kripke structure is successfully internalized, so subsequent proofs are unaware of the exact modality, making the whole NbE proof structure very general.

Lemma 3.6 (Monotonicity)**.** *Given* $\mathcal{D} :: A \approx B \in \mathcal{U}_i$ *and a UMoT* κ*,*

- $A[\kappa] \approx B[\kappa] \in \mathcal{U}_i$*;*
- *if* $\mathcal{E} :: A[\kappa] \approx B[\kappa] \in \mathcal{U}_i$ *and* $a \approx b \in \textbf{\textit{El}}_i(\mathcal{D})$*, then* $a[\kappa] \approx b[\kappa] \in \textbf{\textit{El}}_i(\mathcal{E})$*.*

Similar to symmetry and transitivity, $\mathcal{E}$ is required in the second statement to ease termination checking in Agda.

3.6.4 Cumulativity and Lowering

Cumulativity states that if two types or values are related at level i, then they are also related at level $1+i$. At first glance, this property is too intuitive to be worth looking into. However, in the function case, the conclusion assumes two related values at level $1+i$, but the premise requires them to be related at level i, so there are some missing pieces. The *lowering* lemma fixes the issue and is mutually proved with cumulativity:

Lemma 3.7 (Cumulativity and lowering)**.** *If $\mathcal{D} :: A \approx B \in \mathcal{U}_i$,*

- *then $\mathcal{D}' :: A \approx B \in \mathcal{U}_{i+1}$;*
- *if $a \approx b \in \boldsymbol{El}_i(\mathcal{D})$, then $a \approx b \in \boldsymbol{El}_{i+1}(\mathcal{D}')$;*
- *if $a \approx b \in \boldsymbol{El}_{i+1}(\mathcal{D}')$, then $a \approx b \in \boldsymbol{El}_i(\mathcal{D})$.*

Note that according to the last statement, related values are lowered from $\mathbf{El}_{1+i}$ to $\mathbf{El}_i$ if A and B are related at level i. In general, lowering is necessary for type constructors in which types can occur in contra-variant positions.

3.6.5 Realizability

Following Abel (2013) and Chapter 2, realizability is an important property and states that related values are read back equal. Realizability of the PER model is essential to establish the completeness and the soundness theorems. More formally,

Theorem 3.8 (Realizability)**.** *Given $\mathcal{D} ::\ A \approx B \in \mathcal{U}_i$,*

- *$A \approx B \in Ty$;*
- *if $c \approx c' \in Ne$, then $\uparrow^A (c) \approx \uparrow^B (c') \in \boldsymbol{El}_i(\mathcal{D})$;*
- *if $a \approx b \in \boldsymbol{El}_i(\mathcal{D})$, then $\downarrow^A (a) \approx \downarrow^B (b) \in Nf$.*

Proof. Induction on i and $\mathcal{D}$. □

The first statement says that if A and B are related, then they are always read back as an equal normal type in syntax. The third statement says that if a and b are related, then they are always read back as an equal normal form in syntax.

3.7 Semantic Judgments And Completeness

The PER model for context stacks and environments is required to define the semantic judgments for the fundamental theorem and completeness. It is extended from the PER model for domain types and values.

Definition 3.3. The equivalence of context stacks $\mathcal{D} :: \vDash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}$ and the equivalence of evaluation environments $\overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\mathcal{D}]\!]$ are defined inductive-recursively as follows:

- $$\mathcal{D} := \frac{}{\vDash \epsilon; \cdot \approx \epsilon; \cdot}$$

  Then $\overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\mathcal{D}]\!]$ is always true.

- $$\mathcal{D} := \frac{\mathcal{J} :: \vDash \overrightarrow{\Gamma} \approx \overrightarrow{\Gamma}'}{\vDash \overrightarrow{\Gamma}; \cdot \approx \overrightarrow{\Gamma}'; \cdot}$$

  Then $\overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\mathcal{D}]\!]$ iff

  – truncations of $\overrightarrow{\rho}$ and $\overrightarrow{\rho}'$ by one are recursively related:

  $$\overrightarrow{\rho} \mid 1 \approx \overrightarrow{\rho}' \mid 1 \in [\![\mathcal{J}]\!]$$

  and

  – first modal offsets are equal: $n = n'$ where $(n, \_) := \overrightarrow{\rho}(0)$ and $(n', \_) := \overrightarrow{\rho}'(0)$.

- $$\mathcal{D} := \frac{\mathcal{J}_1 :: \vDash \overrightarrow{\Gamma}; \Gamma \approx \overrightarrow{\Gamma}'; \Gamma' \qquad \mathcal{J}_2 :: \forall\ \overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\mathcal{J}_1]\!]\ .\ [\![T]\!](\overrightarrow{\rho}) \approx [\![T']\!](\overrightarrow{\rho}') \in \mathcal{U}_i}{\vDash \overrightarrow{\Gamma}; \Gamma.T \approx \overrightarrow{\Gamma}'; \Gamma'.T}$$

Then $\overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\mathcal{D}]\!]$ iff

– $\texttt{drop}(\overrightarrow{\rho})$ and $\texttt{drop}(\overrightarrow{\rho}')$ are recursively related:

$$\mathcal{E} :: \texttt{drop}(\overrightarrow{\rho}) \approx \texttt{drop}(\overrightarrow{\rho}') \in [\![\mathcal{J}_1]\!]$$

where $\texttt{drop}\_$ is defined in Sec. 2.5 and drops the topmost mapping from the topmost Env, and

– the topmost values are related by the evaluations of T:

$$\rho(0) \approx \rho'(0) \in \mathbf{El}_i(\mathcal{J}_2(\mathcal{E}))$$

where $(\_, \rho) := \overrightarrow{\rho}(0)$ and $(\_, \rho') := \overrightarrow{\rho}'(0)$.

Definition 3.4. The semantic judgments for completeness are defined as follows:

$$\frac{\vDash \overrightarrow{\Gamma} \approx \overrightarrow{\Gamma}}{\vDash \overrightarrow{\Gamma}} \qquad \frac{\overrightarrow{\Gamma} \vDash t \approx t : T}{\overrightarrow{\Gamma} \vDash t : T} \qquad \frac{\overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} \approx \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} : \overrightarrow{\Delta}}$$

where $\overrightarrow{\Gamma} \vDash t \approx t' : T$ iff

- $\overrightarrow{\Gamma}$ is semantically well formed: $\mathcal{D} :: \vDash \overrightarrow{\Gamma}$, and
- there exists a universe level i, such that for any related $\overrightarrow{\rho}$ and $\overrightarrow{\rho}'$, i.e. $\mathcal{E} :: \overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\mathcal{D}]\!]$,
  – evaluations of T are related types: $\mathcal{J} :: [\![T]\!](\overrightarrow{\rho}) \approx [\![T]\!](\overrightarrow{\rho}') \in \mathcal{U}_i$, and
  – evaluations of t and t' are related values: $[\![t]\!](\overrightarrow{\rho}) \approx [\![t']\!](\overrightarrow{\rho}') \in \mathbf{El}_i(\mathcal{J})$;

and $\overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}$ iff

- $\overrightarrow{\Gamma}$ and $\overrightarrow{\Delta}$ are semantically well formed: $\mathcal{D} :: \vDash \overrightarrow{\Gamma}$ and $\mathcal{E} :: \vDash \overrightarrow{\Delta}$, and
- for any related $\overrightarrow{\rho}$ and $\overrightarrow{\rho}'$, i.e. $\overrightarrow{\rho} \approx \overrightarrow{\rho}' \in [\![\mathcal{D}]\!]$, $\overrightarrow{\sigma}$ and $\overrightarrow{\sigma}'$ evaluate to related environments: $[\![\overrightarrow{\sigma}]\!](\overrightarrow{\rho}) \approx [\![\overrightarrow{\sigma}']\!](\overrightarrow{\rho}') \in [\![\mathcal{E}]\!]$.

The fundamental theorem states that the semantic judgments are sound w.r.t. the syntactic judgments:

Theorem 3.9 (Fundamental)**.**

- *If* $\vdash \overrightarrow{\Gamma}$*, then* $\vDash \overrightarrow{\Gamma}$*.*
- *If* $\overrightarrow{\Gamma} \vdash t : T$*, then* $\overrightarrow{\Gamma} \vDash t : T$*.*
- *If* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$*, then* $\overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} : \overrightarrow{\Delta}$*.*
- *If* $\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Gamma}'$*, then* $\vDash \overrightarrow{\Gamma} \approx \overrightarrow{\Gamma}'$*.*
- *If* $\overrightarrow{\Gamma} \vdash t \approx t' : T$*, then* $\overrightarrow{\Gamma} \vDash t \approx t' : T$*.*
- *If* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}$*, then* $\overrightarrow{\Gamma} \vDash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}$*.*

Proof. Proceed by a mutual induction on premises. □

The completeness theorem states that syntactic equivalent terms evaluate to equal normal forms:

Theorem 3.10 (Completeness)**.** *If* $\overrightarrow{\Gamma} \vdash t \approx t' : T$*, then* $\mathsf{nbe}^{T}_{\overrightarrow{\Gamma}}(t) = \mathsf{nbe}^{T}_{\overrightarrow{\Gamma}}(t')$*.*

Proof. The fundamental theorem implies $\overrightarrow{\Gamma} \vDash t \approx t' : T$ and also $\mathcal{D} :: \vDash \overrightarrow{\Gamma}$. Moreover, the initial environment is related: $\uparrow^{\overrightarrow{\Gamma}} \approx \uparrow^{\overrightarrow{\Gamma}} \in [\![\mathcal{D}]\!]$. By definition of the semantic judgment, $[\![t]\!](\uparrow^{\overrightarrow{\Gamma}})$ and $[\![t']\!](\uparrow^{\overrightarrow{\Gamma}})$ are related, and due to realizability, they have equal normal forms. □

The fundamental theorem of the PER model implies a few consequences which are useful but very challenging to prove syntactically:

Lemma 3.11 (Equal universe levels)**.** *If* $\overrightarrow{\Gamma} \vdash \mathtt{Ty}_i \approx \mathtt{Ty}_j : \mathtt{Ty}_k$*, then* $i = j$*.*

Proof. Completeness says that $\mathtt{Ty}_i$ and $\mathtt{Ty}_j$ have equal normal form, which implies $i = j$. □

Due to the previous lemma, the following "exact inversion" lemma is:

Lemma 3.12 (Exact inversion)**.**

- *If* $\overrightarrow{\Gamma} \vdash \square T : \boldsymbol{Ty}_i$, *then* $\overrightarrow{\Gamma}; \cdot \vdash T : \boldsymbol{Ty}_i$.
- *If* $\overrightarrow{\Gamma}; \Gamma \vdash \Pi(x : S).T : \boldsymbol{Ty}_i$, *then* $\overrightarrow{\Gamma}; \Gamma \vdash S : \boldsymbol{Ty}_i$ *and* $\overrightarrow{\Gamma}; \Gamma, x : S \vdash T : \boldsymbol{Ty}_i$.

A mere induction on the syntactic derivations can only conclude larger universe levels. The universe levels are tightened due to the fundamental theorem.

3.8 Gluing Model

After proving the completeness theorem, I then move on to the soundness theorem. Following Chapter 2, the soundness theorem would require a Kripke gluing model to relate domain values and syntactic terms. The gluing model is Kripke because it respects restricted weakenings. It turns out that the definitions of restricted weakenings (Sec. 2.10) and the UMoT extraction operation (Sec. 2.11) from $\lambda^{\square}$ are carried over unchanged to Mint. Due to dependent types, the situation here is more complex because the gluing model is defined by recursion on the PER model.

Following $\lambda^{\square}$ in Chapter 2, I first give three definitions which serve a purpose similar to *Ty*, *Ne* and *Nf* in completeness. Recall that for a domain value a, I write $a[\overrightarrow{\sigma}]$ for $a[\mathtt{mt}(\overrightarrow{\sigma})]$, where $\mathtt{mt}(\_)$ converts a K-substitution to a UMoT and is defined in Sec. 2.11. For a PER P, we write $p \in P$ for $p \approx p \in P$. Next, I define three necessary judgments for the realizability theorem (c.f. Theorem 3.14), with similar roles to $\underline{A}_{\overrightarrow{\Gamma}}$ and $\overline{A}_{\overrightarrow{\Gamma}}$ in Sec. 2.11:

Definition 3.5. Given $\mathcal{D} :: A \approx B \in \mathcal{U}_i$,

$$\frac{A \approx B \in \mathit{Ty} \qquad \begin{array}{c}\overrightarrow{\Gamma} \vdash T : \mathtt{Ty}_i \\ \forall\ \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}\ .\ \overrightarrow{\Delta} \vdash T[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Ty}}_{\mathtt{map}(\Delta \mapsto |\Delta|, \overrightarrow{\Delta})}(A[\overrightarrow{\sigma}]) : \mathtt{Ty}_i\end{array}}{\overrightarrow{\Gamma} \vdash T\ \overline{\circledR}_i\ \mathcal{D}}$$

$$\frac{\overrightarrow{\Gamma} \vdash T\ \circledR_i\ \mathcal{D} \qquad c \in \mathit{Ne} \qquad \begin{array}{c}\overrightarrow{\Gamma} \vdash t : T \\ \forall\ \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}\ .\ \overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\mathtt{map}(\Delta \mapsto |\Delta|, \overrightarrow{\Delta})}(c[\overrightarrow{\sigma}]) : T[\overrightarrow{\sigma}]\end{array}}{\overrightarrow{\Gamma} \vdash t : T\ \underline{\circledR}_i\ c \in \mathbf{El}_i(\mathcal{D})}$$

$$\frac{\begin{array}{c}\overrightarrow{\Gamma} \vdash t : T \qquad \overrightarrow{\Gamma} \vdash T\ \circledR_i\ \mathcal{D} \qquad \downarrow^{A}(a) \approx \downarrow^{B}(a) \in \mathit{Nf} \\ \forall\ \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}\ .\ \overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Nf}}_{\mathtt{map}(\Delta \mapsto |\Delta|, \overrightarrow{\Delta})}(\downarrow^{A}(a)[\overrightarrow{\sigma}]) : T[\overrightarrow{\sigma}]\end{array}}{\overrightarrow{\Gamma} \vdash t : T\ \overline{\circledR}_i\ a \in \mathbf{El}_i(\mathcal{D})}$$

The most important conditions of all three judgments are the last universal quantifications. The first judgment $\overrightarrow{\Gamma} \vdash T \ \overline{®}_i \ \mathcal{D}$ states that T is equivalent to the readback of A (or equally the readback of B) under all valid restricted weakenings. Similarly, the third judgment $\overrightarrow{\Gamma} \vdash t : T \ \overline{®}_i \ a \in \mathbf{El}_i(\mathcal{D})$ states that t is equivalent to the readback of a under all valid restricted weakenings. This condition is particularly important, as it will be instantiated to obtain the soundness theorem.

Then I move on to defining the gluing model. The gluing model includes two judgments and is defined by recursion on the PER model $\mathcal{D} :: A \approx B \in \mathcal{U}_i$. By recursion on $\mathcal{D}$, $\overrightarrow{\Gamma} \vdash T \ ®_i \ \mathcal{D}$ defines the conditions for T to be eventually syntactically equivalent to the readbacks of A and B; and $\overrightarrow{\Gamma} \vdash t : T \ ®_i \ a \in \mathbf{El}_i(\mathcal{D})$ defines the conditions for $t : T$ to be syntactically equivalent to the readback of a, knowing that $a \in \mathbf{El}_i(\mathcal{D})$. The verbosity of the actual definition is to ensure a correct technical setup, so that realizability can be established. In particular, all recursive conditions are universally quantified with a restricted weakening $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}$ to ensure the gluing model is monotonic.

Definition 3.6. Given $\mathcal{D} :: A \approx B \in \mathcal{U}_i$,

- $\overrightarrow{\Gamma} \vdash T \ ®_i \ \mathcal{D}$ says that T is related to domain types A and B in $\overrightarrow{\Gamma}$ as a type at level i.
- $\overrightarrow{\Gamma} \vdash t : T \ ®_i \ a \in \mathbf{El}_i(\mathcal{D})$ says that in $\overrightarrow{\Gamma}$, t of type T is related to domain value a in $\mathbf{El}_i(\mathcal{D})$.

$\overrightarrow{\Gamma} \vdash T \ ®_i \ \mathcal{D}$ and $\overrightarrow{\Gamma} \vdash t : T \ ®_i \ a \in \mathbf{El}_i(\mathcal{D})$ are defined mutually by first well-founded recursion on i and then recursion on $\mathcal{D}$:

- $$\mathcal{D} := \frac{C \approx C' \in Ne}{\uparrow^{A} (C) \approx \uparrow^{A'} (C') \in \mathcal{U}_i}$$

  $\overrightarrow{\Gamma} \vdash T \ ®_i \ \mathcal{D}$ iff

  - T is a type at level i: $\Gamma \vdash T : \mathtt{Ty}_i$.

– For any restricted weakening $\overrightarrow{\sigma}$, that is $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}$, s.t.

$$\overrightarrow{\Delta} \vdash T[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(C[\overrightarrow{\sigma}]) : \mathtt{Ty}_i$$

$\overrightarrow{\Gamma} \vdash t : T \circledR_i \uparrow^{A''} (c) \in \mathbf{El}_i(\mathcal{D})$ iff

- $c \in Ne$.
- T is a type at level i: $\Gamma \vdash T : \mathtt{Ty}_i$.
- t is well typed: $\Gamma \vdash t : T$.
- For any restricted weakening $\overrightarrow{\sigma}$, that is $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}$, s.t.

$$\overrightarrow{\Delta} \vdash T[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(C[\overrightarrow{\sigma}]) : \mathtt{Ty}_i$$

  and

$$\overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c[\overrightarrow{\sigma}]) : T[\overrightarrow{\sigma}]$$

•

$$\mathcal{D} := \frac{}{\mathsf{N} \approx \mathsf{N} \in \mathcal{U}_i}$$

$\overrightarrow{\Gamma} \vdash T \circledR_i \mathcal{D}$ iff $\overrightarrow{\Gamma} \vdash T \approx \mathtt{Nat} : \mathtt{Ty}_i$.
$\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathcal{D})$ iff $\overrightarrow{\Gamma} \vdash t : \mathtt{Nat} \circledR a \in Nat$ and $\overrightarrow{\Gamma} \vdash T \approx \mathtt{Nat} : \mathtt{Ty}_i$, where $\overrightarrow{\Gamma} \vdash t : \mathtt{Nat} \circledR a \in Nat$ is an auxiliary inductive relation to relate syntactic terms of type $\mathtt{Nat}$ and domain values of PER Nat:

$$\frac{\overrightarrow{\Gamma} \vdash t \approx \mathsf{zero} : \mathtt{Nat}}{\overrightarrow{\Gamma} \vdash t : \mathtt{Nat} \circledR \mathsf{ze} \in Nat} \qquad \frac{\overrightarrow{\Gamma} \vdash t \approx \mathsf{succ}\ t' : \mathtt{Nat} \qquad \overrightarrow{\Gamma} \vdash t' : \mathtt{Nat} \circledR b \in Nat}{\overrightarrow{\Gamma} \vdash t : \mathtt{Nat} \circledR \mathsf{su}(b) \in Nat}$$

$$\frac{c \in Ne \qquad \forall\ \overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}\ .\ \overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}] \approx \mathsf{R}^{\mathsf{Ne}}_{\overrightarrow{z}}(c[\overrightarrow{\sigma}]) : \mathtt{Nat}}{\overrightarrow{\Gamma} \vdash t : \mathtt{Nat} \circledR \uparrow^{\mathsf{N}} (c) \in Nat}$$

- $$\mathcal{D} := \frac{j < i}{\mathsf{U}_j \approx \mathsf{U}_j \in \mathcal{U}_i}$$

  $\overrightarrow{\Gamma} \vdash T \circledR_i \mathcal{D}$ iff $\overrightarrow{\Gamma} \vdash T \approx \mathtt{Ty}_j : \mathtt{Ty}_i$.

  $\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathcal{D})$ iff

  - t is well typed: $\Gamma \vdash t : T$.
  - T is equivalent to $\mathtt{Ty}_j$: $\overrightarrow{\Gamma} \vdash T \approx \mathtt{Ty}_j : \mathtt{Ty}_i$.
  - a is in PER $\mathcal{U}_j$: $\mathcal{E} :: a \in \mathcal{U}_j$.
  - t and a are recursively related as types by well-foundedness: $\overrightarrow{\Gamma} \vdash t \circledR_j \mathcal{E}$.

  Note that $\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathcal{D})$ refers to $\overrightarrow{\Gamma} \vdash t \circledR_j \mathcal{E}$, so the definition requires a well-founded recursion on i.

- $$\mathcal{D} := \frac{\mathcal{E} :: \forall\ \kappa\ .\ A'[\kappa] \approx A''[\kappa] \in \mathcal{U}_i}{\blacksquare A' \approx \blacksquare A'' \in \mathcal{U}_i}$$

  $\overrightarrow{\Gamma} \vdash T \circledR_i \mathcal{D}$ iff there exists some T', such that

  - T is equivalent to $\square T'$: $\overrightarrow{\Gamma} \vdash T \approx \square T' : \mathtt{Ty}_i$.
  - For any $\overrightarrow{\Delta}'$ such that $\vdash \overrightarrow{\Delta}; \overrightarrow{\Delta}'$ and any restricted weakening $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}$, $T'[\overrightarrow{\sigma}; \Uparrow^{|\overrightarrow{\Delta}'|}]$ and $\mathcal{E}(\overrightarrow{\sigma}; \Uparrow^{|\overrightarrow{\Delta}'|})$ are recursively related as types:

  $$\overrightarrow{\Delta}; \overrightarrow{\Delta}' \vdash T'[\overrightarrow{\sigma}; \Uparrow^{|\overrightarrow{\Delta}'|}] \circledR_i \mathcal{E}(\overrightarrow{\sigma}; \Uparrow^{|\overrightarrow{\Delta}'|})$$

  $\overrightarrow{\Gamma} \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathcal{D})$ iff there exists some T', such that

  - t is well typed: $\overrightarrow{\Gamma} \vdash t : T$.
  - a is in $\mathbf{El}_i(\mathcal{D})$: $a \in \mathbf{El}_i(\mathcal{D})$.

- T is equivalent to $\square T'$: $\overrightarrow{\Gamma} \vdash T \approx \square T' : \mathtt{Ty}_i$.
- For any $\overrightarrow{\Delta}'$ such that $\vdash \overrightarrow{\Delta}; \overrightarrow{\Delta}'$ and any restricted weakening $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}$, the results of eliminating $t[\overrightarrow{\sigma}]$ and $a[\overrightarrow{\sigma}]$ are recursively related as terms:
$$\overrightarrow{\Delta}; \overrightarrow{\Delta}' \vdash \mathtt{unbox}_{|\overrightarrow{\Delta}'|}\ (t[\overrightarrow{\sigma}]) : T'[\overrightarrow{\sigma}; \Uparrow^{|\overrightarrow{\Delta}'|}] \circledR_i \mathtt{unbox} \cdot (|\overrightarrow{\Delta}'|, a[\overrightarrow{\sigma}]) \in \mathbf{El}_i(\mathcal{E}[\mathtt{mt}(\overrightarrow{\sigma}); \Uparrow^{|\overrightarrow{\Delta}'|}])$$

•

$$\mathcal{D} := \frac{\begin{array}{c}\mathcal{E}_1 :: \forall\ \kappa.\ A'[\kappa] \approx A''[\kappa] \in \mathcal{U}_i \\ \mathcal{E}_2 :: \forall\ \kappa, a \approx a' \in \mathbf{El}_i(\mathcal{E}_1[\kappa])\ .\ [\![T']\!](\mathtt{lext}(\overrightarrow{\rho}'[\kappa], a)) \approx [\![T'']\!](\mathtt{lext}(\overrightarrow{\rho}''[\kappa], a')) \in \mathcal{U}_i\end{array}}{Pi(A', T', \overrightarrow{\rho}') \approx Pi(A'', T'', \overrightarrow{\rho}'') \in \mathcal{U}_i}$$

$\overrightarrow{\Gamma}; \Gamma \vdash T \circledR_i \mathcal{D}$ iff there exist T_1 and T_2, such that

- T and $\Pi(x : T_1).T_2$ are equivalent: $\overrightarrow{\Gamma}; \Gamma \vdash T \approx \Pi(x : T_1).T_2 : \mathtt{Ty}_i$.
- T_2 is well typed: $\overrightarrow{\Gamma}; \Gamma, x : T_1 \vdash T_2 : \mathtt{Ty}_i$.
- For any restricted weakening $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}; \Gamma$,
  * $T_1[\overrightarrow{\sigma}]$ and $\mathcal{E}_1(\overrightarrow{\sigma})$ are recursively related: $\overrightarrow{\Delta} \vdash T_1[\overrightarrow{\sigma}] \circledR_i \mathcal{E}_1(\overrightarrow{\sigma})$, and
  * For any related s and b, i.e. $\overrightarrow{\Delta} \vdash s : T_1[\overrightarrow{\sigma}] \circledR_i b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$, and $\mathcal{E}_3 :: b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$, $T_2[\overrightarrow{\sigma}, s]$ and $\mathcal{E}_2(\overrightarrow{\sigma}, \mathcal{E}_3)$ are recursively related as types:
  $$\overrightarrow{\Delta} \vdash T_2[\overrightarrow{\sigma}, s] \circledR_i \mathcal{E}_2(\overrightarrow{\sigma}, \mathcal{E}_3)$$
  Note that this condition requires $\mathcal{E}_3 :: b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$ as an assumption, which technically can be derived from $\overrightarrow{\Delta} \vdash s : T_1[\overrightarrow{\sigma}] \circledR_i b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$. However, this fact requires a proof and thus cannot be used at the time of definition. Adding this assumption simplifies our definition.

$\overrightarrow{\Gamma}; \Gamma \vdash t : T \circledR_i a \in \mathbf{El}_i(\mathcal{D})$ iff there exist T_1 and T_2, such that

- t is well typed: $\overrightarrow{\Gamma}; \Gamma \vdash t : T$.
- a is in the PER $\mathbf{El}_i(\mathcal{D})$: $a \in \mathbf{El}_i(\mathcal{D})$.

– T and $\Pi(x : T_1).T_2$ are equivalent: $\overrightarrow{\Gamma}; \Gamma \vdash T \approx \Pi(x : T_1).T_2 : \mathtt{Ty}_i$.

– T_2 is well typed: $\overrightarrow{\Gamma}; \Gamma, x : T_1 \vdash T_2 : \mathtt{Ty}_i$.

– For any restricted weakening $\overrightarrow{\Delta} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Gamma}; \Gamma$,

  * $T_1[\overrightarrow{\sigma}]$ and $\mathcal{E}_1(\overrightarrow{\sigma})$ are recursively related: $\overrightarrow{\Delta} \vdash T_1[\overrightarrow{\sigma}] \circledR_i \mathcal{E}_1(\overrightarrow{\sigma})$, and
  * For any related s and b, i.e. $\overrightarrow{\Delta} \vdash s : T_1[\overrightarrow{\sigma}] \circledR_i b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$, and $\mathcal{E}_3 :: b \in \mathbf{El}_i(\mathcal{E}_1(\overrightarrow{\sigma}))$, the results of eliminating $t[\overrightarrow{\sigma}]$ and $a[\overrightarrow{\sigma}]$ are recursively related as terms :

$$\overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}]\ s : T_2[\overrightarrow{\sigma}, s] \circledR_i a[\overrightarrow{\sigma}] \cdot b \in \mathbf{El}_i(\mathcal{E}_2(\overrightarrow{\sigma}, \mathcal{E}_3))$$

3.9 Properties of Gluing Model

Similar to the PER model, properties of the gluing model also require adjustments for their statements and their proofs to be more type-theoretic.

3.9.1 Monotonicity

Monotonicity ensures that the gluing model is stable under restricted weakenings. Restricted weakenings apply to both syntax and semantics because they contain modal extensions to instruct both sides to travel among Kripke worlds. Following a similar strategy as in the PER model, the additional typing derivations $\mathcal{D}$ and $\mathcal{E}$ characterize $A \approx B$ and $A[\overrightarrow{\sigma}] \approx B[\overrightarrow{\sigma}]$ resp. to expose more clearly the proof structure and simplify the termination argument.

Lemma 3.13 (Monotonicity)**.** *Given a restricted weakening $\overrightarrow{\Gamma} \vdash_r \overrightarrow{\sigma} : \overrightarrow{\Delta}$, $\mathcal{D} :: A \approx B \in \mathcal{U}_i$ and $\mathcal{E} :: A[\overrightarrow{\sigma}] \approx B[\overrightarrow{\sigma}] \in \mathcal{U}_i$,*

- *if $\overrightarrow{\Delta} \vdash T \circledR_i \mathcal{D}$, then $\overrightarrow{\Gamma} \vdash T[\overrightarrow{\sigma}] \circledR_i \mathcal{E}$;*
- *if $\overrightarrow{\Delta} \vdash t : T \circledR_i a \in \boldsymbol{El}_i(\mathcal{D})$, then $\overrightarrow{\Gamma} \vdash t[\overrightarrow{\sigma}] : T[\overrightarrow{\sigma}] \circledR_i a[\overrightarrow{\sigma}] \in \boldsymbol{El}_i(\mathcal{E})$.*

3.9.2 Realizability

In completeness, realizability morally states that **El** is subsumed by *Nf*. In soundness, realizability has a similar structure. The realizability theorem states that the gluing models of types and terms are subsumed by $\overrightarrow{\Gamma} \vdash T \mathrel{\overline{\circledR}_i} \mathcal{D}$ and $\overrightarrow{\Gamma} \vdash t : T \mathrel{\overline{\circledR}_i} a \in \mathbf{El}_i(\mathcal{D})$ respectively, which constitutes the second step to the soundness proof.

Theorem 3.14 (Realizability). *Given $\mathcal{D} :: A \approx B \in \mathcal{U}_i$,*

- *if $\overrightarrow{\Gamma} \vdash T \mathrel{\circledR_i} \mathcal{D}$, then $\overrightarrow{\Gamma} \vdash T \mathrel{\overline{\circledR}_i} \mathcal{D}$.*
- *if $\overrightarrow{\Gamma} \vdash t : T \mathrel{\underline{\circledR}_i} c \in \boldsymbol{El}_i(\mathcal{D})$, then $\overrightarrow{\Gamma} \vdash t : T \mathrel{\circledR_i} \uparrow^A (c) \in \boldsymbol{El}_i(\mathcal{D})$;*
- *if $\overrightarrow{\Gamma} \vdash t : T \mathrel{\circledR_i} a \in \boldsymbol{El}_i(\mathcal{D})$, then $\overrightarrow{\Gamma} \vdash t : T \mathrel{\overline{\circledR}_i} a \in \boldsymbol{El}_i(\mathcal{D})$;*

3.9.3 Cumulativity and Lowering

Similar to the PER model, cumulativity of the gluing model also requires a lowering statement to handle the function cases and contravariant occurrences in type constructors in general:

Lemma 3.15 (Cumulativity and lowering). *Given $\mathcal{D} :: A \approx B \in \mathcal{U}_i$ and $\mathcal{E} :: A \approx B \in \mathcal{U}_{1+i}$ which we obtain by applying Lemma 3.7 to $\mathcal{D}$,*

- *if $\overrightarrow{\Gamma} \vdash T \mathrel{\circledR_i} \mathcal{D}$, then $\overrightarrow{\Gamma} \vdash T \mathrel{\circledR_{1+i}} \mathcal{E}$;*
- *if $\overrightarrow{\Gamma} \vdash t : T \mathrel{\circledR_i} a \in \boldsymbol{El}_i(\mathcal{D})$, then $\overrightarrow{\Gamma} \vdash t : T \mathrel{\circledR_{1+i}} a \in \boldsymbol{El}_{1+i}(\mathcal{E})$;*
- *if $\overrightarrow{\Gamma} \vdash t : T \mathrel{\circledR_{1+i}} a \in \boldsymbol{El}_{1+i}(\mathcal{E})$ and $\overrightarrow{\Gamma} \vdash T \mathrel{\circledR_i} \mathcal{D}$, then $\overrightarrow{\Gamma} \vdash t : T \mathrel{\circledR_i} a \in \boldsymbol{El}_i(\mathcal{D})$.*

The first two statements are just cumulativity. However, there is one more complication here in lowering: $\overrightarrow{\Gamma} \vdash t : T \mathrel{\circledR_i} a \in \mathbf{El}_i(\mathcal{D})$ does not directly imply $\overrightarrow{\Gamma} \vdash t : T \mathrel{\circledR_{1+i}} a \in \mathbf{El}_{1+i}(\mathcal{E})$! This is because the gluing model contains syntactic information about types so the lowering statement must in addition have an assumption about T and $\mathcal{D}$'s relation at a lower level, which is introduced by $\overrightarrow{\Gamma} \vdash T \mathrel{\circledR_i} \mathcal{D}$.

3.10 Fundamental Theorems and Soundness

In this section, I generalize the gluing model to K-substitutions and evaluation environments. This gluing model is in fact more complex than existing proofs on paper (Abel, 2013; Gratzer et al., 2019; Abel et al., 2017), in that this gluing model is again defined through induction-recursion, while existing proofs directly proceed by recursion on the structure of the domain contexts (or context stacks in the case of MINT). This existing proof technique will not work in mechanization, because this technique heavily relies on cumulativity to bring the gluing model for terms and values to a limit, so the generalization to K-substitutions and environments does not care about universe levels. Evidently, Agda does not support taking limits, so the only solution is to always keep track of universe levels. In this gluing model, an inductive definition $\Vdash \overrightarrow{\Gamma}$ defines the semantic well-formedness of context stacks, in which universe levels are maintained.

Definition 3.7. The semantic well formedness of context stacks $\mathcal{D} :: \Vdash \overrightarrow{\Gamma}$ and the gluing model for K-substitutions and environments $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \mathcal{D} \circledR \overrightarrow{\rho}$ are defined inductive-recursively:

- $$\mathcal{D} := \frac{}{\Vdash \epsilon; \cdot}$$

  $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \mathcal{D} \circledR \overrightarrow{\rho}$ iff $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \epsilon; \cdot$.

- $$\mathcal{D} := \frac{\mathcal{E} :: \Vdash \overrightarrow{\Gamma}}{\Vdash \overrightarrow{\Gamma}; \cdot}$$

  $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \mathcal{D} \circledR \overrightarrow{\rho}$ iff

  - $\overrightarrow{\sigma}$ is well typed: $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}; \cdot$.
  - There exists a K-substitution $\overrightarrow{\sigma}'$ and an modal offset n, such that
    - $\overrightarrow{\sigma}'$ is $\overrightarrow{\sigma}$'s truncation: $\overrightarrow{\Delta} \mid n \vdash \overrightarrow{\sigma} \mid 1 \approx \overrightarrow{\sigma}' : \overrightarrow{\Gamma}$,
    - modal offsets are equal: $\mathcal{O}(\overrightarrow{\sigma}, 1) = \mathcal{O}(\overrightarrow{\rho}, 1) = n$, and
    - $\overrightarrow{\sigma}'$ and $\overrightarrow{\rho} \mid 1$ are recursively related: $\overrightarrow{\Delta} \mid n \vdash \overrightarrow{\sigma}' : \mathcal{E} \circledR \overrightarrow{\rho} \mid 1$.

- $$\mathcal{D} := \frac{\exists i \qquad \mathcal{E} :: \Vdash \overrightarrow{\Gamma}; \Gamma \qquad \forall \overrightarrow{\Delta} \vdash \overrightarrow{\delta} : \mathcal{D} \circledR \overrightarrow{\rho}\ .\ \Sigma(\mathcal{J} :: [\![T]\!](\overrightarrow{\rho}) \in \mathcal{U}_i).\ \overrightarrow{\Delta} \vdash T[\overrightarrow{\delta}] \circledR_i \mathcal{J}}{\Vdash \overrightarrow{\Gamma}; \Gamma.T}$$

  $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \mathcal{D} \circledR \overrightarrow{\rho}$ iff

  - $\overrightarrow{\sigma}$ is well typed: $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}; \Gamma.T$.
  - There exists a K-substitution $\overrightarrow{\sigma}'$ and t, such that
    * $\overrightarrow{\sigma}'$ is $\overrightarrow{\sigma}$ with the topmost term dropped: $\overrightarrow{\Delta} \vdash \mathsf{wk} \circ \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Gamma}; \Gamma$,
    * t is that topmost term: $\overrightarrow{\Delta} \vdash v_0[\overrightarrow{\sigma}] \approx t : T[\overrightarrow{\sigma}']$.
    * T evaluates in $\mathtt{drop}(\overrightarrow{\rho})$ and the result is in $\mathcal{U}_i$: $\mathcal{J}' :: [\![T]\!](\mathtt{drop}(\overrightarrow{\rho})) \in \mathcal{U}_i$.
    * t and $\overrightarrow{\rho}(0)$ are related at level i: $\overrightarrow{\Delta} \vdash t : T[\overrightarrow{\sigma}'] \circledR_i \rho(0) \in \mathbf{El}_i(\mathcal{J}')$, where $(\_, \rho) := \overrightarrow{\rho}(0)$,
    * $\overrightarrow{\sigma}'$ and $\mathtt{drop}(\overrightarrow{\rho})$ are recursively related: $\overrightarrow{\Delta} \vdash \overrightarrow{\sigma}' : \mathcal{E} \circledR \mathtt{drop}(\overrightarrow{\rho})$.

The semantic judgments for soundness is given in terms of two gluing models.

Definition 3.8. The semantic judgments for soundness are defined as follows:

$$\frac{\exists i \qquad \mathcal{D} :: \Vdash \overrightarrow{\Gamma} \qquad \forall\, \overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \mathcal{D} \circledR \overrightarrow{\rho}\ .\ \overrightarrow{\Delta} \vdash t[\overrightarrow{\sigma}] : T[\overrightarrow{\sigma}] \circledR_i [\![t]\!](\overrightarrow{\rho}) \in \mathbf{El}_i([\![T]\!](\overrightarrow{\rho}))}{\overrightarrow{\Gamma} \Vdash t : T}$$

$$\frac{\mathcal{D}_1 :: \Vdash \overrightarrow{\Gamma} \qquad \mathcal{D}_2 :: \Vdash \overrightarrow{\Gamma}' \qquad \forall\, \overrightarrow{\Delta} \vdash \overrightarrow{\sigma} : \mathcal{D}_1 \circledR \overrightarrow{\rho}\ .\ \overrightarrow{\Delta} \vdash \overrightarrow{\delta} \circ \overrightarrow{\sigma} : \mathcal{D}_2 \circledR [\![\overrightarrow{\delta}]\!](\overrightarrow{\rho})}{\overrightarrow{\Gamma} \Vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}'}$$

Finally,

Theorem 3.16 (Fundamental)**.**

- *If* $\vdash \overrightarrow{\Gamma}$*, then* $\Vdash \overrightarrow{\Gamma}$.
- *If* $\overrightarrow{\Gamma} \vdash t : T$*, then* $\overrightarrow{\Gamma} \Vdash t : T$.
- *If* $\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$*, then* $\overrightarrow{\Gamma} \Vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}$.

Theorem 3.17 (Soundness)**.** *If* $\overrightarrow{\Gamma} \vdash t : T$*, then* $\overrightarrow{\Gamma} \vdash t \approx \mathit{nbe}^{T}_{\overrightarrow{\Gamma}}(t) : T$.

Proof. First apply the fundamental theorems and obtain $\overrightarrow{\Gamma} \Vdash t : T$. Moreover, given $\overrightarrow{\Gamma} \vdash \overrightarrow{I} : \overrightarrow{\Gamma} \circledR \uparrow^{\overrightarrow{\Gamma}}$, t and $[\![t]\!](\uparrow^{\overrightarrow{\Gamma}})$ are related. The goal is concluded by further applying realizability. □

The fundamental theorem for the gluing model leads to a few more consequences which are difficult to prove syntactically. First, standard injectivity and canonicity hold in Mint.

Lemma 3.18 (Injectivity of Type Constructors)**.**

- *If* $\overrightarrow{\Gamma} \vdash \square T_1 \approx \square T_2 : \texttt{Ty}_i$*, then* $\overrightarrow{\Gamma}; \cdot \vdash T_1 \approx T_2 : \texttt{Ty}_i$*.*
- *If* $\overrightarrow{\Gamma}; \Gamma \vdash \Pi(x : S_1).T_1 \approx \Pi(x : S_2).T_2 : \texttt{Ty}_i$*, then* $\overrightarrow{\Gamma}; \Gamma \vdash S_1 \approx S_2 : \texttt{Ty}_i$ *and* $\overrightarrow{\Gamma}; \Gamma, x : S_1 \vdash T_1 \approx T_2 : \texttt{Ty}_i$*.*

Lemma 3.19 (Canonicity of `Nat`)**.** *If* $\epsilon; \cdot \vdash t : \texttt{Nat}$*, then* $\epsilon; \cdot \vdash t \approx \texttt{succ}^n\ \texttt{zero} : \texttt{Nat}$ *for some number* n*.*

The following lemma about universe levels is also interesting. It says that if two types are equivalent and they are well typed at a different level, then they are equivalent also at that level. This lemma is intuitive but very challenging to prove syntactically.

Lemma 3.20 (Type equivalence)**.** *If* $\overrightarrow{\Gamma} \vdash T_1 \approx T_2 : \texttt{Ty}_i$*,* $\overrightarrow{\Gamma} \vdash T_1 : \texttt{Ty}_j$ *and* $\overrightarrow{\Gamma} \vdash T_2 : \texttt{Ty}_j$*, then* $\overrightarrow{\Gamma} \vdash T_1 \approx T_2 : \texttt{Ty}_j$*.*

Proof. By $\overrightarrow{\Gamma} \vdash T_1 \approx T_2 : \texttt{Ty}_i$ and completeness, T_1 and T_2 have equal normal form. The goal is concluded by soundness and transitivity. □

As the final and conclusive theorem, I show the consistency of Mint.

Theorem 3.21 (Consistency)**.** *There is no closed term of type* $\Pi(x : \texttt{Ty}_i).x$*. That is, there is no* t *such that the following judgment holds:*

$$\epsilon; \cdot \vdash t : \Pi(x : \texttt{Ty}_i).x$$

The theorem effectively states that there is no generic way to construct a term for an arbitrary type. If Mint has a bottom type, then the consistency proof can be simply reduced to the consistency of our meta-language (i.e. Agda). The current definition of Mint does not have a bottom type, so the proof of this theorem is set up as follows:

Proof. Note that consistency is equivalent to proving that there is no t' such that

$$\epsilon; x : \texttt{Ty}_i \vdash t' : x$$

Note that, by soundness, t' must be equivalent to some neutral term u by NbE, because its type (x) is neutral. So our goal is to show that this u also does not exist:

$$\epsilon; x : \texttt{Ty}_i \vdash u : x$$

Now we do induction on u. We show that in $\epsilon; x : \texttt{Ty}_i$, there does not exist a neutral term of any type other than $\texttt{Ty}_j$ (i and j are not necessarily equal due to cumulativity), and thus it is impossible for u to have type x. Otherwise, that would require $\texttt{Ty}_i$ and x to be equivalent, which can be rejected by completeness as they do not evaluate to the same normal form. $\square$

3.11 Summary

My investigation of the Kripke-style type theories has reached the end here. Previously, I have looked in the normalization problem of $\lambda^{\square}$ and scale the whole setup quite naturally to MINT. This chapter combines the technique developed in Chapter 2 and Abel (2013)'s method and eventually proves MINT's normalization property and other important properties.

The example shows that MINT already finds an application as a program logic for MetaML, MetaOCaml, etc.. MINT in general can also be used as a syntactic theory generally for a mathematical setup where a necessity modality is involved (see e.g. (Licata et al., 2018)). However, MINT is not very satisfactory as a meta-programming system in a proof assistant. More specifically, as it is right now, it does not support any intensional analysis operations. Meanwhile, in proof assistants, users often need to analyze the syntactic structures of terms when developing decision procedures and proof heuristics. Therefore, in the next Part, I switch to another style, which opens the path to supporting intensional analysis coherently in type theory.

Part II

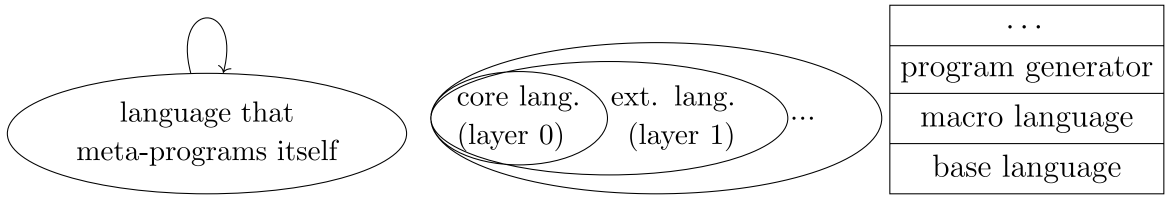
Layered Modal Type Theories

4

A Layered Modal Type Theory for Intensional Analysis

In Part I, I introduced two Kripke-style type theories. The Kripke-style $S4$ faithfully models the quasi-quoting style under Curry-Howard correspondence as discussed in Chapter 1 and its laws correspond to code composition and code running in meta-programming. However, it is not clear how Kripke-style systems should support intensional analysis. In fact, the Kripke-style systems are more suitable for being program logics for some practical meta-programming systems like MetaML and MetaOCaml, than being a meta-programming system itself, due to the extensionality of $\square$ introduced by the η rule. Then how exactly can a type theory support meta-programming with intensional analysis without jeopardizing its normalization property?

Let us take a step back and reconsider what other work has accomplished. On the practical side, in Chapter 1, I have discussed a number of tools readily available in various proof assistants. Meta-programming in Agda (van der Walt and Swierstra, 2012), in Idris (Christiansen and Brady, 2016), in Lean (Ebner et al., 2017) and in Coq



(a) homogeneous style (b) layered style (c) heterogeneous style

Figure 4.1: Comparison between heterogenous, homogenous, and layered meta-programming systems

using MetaCoq (Sozeau et al., 2020; Anand et al., 2018) are implemented by reflection. Through reflection, users obtain the syntax of programs as untyped abstract syntax trees (ASTs). Meta-programs in this sense are just regular programs that manipulate these ASTs. A major drawback of reflection is that the these ASTs are untyped, so there is no guarantee that code generated by meta-programs can be successfully evaluated to valid programs. In other words, reflection effectively makes no use of powerful dependent types implemented in these proof assistants. Mtac (Ziliani et al., 2013; Kaiser et al., 2018) slightly improves the situation. It is implemented by instrumenting Coq's typechecking kernel and function as an extension of Coq's proof language, so it provides a frontend which helps to write meta-programs that return well-typed code. Nevertheless, it still does not provide a formal guarantee for well-scopedness of generated code, nor do they have critical type-theoretic properties like confluence and normalization.

On the theoretical end, foundations that combine meta-programming with type-theory typically fall into two styles: the homogeneous style and the heterogeneous style. Homogeneous meta-programming uses a single language capable of meta-programming itself (depicted in Fig. 4.1a). In Part I, $\lambda^{\Box}$ by Davies and Pfenning (2001) and Mint fall into this style. Other than the Kripke style, Davies and Pfenning also formulate $\lambda^{\Box}$ in the dual-context style, where $\Box$ separate meta-programming and programming into two different zones. The dual-context style is the basis for the type theories in this part. Nanevski et al. (2008) subsequently extend $\lambda^{\Box}$ in the dual-context style with contextual types, allowing meta-programming on open code. As previously discussed, these systems do not suggest how intensional analysis can be supported. In fact, supporting intensional analysis in the homogeneous style while retaining properties like confluence

System	Quotation	Intensional analysis	Code running	Normalization
Reflection, instrumentation	✓	✓	✓	
Moebius	✓	✓	✓	
(Contextual) $\lambda^{\Box}$	✓		✓	✓
Cocon	✓	✓		✓
2LTT	✓			✓
Layered modal type theory	✓	✓	✓	✓

Table 4.1: A comparison of features among different systems

and normalization has been fraught with difficulties (c.f. (Schürmann et al., 2001)). A significant step towards supporting pattern matching on open code in a homogeneous style is taken in Moebius (Jang et al., 2022). Moebius is based on System F-style polymorphism. However, its pattern matching does not guarantee coverage and Moebius lacks normalization.

A heterogeneous system distinguishes between the meta-language and the object language (illustrated by Fig. 4.1c). Recently, Kovács (2022) adapts 2-level type theory (2LTT), originally conceived for homotopy type theory, to dependently typed meta-programming. Here, a dependently typed meta-language sits on top of a less expressive object language. However, this type theory does not support intensional analysis and does not support running code. It is used to generate code running in a different environments (e.g. generate Haskell code in a dependent type theory). In contrast, Cocon (Pientka et al., 2019), another 2-level type theory following in the footsteps of previous work (Davies and Pfenning, 2001; Nanevski et al., 2008), supports modeling open code and intensional code analysis. In Cocon, MLTT sits on top of the logical framework LF, which encodes formal systems. In particular, this allows us to encode sub-languages of MLTT (for example simply-typed or polymorphic lambda-calculus) in LF and then write meta-programs about them by pattern matching on code. In principle, an explicit interpretation function can be defined to interpret terms in the sub-language into MLTT itself and execute them as MLTT programs. Although this provides a powerful framework for meta-programming and these heterogeneous systems are modular, this comes at a price: a definition in one level is not directly accessible or reused in the other level. Further, Cocon requires explicit interpretation functions to be able to execute code. More importantly, this separation into two languages leads to two separate investigations of meta-theoretic properties for two languages and ultimately two separate normalization arguments. How to elegantly scale these languages to multiple levels is not obvious.

Table 4.1 summarizes some previous work. In the table, I list four features that I am concerned about in this part:

- quotation, which obtains the syntactic representation of programs;
- pattern matching on code, which is the most general form of intensional analysis;
- code running, which extracts a term of type T given code of type T for all T;
- normalization, which implies the logical consistency of the system.

Among the four, quotation exists in all listed work. The practical tools that use reflection or instrumentation also support intensional analysis and code running, but as discussed previously, they do not have a type-theoretic foundation and hence normalization. For the foundational systems that have normalization, it is easy for the homogeneous systems to support code running but not intensional analysis, while the heterogeneous systems are the other way around. Therefore, the research question is, is there another style for meta-programming such that all these features are supported?

The question is positively answered by *the layered style* developed in this part. Illustrated in Fig. 4.1b, the layered style is in the middle of the homogeneous and heterogeneous styles. Syntactically, a layered modal type theory employs a uniform syntax just like the homogeneous style. However, its typing judgments are indexed by a layering index. Effectively, a layered system has a fixed number of layers of languages. The relation among these languages is characterized by *the matryoshka principle*: the language at layer i is *contained* in its meta-language at layer $i + 1$. What is added to layer i at layer $i+1$ is the ability to inspect and analyze code from the language at layer i. More importantly, the equivalence judgments are also parameterized by a layering index. This allows us to distinguish and control computational behaviors at different layers.

As a principle, β and η equivalence is only allowed at the highest layer, so all lower layers are treated as static code, which is only identified by its syntax. Controlling the computational behaviour through layers is necessary for sound intensional analysis and ultimately leads to establish normalization. This informal intuition of the matryoshka principle is formally characterized by two guiding lemmas: the lifting lemma, which enables code running, and the static code lemma, which enables pattern matching on code.

On the other hand, a layered modal type theory has similarities to the heterogeneous style in the semantics. Semantically, the layering index clearly distinguishes different languages, each of which requires its own model. In the semantics, the model for each lower layer must describe how the code runs at the highest layer and how its syntax is analyzed, much like a heterogeneous system. The main difference between a layered system and a heterogeneous one in the semantics is that in the former, models among different layers are related by the *layering restriction lemma* (c.f. Lemma 4.21), a semantic counterpart of the matryoshka principle, whereas in a heterogeneous system, different languages might be completely unrelated.

In this chapter, I introduce a simply typed layered modal type theory with context variables (Pientka, 2008), which is capable of code running and pattern matching on code, a general form of intensional analysis. I follow Abel et al. (2018) and give a weak normalization proof by weak-head reductions and a conversion checking algorithm. The advantage of this conversion checking algorithm compared to the previous one based on strong normalization is that this algorithm fails faster if two terms are actually not convertible. However, the downside is that the semantic construction is more verbose and there are more technical details. This complication remains carried over to dependent types, which is based on the development in this chapter. A strong normalization based on NbE in Part I would be nice; it at least will lead to a simpler normalization proof and a trivial conversion checking algorithm, though I expect a major complication in the PER model and I have little idea about how to handle this problem at this moment. I will discuss this difficulty towards the end of this thesis (c.f. Sec. 6.5.2).

A simpler layered modal type theory has been published (Hu and Pientka, 2024b), where a strong normalization proof is given by a presheaf model. This chapter extends this simpler system with context variables and context meta-functions. The technical development in this chapter is completely new.

In this chapter, just similar to the Kripke-style systems, I will set up a framework for investigations that will scale up to dependent types.

4.1 Example Programs in 2-layered Modal Type Theory

In this section, let us reconsider the `meta-mult2` example in Chapter 1, which computes the code of an iterated sum to compute multiplication, in 2-layered modal type theory. Then I will show how to optimize the generated code by `meta-mult2` using pattern matching on code.

4.1.1 A Layered Multiplication Function

The `meta-mult2` example in Chapter 1 can be directly translated to 2-layered modal type theory as follows:

```
meta-mult2 : Nat → □ (Nat → Nat)
meta-mult2 zero      = box (λ x. 0)
meta-mult2 (succ n) = letbox u ← meta-mult2 n in box (λ x. (u x) + x)
```

The `meta-mult2` function takes a natural number as a multiplier and returns code (denoted by □) of a function type `Nat → Nat`. The `meta-mult2` function is defined by a recursion on the input number. If it is `zero`, then the generated code is the constant function of `0` since any number multiplied by `0` is `0`. In the successor case, the recursive call `meta-mult2 n` computes the code for multiplying some number with `n`. Since the return type of the recursive call is `□ (Nat → Nat)`, `letbox` eliminates the □ type and binds the resulting code of type `Nat → Nat` to `u`. The new variable `u` is a *meta-variable* and can be introduced by `letbox`. A meta-variable is a placeholder for code. It remains accessible under a `box` constructor. Program variables like `n`, on the other hand, are not accessible inside of `box` for code constructions, leading to a clear phase separation between program variables and meta-variables. The final code inside `box` first uses `u x` to construct the iterated sum of `n` `x`'s and then adds `x` to the result.

The following are examples for running the `meta-mult2` function:

```
meta-mult2 1 ≈ box (λ x. (λ x. 0) x + x)
             ≉ box (λ x. 0 + x) ≉ box (λ x. x)
meta-mult2 2 ≈ box (λ x. (λ x. (λ x. 0) x + x) x + x)
```

Note that the right-hand sides contain multiple redundant redexes such as `(λ x. 0) x`. However, since they represent code, they are not reduced inside of `box`. In the remainder of this section, I will show how to generate more efficient code and optimize generated code using intensional code analysis.

Although there is room to optimize the generated code, we can already *run* it, which is critical for a meta-programming system. The following program

```
letbox u ← meta-mult2 2 in u : Nat → Nat ≈ λ x. x + x
```

uses the `meta-mult2` meta-function to generate a regular function adding the same number twice. Since `u` has a function type, it can be invoked with an argument as expected:

```
letbox u ← meta-mult2 2 in u 5 ≈ 10
```

gives `10`.

4.1.2 Contextual Types for Open Code

One immediate opportunity for the optimization of the `meta-mult2` function are the redundant redices. The result code should simply be an iterated sum of `x`'s. Following Nanevski et al. (2008, Sec. 6), 2-layered modal type theory is extended with contextual types, enabling meta-programming with open code. Rather than constructing intermediate λ abstractions, the following meta-function uses *contextual types* to recursively compute an iterated sum of `x`'s:

```
meta-mult3 : Nat → □ (x : Nat ⊢ Nat)
meta-mult3 zero     = box (x. 0)
meta-mult3 (succ n) = letbox u ← meta-mult3 n in box (x. u[x/x] + x)
```

The `meta-mult3` function is nearly identical to `meta-mult2`. However, its return type has become a contextual type `□ (x : Nat ⊢ Nat)`. This type denotes a code of type `Nat` (to the right of the turnstile) with a regular context with an open variable `x` of type `Nat` (to the left of the turnstile). In general, the number of open variables is arbitrary. A contextual type with no open variable degenerates to a closed □ type.

With contextual types, code inside `box`, can directly refer to open variables like `x`. The base case does not use `x`. The `succ` case similarly invokes the recursive call `meta-mult3 n` and uses `letbox` to bind a new meta-variable `u` to a type of open code `(x : Nat ⊢ Nat)`. This binding is read that `u` has type `Nat` with an open variable `x` of type `Nat`. When using a meta-variable, its regular variables must be instantiated. In this case, an identity (or more generally a simple renaming) substitution `[x/x]` suffices. This substitution eagerly replaces `x` for `x` when `u` is replaced by a concrete piece of code.

The `meta-mult3` function then results in cleaner and more compact code compared to the previously generated code by `meta-mult2`:

```
meta-mult3 1 ≈ box (x. 0 + x)       ≉ box (x. x)
meta-mult3 2 ≈ box (x. (0 + x) + x) ≉ box (x. x + x)
```

Thanks to the substitutions, redundant redexes have been completely eliminated. However, some redundant `0`'s are left behind, which I will handle in the next example. Contextual types still allow code running as long as regular variables are instantiated. This program

```
  letbox u ← meta-mult3 2 in λ y. u[y/x]
≈ λ y. (0 + y) + y ≈ λ y. y + y
```

generates a regular function computing twice the input by substituting the open variable `x` of `u` with the `y` introduced by a λ. The generated function behaves identically to the one generated by `meta-mult2` above. In the generated function, since computation does propagate under λ, the redundant `0` does go away by computation. The open variable `x` of `u` can also be substituted by a concrete number:

```
letbox u ← meta-mult3 2 in u[5/x] ≈ 10
```

In the next section, I will use pattern matching on code to optimize away the redundant `0`'s as `0` is the unit element of addition. However, this optimization cannot be performed automatically as `0 + x` and `x` are simply two different syntax trees inside of `box` and thus their codes are not convertible.

4.1.3 Pattern Matching for Intensional Analysis

An easy way to improve the previous implementation is to *pattern match* on the resulting code and remove all occurrences of `0`. However, supporting pattern matching on code in a type-theoretic setting has been notoriously difficult. Previous attempts in the homogeneous style fail to retain the normalization property. To illustrate, consider the intensional `isapp` function (Kavvos, 2021; Gabbay and Nanevski, 2013). This function simply looks at the structure of the input code and returns `true` if this piece of code is a function application, or `false` otherwise. Note that `isapp`'s behavior purely depends on the syntactic structure of its argument. In 2-layered modal type theory, this function can be implemented by pattern matching on code:

```
isapp : (g : Ctx) ⇒ □ (g ⊢ Nat) → Bool
isapp g x = match x with | ?u ?u' ⇒ true | _ ⇒ false
```

This function is polymorphic in the regular context `g` due to the *meta-function type* `(g : Ctx) ⇒ ...`, following Pientka (2008). A meta-function type introduces a regular context variable `g`, which can be used later in contextual types, e.g. `□ (g ⊢ Nat)` in the example. It means that the `isapp` function works for any regular context. In the implementation, `ispp` pattern-matches on the input code `x`. In the first branch, the result is `true` if `x` is some function application. Here, `?u` and `?u'` are both *pattern variables*. Question marks are used to distinguish pattern variables and constants, e.g. `zero` and `succ` which are the constructors of `Nat`. This distinction is only necessary in the patterns, and these question marks do not appear in the body of a branch. The pattern variables `u` and `u'` capture the code of the function and the argument respectively if `x` is a function application. The other branches are captured by the wildcard and all return `false`. Let us see how this function behaves:

```
isapp . (box ((λ x. x) 10)) ≈ true
isapp . (box 10)            ≈ false
```

where . denotes the empty context.

A problem which previous attempts encounter is what should intensional analysis do to `box`'ed meta-variables as scrutinees. Consider the following two execution of the same program, for some well-typed closed code `t` and `s`,

```
  letbox u ← box (t s) in isapp . (box u)
≈ isapp . (box (t s))           ≈ true
  letbox u ← box (t s) in isapp . (box u)
≈ letbox u ← box (t s) in false ≈ false
```

In the second execution, `isapp (box u)` is evaluated first, and then the overall result is `false`. In layered modal type theory, however, this issue does not exist because pattern matching on `box u` is neutral. Hence, the program only evaluates to `true`. This is a subtle but critical design decision which ultimately enables sound intensional analysis and normalization.

With sound pattern matching on code, a simple arithmetic simplifier is implemented to remove the redundant `0`'s in the code generated previously by `meta-mult3`:

```
simp : □ (x : Nat ⊢ Nat) → □ (x : Nat ⊢ Nat)
simp y = match y with
         | 0 + ?u    ⇒ box (x. u[x/x])
         | ?u + ?u' ⇒ letbox u1 = simp (box (x. u[x/x]))
                      in box (x. u1[x/x] + u'[x/x])
         | _         ⇒ y
```

In the first case, `0` is removed from the addition. In the second case, only the first addend is recursively simplified because redundant `0`'s appear only in the leftmost addend. In the last case, no optimization is needed. Since pattern matching is covering, this default case is necessary. At last, a wrapper function `mult-simp` invokes `simp` to simplify the code generated by `meta-mult3`:

```
mult-simp : Nat → □ (x : Nat ⊢ Nat)
mult-simp n = simp (meta-mult3 n)
```

The `mult-simp` function generates the most concise code:

```
mult-simp 1 = box (x. x)
mult-simp 2 = box (x. x + x)
```

With pattern matching on code, users have full control over generated code.

4.2 Syntax And Well-formedness

In this section, I introduce the syntax and the judgments for 2-layered modal type theory. This type theory extends simply typed λ-calculus (STLC) with the power of meta-programming: code running, pattern matching on code, as well as context polymorphism. All these features coexist coherently: a weak normalization proof and the decidability of convertibility will be proved towards the end of this chapter. In this section, the somewhat vague and informal matryoshka principle and the diagram in Fig. 4.1b are captured by two guiding lemmas: the lifting lemma, which enables code running, and the static code lemma, which enables pattern matching on code. The diagram in Fig. 4.1b intuitively depicts the layered typing judgment $\Psi; \Gamma \vdash_i t : T$

$\boxed{\vdash \Psi}$ Meta-context Ψ is well-formed. $\boxed{\Psi \vdash B}$ Meta-binding B is well-formed.

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Psi \qquad \Psi \vdash B}{\vdash \Psi, B} \qquad \frac{\vdash \Psi}{\Psi \vdash g : \mathsf{Ctx}} \qquad \frac{\Psi \vdash_0 \Gamma \qquad \Psi \vdash_0 T}{\Psi \vdash u : (\Gamma \vdash T)}$$

$\boxed{\Psi \vdash_i \Gamma}$ Regular context Γ is well-formed in Ψ at layer i.

$$\frac{\vdash \Psi}{\Psi \vdash_i \cdot} \qquad \frac{\vdash \Psi \qquad g : \mathsf{Ctx} \in \Psi}{\Psi \vdash_i g} \qquad \frac{\Psi \vdash_i \Gamma \qquad \Psi \vdash_i T}{\Psi \vdash_i \Gamma, x : T}$$

$\boxed{\Psi \vdash_i T}$ Type T is well-formed in Ψ at layer i.

$$\frac{\vdash \Psi}{\Psi \vdash_i \texttt{Nat}} \qquad \frac{\Psi \vdash_i S \qquad \Psi \vdash_i T}{\Psi \vdash_i S \longrightarrow T} \qquad \frac{\Psi \vdash_0 \Delta \qquad \Psi \vdash_0 T}{\Psi \vdash_1 \Box(\Delta \vdash T)} \qquad \frac{\Psi, g : \mathsf{Ctx} \vdash_1 T}{\Psi \vdash_1 (g : \mathsf{Ctx}) \Rightarrow T}$$

Figure 4.2: Well-formedness of meta- and regular contexts and types

indexed by $i \in [0, 1]$. When $i = 0$, t is a term in STLC. When $i = 1$, t could also be meta-programs. In other words, layer 0 is subsumed by layer 1. This relation is critical to obtain a better understanding of layered systems.

The syntax is introduced in two parts. I will discuss its types first, and then its terms.

$$\begin{array}{rcll}
i & & & \text{(Layer, } i \in [0,1]) \\
x, y & & & \text{(Regular Variables)} \\
u & & & \text{(Meta-variables)} \\
g & & & \text{(Contextual Variables)} \\
S, T & := & \texttt{Nat} \mid \Box(\Gamma \vdash T) \mid S \longrightarrow T \mid (g : \mathsf{Ctx}) \Rightarrow T & \text{(Types)} \\
B & := & u : (\Gamma \vdash T) \mid g : \mathsf{Ctx} & \text{(Meta-Bindings)} \\
\Phi, \Psi & := & \cdot \mid \Phi, B & \text{(Meta-contexts)} \\
\Gamma, \Delta & := & \cdot \mid g \mid \Gamma, x : T & \text{(Regular Contexts)}
\end{array}$$

Layered modal type theory supports natural numbers ($\texttt{Nat}$), contextual types ($\Box(\Gamma \vdash T)$), function types ($S \longrightarrow T$), and meta-functions ($(g : \mathsf{Ctx}) \Rightarrow T$), all of which have been introduced in Sec. 4.1. What distinguishes layered modal type theory from the Kripke-style systems is the various kinds of variables. In this case, there are three kinds of variables:

- regular variables ranged over by x and y;
- meta-variables that represent holes of code, ranged over by u;
- context variables that stand for regular contexts, ranged over by g.

Regular variables are stored in regular regular contexts and are bound to types. Meta-variables are stored in meta-contexts. A binding $u : (\Gamma \vdash T)$ says that u stands for a piece of code of type T, with open variables in Γ. The length of Γ is arbitrary and may contain a context variable. Context variables are also stored in the meta-context and can be referred to in Γ in $\square(\Gamma \vdash T)$. They are meant to be substituted by a concrete regular context. The context variables provide abstraction over all contexts, so that a function can be agnostic to regular contexts. Both kinds of contexts are part of the typing judgment $\Psi; \Gamma \vdash_i t : T$ to be discussed shortly as a context pair $\Psi; \Gamma$.

The matryoshka principle is reflected in the layering index i. Since this particular system which I am introducing is 2-layered, i is either 0 or 1. The index i is responsible for indexing almost all judgments in the system. The well-formedness of contexts and types is described in Fig. 4.2. These three judgments are mutually defined.

- $\vdash \Psi$ states the well-formedness of a meta-context Ψ.
- $\Psi \vdash_i \Gamma$ states the well-formedness of a regular context Γ at layer i. The base case of a regular context can either be an empty regular context, or be a well-scoped contextual variable g. In the latter case, g can be substituted by another regular context.
- $\Psi \vdash_i T$ states that T is well-formed in Ψ at layer i. Note that the well-formedness of T does not depend on any regular context, so this system is not completely dependently typed.

In $\Psi \vdash_i T$, for `Nat` and functions, which come from STLC, their well-formedness is parameterized by i. This makes them available at both layers. Whereas, for contextual types ($\square(\Gamma \vdash T)$) and meta-functions ($(g : \mathsf{Ctx}) \Rightarrow T$), since they are for meta-programming, they are only available at layer 1. The distinction in layers is an important characteristic of layered systems.

When a (meta-)programmer writes meta-programs, they conceptually distinguish between programs that are dynamic and compute, and code that is static and syntactic. In a homogeneous system, this distinction is captured by types, i.e. program t has type T while code has type $\square(\Gamma \vdash T)$. However, a term t itself does not provide information about whether it is inside of a `box` (hence treated as code), or outside of a `box` (hence a program). For example, only knowing that `succ zero` has type `Nat` does not reveal whether it is a piece of code or a program. The typing judgment for homogeneous systems like $\Psi; \Gamma \vdash t : T$ Pfenning and Davies (2001); Davies and Pfenning (2001) only provides typing information, and does not a priori determine whether t should be considered as code or as a program. Even though one major advantage of a homogeneous system is to use the same language for code and programs, this lack of information is the key reason for the challenges that we face when combining type theory and intensional analysis.

Layered modal type theory makes the distinction between code and programs explicit. When $i = 0$, terms are code and do not compute, and when $i = 1$, terms are programs and therefore have rich reduction behaviors. More specifically, layering controls

1. what types are valid at each layer,
2. what terms are well-typed at each layer, and
3. what terms are equivalent at each layer.

The well-formedness of types $\Psi \vdash_i T$ addresses the control of valid types at each layer. The matryoshka principle is characterized by the following lifting lemma, which states that regular contexts and types at layer 0 are subsumed by layer 1. In other words, the language at layer 1 may refer to all types in STLC.

Lemma 4.1 (Lifting for types and regular contexts)**.**

- *If* $\Psi \vdash_0 T$*, then* $\Psi \vdash_1 T$*.*
- *If* $\Psi \vdash_0 \Gamma$*, then* $\Psi \vdash_1 \Gamma$*.*

$\boxed{\Psi; \Gamma \vdash_i t : T}$ Term t has type A in contexts Ψ and Γ at layer i.

$$\frac{\Psi \vdash_i \Gamma \qquad x : T \in \Gamma}{\Psi; \Gamma \vdash_i x : T} \qquad \frac{\Psi; \Gamma \vdash_i \delta : \Delta \qquad u : (\Delta \vdash T) \in \Psi}{\Psi; \Gamma \vdash_i u^\delta : T} \qquad \frac{\Psi \vdash_i \Gamma}{\Psi; \Gamma \vdash_i \mathsf{zero} : \mathtt{Nat}}$$

$$\frac{\Psi; \Gamma \vdash_i t : \mathtt{Nat}}{\Psi; \Gamma \vdash_i \mathsf{succ}\ t : \mathtt{Nat}} \qquad \frac{\Psi; \Gamma, x : S \vdash_i t : T}{\Psi; \Gamma \vdash_i \lambda x.t : S \longrightarrow T} \qquad \frac{\Psi; \Gamma \vdash_i t : S \longrightarrow T \qquad \Psi; \Gamma \vdash_i s : S}{\Psi; \Gamma \vdash_i t\ s : T}$$

$$\frac{\Psi \vdash_1 \Gamma \qquad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \mathtt{box}\ t : \square(\Delta \vdash T)} \qquad \frac{\begin{array}{c}\Psi; \Gamma \vdash_1 s : \square(\Delta \vdash T) \qquad \Psi \vdash_0 \Delta \qquad \Psi \vdash_0 T \\ \Psi \vdash_1 T' \qquad \Psi, u : (\Delta \vdash T); \Gamma \vdash_1 t : T'\end{array}}{\Psi; \Gamma \vdash_1 \mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t : T'}$$

$$\frac{\Psi \vdash_1 \Gamma \qquad \Psi, g : \mathsf{Ctx}; \Gamma \vdash_1 t : T}{\Psi; \Gamma \vdash_1 \Lambda g.t : (g : \mathsf{Ctx}) \Rightarrow T} \qquad \frac{\Psi; \Gamma \vdash_1 t : (g : \mathsf{Ctx}) \Rightarrow T \qquad \Psi \vdash_0 \Delta}{\Psi; \Gamma \vdash_1 t\ \$\ \Delta : T[\Delta/g]}$$

$\boxed{\Psi; \Gamma \vdash_i \delta : \Delta}$ Regular substitution δ substitutes Δ for Γ in meta-context Ψ at layer i.

$$\frac{\Psi \vdash_i \Gamma}{\Psi; \Gamma \vdash_i \cdot : \cdot} \qquad \frac{\Psi \vdash_i g, \Gamma}{\Psi; g, \Gamma \vdash_i \mathsf{wk} : g} \qquad \frac{\Psi; \Gamma \vdash_i \delta : \Delta \qquad \Psi; \Gamma \vdash_i t : T}{\Psi; \Gamma \vdash_i \delta, t/x : \Delta, x : T}$$

Figure 4.3: Typing rules for layered modal type theory

Proof. Induction on the derivations at layer 0. ☐

The following is the syntax of terms:

$$\begin{array}{rcll}
\delta & := & \cdot \mid \mathsf{wk} \mid \delta, t/x & \text{(Regular Substitutions)} \\
s, t & := & x \mid u^\delta & \text{(Terms)} \\
& & \mid \mathsf{zero} \mid \mathsf{succ}\ t & \text{(Natural Numbers)} \\
& & \mid \lambda x.t \mid s\ t & \text{(Functions)} \\
& & \mid \mathtt{box}\ t \mid \mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t \mid \mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b} & \text{(Box)} \\
& & \mid \Lambda g.t \mid t\ \$\ \Gamma & \text{(Meta-functions)} \\
b & := & \mathsf{var}_x \Rightarrow t \mid \mathsf{genvar}_{g,T} \Rightarrow t \mid \mathsf{zero} \Rightarrow t \mid \mathsf{succ}\ ?u \Rightarrow t & \text{(Branches)} \\
& & \mid \lambda x.?u \Rightarrow t \mid ?u\ ?u' \Rightarrow t &
\end{array}$$

Natural numbers are constructed by zero and succ t. A recursor for Nat is possible, but it is deferred to Appendix D for conciseness. Function abstractions and applications are

$\boxed{\Psi; \Gamma \vdash_i t \approx t' : T}$ Term t and t' are equivalent in contexts Ψ and Γ at layer i

$$\frac{\Psi; \Gamma, x : S \vdash_1 t : T \qquad \Psi; \Gamma \vdash_1 s : S}{\Psi; \Gamma \vdash_1 (\lambda x.t)\ s \approx t[s/x] : T} \qquad \frac{\Psi; \cdot \vdash_0 s : T \qquad \Psi, u : T; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \texttt{letbox}\ u \leftarrow \texttt{box}\ s\ \texttt{in}\ t \approx t[s/u] : T'}$$

$$\frac{\Psi; \Gamma \vdash_1 t : S \longrightarrow T}{\Psi; \Gamma \vdash_1 t \approx \lambda x.(t\ x) : S \longrightarrow T} \qquad \frac{\Psi \vdash_1 \Gamma \qquad \Psi, g : \mathsf{Ctx}; \Gamma \vdash_1 t : T \qquad \Psi \vdash_0 \Delta}{\Psi; \Gamma \vdash_1 (\Lambda g.t)\ \$\ \Delta \approx t[\Delta/g] : T[\mathsf{id}_\Psi, \Delta/g]}$$

$$\frac{\Psi; \Gamma \vdash_1 t : (g : \mathsf{Ctx}) \Rightarrow T}{\Psi; \Gamma \vdash_1 t \approx \Lambda g.(t\ \$\ g) : (g : \mathsf{Ctx}) \Rightarrow T}$$

Figure 4.4: Equivalence judgment

standard. The term `box` t constructs a contextual type. There are two possible ways to eliminate a piece of code. Code composition and running are achieved by `letbox`, which binds code to a meta-variable. Pattern matching on code analyzes the form of the syntax of a given piece of code, and chooses the right branch to continue. Since I am building a type theory, pattern matching must be covering. The coverage of pattern matching can be checked depending on the type of the scrutinee. The discussion on pattern matching and its coverage is deferred to Sec. 4.3. Finally, meta-functions are similar to functions, with the only difference in the context arguments. Unlike a regular variable, each use of a meta-variable u requires a local substitution δ to replace the open variables with concrete terms.

Following the well-formedness of types, all the terms related to meta-programming can only be well-typed at layer 1. For example, for the introduction rule for contextual types,

$$\frac{\Psi \vdash_1 \Gamma \qquad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \texttt{box}\ t : \square(\Delta \vdash T)}$$

`box` t is well-typed at layer 1, only if the code t is well-typed at layer 0. Now a clear line is drawn between code and programs: code lives at layer 0 while programs live at layer 1. There are two elimination principles for contextual types: `letbox` and pattern matching on code. The discussion on pattern matching is postponed to the next section

(Sec. 4.3).

The typing judgments are defined in Fig. 4.3 and demonstrate how layering controls well-typed terms at each layer. The rules for terms coming from STLC, i.e. zero, succ, λ and function applications, are standard and are stated generically using the layer i. In particular this means that the syntax of terms from STLC is identical for both code and programs. However, the layer i in the typing judgment determines whether a given term of type $\mathtt{Nat}$ or a function is code or a program. The well-formedness of contexts are added to the premises of the regular variable rule and the zero rule to enforce the coherence between terms and types at layer i. Note that terms from STLC can extend the regular context via λ regardless of layers and they can only refer to but not introduce meta-variables. When referring to a meta-variable u, a regular substitution δ is needed to replace all variables in the regular context Δ, as specified by the superscript. The coherence between terms and types requires terms at layer i to have types at the same layer (Lemma 4.5).

The typing judgment for regular substitutions $\Psi; \Gamma \vdash_i \delta : \Delta$ is also layered similar to the typing of terms. The case for wk requires the regular context to be (g, Γ), so that the base case of the regular context must be a context variable. Due to the typing judgments, it is not possible to compose wk with $\cdot$, i.e.

$$\mathsf{wk} \circ (\cdot, t_1/x_1, \cdots, t_n/x_n)$$

is not a valid substitution, though $\cdot \circ \mathsf{wk}$ is valid and computes to $\cdot$. The substitution operation in layered modal type theory is intuitive and is defined in Appendix C for completeness.

Due to layering in the typing rules, the equivalence rules are also layered. One distinguished feature of layered modal type theory is that its computational behaviors are controlled by the layering index. There are three groups of equivalence rules: the PER rules which include symmetry and transitivity, congruence rules which are naturally derived from the typing rules, and the computation rules which describe β and η equivalence. The PER and congruence rules apply to all layers, but the computation rules *only* apply to layer 1. The β and η rules are in Fig. 4.4. $[s/x]$ and $[s/u]$ are regular and meta-substitutions, respectively. They substitute s for x and for u everywhere as expected (see Appendix C). The lack of computation at layer 0 ensures that terms at layer 0 are identified *only* by their syntactic structures and indeed behave as code:

Lemma 4.2 (Static code)**.** *If $\Psi; \Gamma \vdash_0 t \approx s : T$, then $t = s$.*

Proof. Induction on the derivation. $\square$

The static code lemma is another guiding lemma which characterizes the matryoshka principle. It implies that the congruence for `box` naturally derived from its typing rule simply meets the expectation:

$$\frac{\Psi \vdash_1 \Gamma \qquad \Psi; \Delta \vdash_0 t \approx t' : T}{\Psi; \Gamma \vdash_1 \texttt{box}\ t \approx \texttt{box}\ t' : \square(\Delta \vdash T)}$$

Due to the static code lemma, $t = t'$, so there is no interesting behavior under `box`. The static status of code safely enables pattern matching on code, which I will discuss in the next section.

Similar to types, the lifting lemma for terms also gives a formal account for the matryoshka principle, which states that terms and regular substitutions at layer 0 are subsumed by layer 1:

Lemma 4.3 (Lifting for terms and regular substitutions)**.**

- *If $\Psi; \Gamma \vdash_0 t : T$, then $\Psi; \Gamma \vdash_1 t : T$.*
- *If $\Psi; \Gamma \vdash_0 \delta : \Delta$, then $\Psi; \Gamma \vdash_1 \delta : \Delta$.*

Proof. Induction on the derivations at layer 0. $\square$

Though a term at layer 0 is code and static, its computational behaviors are recovered by lifting it to layer 1. The lifting lemma is what enables code running, which is crucial for a meta-programming system. The term lifting behavior can be triggered by the β rule for $\square$. For some well-typed terms t and s at layer 0 and a regular substitution δ that does not refer to u:

$$\texttt{letbox}\ u \leftarrow \texttt{box}\ ((\lambda x.t)\ s)\ \texttt{in}\ u^{\delta} \approx ((\lambda x.t)\ s)[\delta] = (\lambda x.t[\delta, x/x])\ (s[\delta]) \approx t[\delta, s[\delta]/x]$$

Due to the β rule, u is replaced by $(\lambda x.t)\ s$. The layer-0 term $(\lambda x.t)\ s$ is then lifted to layer 1 on the right hand side and computes. Thus its computational behavior is revived and it is further reduced after δ is applied.

4.3 Pattern Matching on Code

The static code lemma has confirmed the static status of code. Therefore, it is possible to intensionally analyze the structure of code. Pattern matching on code ($\mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b}$) is another elimination form of $\square(\Gamma \vdash T)$, where $\overrightarrow{b}$ is a list of *all possible branches* of t. The branches only need to match terms in STLC (from layer 0) because pattern matching is only available at layer 1 and the scrutinee is code from layer 0. Nested patterns like $(\lambda y.?u)\ ?u'$ are not directly supported to keep the system simple, but they can be encoded in multiple nested pattern matching expressions. Supporting a general recursion principle for code (e.g. (Pientka et al., 2019; Hu et al., 2022)) would require type polymorphism in addition to context variables. Type polymorphism is needed to handle the case for λ, where the type of the function body and the type of the function itself cannot be the same. However, this limitation comes from the syntactic theory for simple types. Later in Chapter 5, type polymorphism is introduced due to dependent types, so that a general recursion principle is possible. This chapter focuses on setting up a basis for syntax and semantics to be extended in Chapter 5.

In addition to the expected branches, e.g. $\mathsf{var}_x \Rightarrow t$ for code of the variable x and $\mathsf{zero} \Rightarrow t$ for the code of zero, I introduce a special branch for generated variables $\mathsf{genvar}_{g,T} \Rightarrow t$, which is required if the regular context of the code contains the context variable g. This special branch is expanded to multiple regular variable branches $\mathsf{var}_x \Rightarrow t$ if the context variable g is instantiated to a concrete local context after a meta-function application $t\ \$\ \Delta'$. Concretely, if g is instantiated to some context Δ', then $\mathsf{genvar}_{g,T} \Rightarrow t$ is expanded to multiple instances of $\mathsf{var}_x \Rightarrow t$ for all $x : T \in \Delta'$. In this way, I ensure that the coverage of pattern matching on code is stable under meta-substitutions even with the presence of context variables. If the regular context of the code contains no context variable at all, then no branch for generated variables is allowed. This action is concretely defined at the end of Appendix C.

There are two additional judgments for typing pattern matching on code.

$\boxed{\Psi; \Gamma \vdash_i t : T}$ Term t has type T in contexts Ψ and Γ at layer i where $i \in [0, 1]$

$$\frac{\Psi \vdash_1 T' \qquad \Psi; \Gamma \vdash_1 s : \Box(\Delta \vdash T) \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'}{\Psi; \Gamma \vdash_1 \mathsf{match}\ s\ \mathsf{with}\ \overrightarrow{b} : T'}$$

$\boxed{\Psi; \Gamma \vdash_1 b : \Delta \vdash T \Rightarrow T'}$ b is a branch of type T' w.r.t. a code of type T open in Δ.

$$\frac{\Psi \vdash_0 g, \Delta \qquad \Psi; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{genvar}_{g,T} \Rightarrow t : g, \Delta \vdash T \Rightarrow T'} \qquad \frac{\Psi \vdash_0 \Delta \qquad x : T \in \Delta \qquad \Psi; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{var}_x \Rightarrow t : \Delta \vdash T \Rightarrow T'}$$

$$\frac{\Psi \vdash_0 \Delta \qquad \Psi; \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{zero} \Rightarrow t : \Delta \vdash \mathtt{Nat} \Rightarrow T'} \qquad \frac{\Psi, u : (\Delta \vdash \mathtt{Nat}); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{succ}\ ?u \Rightarrow t : \Delta \vdash \mathtt{Nat} \Rightarrow T'}$$

$$\frac{\Psi, u : (\Delta, x : S \vdash T); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \lambda x.?u \Rightarrow t : \Delta \vdash S \longrightarrow T \Rightarrow T'}$$

$$\frac{\forall\ \Psi \vdash_0 S\ .\ \Psi, u : (\Delta \vdash S \longrightarrow T), u' : (\Delta \vdash S); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 ?u\ ?u' \Rightarrow t : \Delta \vdash T \Rightarrow T'}$$

$\boxed{\Psi; \Gamma \vdash_i t \approx t' : T}$ Terms t and t' are equivalent (β rules for match)

$$\frac{x : T \in \Delta \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T' \qquad \overrightarrow{b}(x) = \mathsf{var}_x \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathtt{box}\ x\ \mathsf{with}\ \overrightarrow{b} \approx t : T'}$$

$$\frac{\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash \mathtt{Nat} \Rightarrow T' \qquad \overrightarrow{b}(\mathsf{zero}) = \mathsf{zero} \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathtt{box}\ \mathsf{zero}\ \mathsf{with}\ \overrightarrow{b} \approx t : T'}$$

$$\frac{\Psi; \Delta \vdash_0 s : \mathtt{Nat} \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash \mathtt{Nat} \Rightarrow T' \qquad \overrightarrow{b}(\mathsf{succ}\ s) = \mathsf{succ}\ ?u \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathtt{box}\ (\mathsf{succ}\ s)\ \mathsf{with}\ \overrightarrow{b} \approx t[s/u] : T'}$$

$$\frac{\Psi; \Delta, x : S \vdash_0 s : T \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash S \longrightarrow T \Rightarrow T' \qquad \overrightarrow{b}(\lambda x.s) = \lambda x.?u \Rightarrow t}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathtt{box}\ (\lambda x.s)\ \mathsf{with}\ \overrightarrow{b} \approx t[s/u] : T'}$$

$$\frac{\begin{array}{c}\Psi; \Delta \vdash_0 t : S \longrightarrow T \\ \Psi; \Delta \vdash_0 s : S \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T' \qquad \overrightarrow{b}(t\ s) = ?u\ ?u' \Rightarrow t\end{array}}{\Psi; \Gamma \vdash_1 \mathsf{match}\ \mathtt{box}\ (t\ s)\ \mathsf{with}\ \overrightarrow{b} \approx t[t/u, s/u'] : T'}$$

Figure 4.5: judgments for pattern matching and branches

- $\Psi; \Gamma \vdash_1 b : \Delta \vdash T \Rrightarrow T'$ is used for pattern matching and checks a branch b in $\Psi; \Gamma$ given $\square(\Delta \vdash T)$ as the type of the scrutinee and T' as the return type.
- The typing rule for match uses the judgment $\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rrightarrow T'$. This judgment is a *covering* generalization of the previous judgment for a single branch and checks all branches in $\overrightarrow{b}$.

In the second judgment, coverage means that $\overrightarrow{b}$ contains all possible branches to handle a scrutinee of type $\square(\Delta \vdash T)$. There are many possible ways to ensure coverage e.g. (Pientka and Abel, 2015). Fig. 4.6 gives one possible solution, which only uses syntax of types to check coverage. Due to layering, code must be in STLC and therefore coverage is guaranteed by the typing information. If T is `Nat`, then there must be branches for zero and succ, but not λ due to type mismatch. A function application could return a `Nat` so a branch for application is necessary. Moreover, there is one branch for each variable bound to `Nat` in Δ. Contrarily, if T is a function, then all constructor branches for `Nat` can be safely excluded. In addition, if the regular context of the code being analyzed contains a context variable, then $\overrightarrow{b}$ must in addition include a branch for generated variables $b_{\textsf{genvar}}$ to handle the instantiation of context variables. This condition is checked by the auxiliary judgment $\Psi; \Gamma \vdash_1 \overrightarrow{b}_{\textsf{var}} : \Delta \vdash T \Rrightarrow T'$, where $\overrightarrow{b}_{\textsf{var}}$ are all possible branches for variables for code of type $\square(\Delta \vdash T)$.

The two new judgments only live at layer 1, the only layer where pattern matching on code occurs.

The rules related to pattern matching on code are listed in Fig. 4.5. All typing rules for individual branches are similar. For example, if the pattern is $\lambda x.?u$, then u captures the body of some λ. The branch body t is checked with u bound to $(\Delta, x : S \vdash T)$, which has a larger regular context than Δ. If the branch matches a function application, the premise requires t is well-typed for all $\Psi \vdash_0 S$. This universal quantification should be read as a higher-order derivation that applies for all $\Psi \vdash_0 S$ (see also (Zeilberger, 2008)) and S is abstracted as a parameter.

The bottom of Fig. 4.5 are the β rules for pattern matching. Based on the structure of the scrutinee, the pattern matcher dispatches to the right branch and propagates instantiations for pattern variables via meta-substitutions to the bodies. Notations like $\overrightarrow{b}(\textsf{succ}\ s)$ denote the lookup of $\overrightarrow{b}$ based on a given shape. For example, $\overrightarrow{b}(\textsf{succ}\ s) =$

$\boxed{\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'}$ Branches of $\overrightarrow{b}$ for a scrutinee of type $\Box(\Delta \vdash T)$ all have type T' and are convering.

$$\frac{\begin{array}{c} \forall\, b \in \overrightarrow{b}\ .\ \Psi; \Gamma \vdash_1 b : \Delta \vdash \texttt{Nat} \Rightarrow T' \qquad b_{\mathsf{zero}} = \mathsf{zero} \Rightarrow t \text{ for some } t \\ b_{\mathsf{succ}} = \mathsf{succ}\ ?u \Rightarrow t \text{ for some } t \qquad b_{\mathsf{rec}} = \mathsf{rec}_{T'}\ ?u\ (x\ y.?u')\ ?u'' \Rightarrow t \text{ for some } t \\ b_{\mathsf{app}} = ?u\ ?u' \Rightarrow t \text{ for some } t \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b}_{\mathsf{var}} : \Delta \vdash \texttt{Nat} \Rightarrow T' \\ \overrightarrow{b} \text{ is a permutation of } \{b_{\mathsf{zero}}, b_{\mathsf{succ}}, b_{\mathsf{rec}}, b_{\mathsf{app}}, b \text{ for all } b \in \overrightarrow{b}_{\mathsf{var}}\} \end{array}}{\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash \texttt{Nat} \Rightarrow T'}$$

$$\frac{\begin{array}{c} \forall\, b \in \overrightarrow{b}\ .\ \Psi; \Gamma \vdash_1 b : \Delta \vdash S \longrightarrow T \Rightarrow T' \\ b_\lambda = \lambda x.?u \Rightarrow t \text{ for some } t \qquad b_{\mathsf{rec}} = \mathsf{rec}_{T'}\ ?u\ (x\ y.?u')\ ?u'' \Rightarrow t \text{ for some } t \\ b_{\mathsf{app}} = ?u\ ?u' \Rightarrow t \text{ for some } t \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b}_{\mathsf{var}} : \Delta \vdash S \longrightarrow T \Rightarrow T' \\ \overrightarrow{b} \text{ is a permutation of } \{b_\lambda, b_{\mathsf{rec}}, b_{\mathsf{app}}, b \text{ for all } b \in \overrightarrow{b}_{\mathsf{var}}\} \end{array}}{\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash S \longrightarrow T \Rightarrow T'}$$

$\boxed{\Psi; \Gamma \vdash_1 \overrightarrow{b}_{\mathsf{var}} : \Delta \vdash T \Rightarrow T'}$ All variable branches $\overrightarrow{b}_{\mathsf{var}}$ for a scrutinee of type $\Box(\Delta \vdash T)$ have type T'.

$$\frac{\begin{array}{c} \forall\, b \in \overrightarrow{b}_{\mathsf{var}}\ .\ \Psi; \Gamma \vdash_1 b : \Delta \vdash T \Rightarrow T' \qquad \forall\, x : T \in \Delta\ .\ b_x = \mathsf{var}_x \Rightarrow t \text{ for some } t \\ \overrightarrow{b} \text{ is a permutation of } \{b_x \text{ for all } x : T \in \Delta\} \qquad \Delta \text{ contains no context variable} \end{array}}{\Psi; \Gamma \vdash_1 \overrightarrow{b}_{\mathsf{var}} : \Delta \vdash T \Rightarrow T'}$$

$$\frac{\begin{array}{c} \forall\, b \in \overrightarrow{b}_{\mathsf{var}}\ .\ \Psi; \Gamma \vdash_1 b : g, \Delta \vdash T \Rightarrow T' \\ \forall\, x : T \in g, \Delta\ .\ b_x = \mathsf{var}_x \Rightarrow t \text{ for some } t \qquad b_{\mathsf{genvar}} = \mathsf{genvar}_{g,T} \Rightarrow t \text{ for some } t \\ \overrightarrow{b} \text{ is a permutation of } \{b_{\mathsf{genvar}}, b_x \text{ for all } x : T \in g, \Delta\} \end{array}}{\Psi; \Gamma \vdash_1 \overrightarrow{b}_{\mathsf{var}} : g, \Delta \vdash T \Rightarrow T'}$$

Figure 4.6: Covering judgment for all branches

$\mathsf{succ}\ ?u \Rightarrow t$ means that the lookup of $\mathsf{succ}\ s$ in $\overrightarrow{b}$ finds the branch $\mathsf{succ}\ ?u \Rightarrow t$. Then s is meant to substitute u in t. This lookup is guaranteed to succeed because $\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'$ is covering. Note that there does not exist a β rule for the branch $\mathsf{genvar}_{g,T} \Rightarrow t$, because there does not exist a representation for a variable in a context variable. Code of the form `box` x necessarily refers to a variable x concretely bound in the regular context. The body t of $\mathsf{genvar}_{g,T} \Rightarrow t$ is only invoked when g is instantiated to some Γ by a meta-substitution (likely due to a meta-function application), and subsequently code `box` y for $y : T \in \Gamma$ is matched. At this point, the meta-substitution action has already created a branch for y, $\mathsf{var}_y \Rightarrow t$. Thus, a β rule for $\mathsf{genvar}_{g,T} \Rightarrow t$ is not present at all.

Finally, the congruence of pattern matching on code relies on the congruence of all branches, which is formulated by

$$\Psi; \Gamma \vdash_1 b \approx b' : \Delta \vdash T \Rightarrow T'$$

and

$$\Psi; \Gamma \vdash_1 \overrightarrow{b} \approx \overrightarrow{b}' : \Delta \vdash T \Rightarrow T'$$

They only include congruence rules so I omit their definitions here.

4.4 Syntactic Properties

In this section, I list a few important syntactic properties of layered modal type theory. The first two properties are presupposition. They verify the syntactic validity of the well-formedness and typing judgments.

Lemma 4.4 (Presupposition of regular contexts and types)**.**

- *If* $\Psi \vdash_i \Gamma$*, then* $\vdash \Psi$*.*
- *If* $\Psi \vdash_i T$*, then* $\vdash \Psi$*.*

Lemma 4.5 (Presupposition of typing)**.**

- *If* $\Psi; \Gamma \vdash_i t : T$*, then* $\Psi \vdash_i \Gamma$ *and* $\Psi \vdash_i T$*.*
- *If* $\Psi; \Gamma \vdash_i \delta : \Delta$*, then* $\Psi \vdash_i \Gamma$ *and* $\Psi \vdash_i \Delta$*.*

The regular substitution lemma proves the stability of terms under regular substitutions and that of regular substitutions under composition.

Lemma 4.6 (Regular substitutions)**.**

- *If* $\Psi; \Gamma \vdash_i t : T$ *and* $\Psi; \Delta \vdash_i \delta : \Gamma$*, then* $\Psi; \Delta \vdash_i t[\delta] : T$*.*

- *If* $\Psi; \Gamma \vdash_i \delta : \Delta$ *and* $\Psi; \Gamma' \vdash_i \delta' : \Gamma$*, then* $\Psi; \Gamma' \vdash_i \delta \circ \delta' : \Delta$*.*

Due to dual contexts in typing judgments, meta-substitutions are introduced to substitute meta-and context variables with code and regular contexts, respectively. The syntax of meta-substitutions is defined as follows:

$$\sigma \quad := \cdot \mid \sigma, t/u \mid \sigma, \Gamma/g \qquad \text{(Meta-substitutions)}$$

The most important cases for applying a meta-substitution are the case for local contexts, where a context variable g is replaced by a concrete context, and the case for meta-variables, where a meta-variable u is replaced by a concrete term:

$$\begin{aligned} g[\sigma] &:= \sigma(g) && \text{(lookup } g \text{ in } \sigma) \\ \Gamma, x : T[\sigma] &:= (\Gamma[\sigma]), x : (T[\sigma]) \\ u^{\delta}[\sigma] &:= \sigma(u)[\delta[\sigma]] && \text{(lookup of } u \text{ in } \sigma) \\ \texttt{letbox}\ u \leftarrow s\ \texttt{in}\ t[\sigma] &:= \texttt{letbox}\ u \leftarrow s[\sigma]\ \texttt{in}\ (t[\sigma, u^{\mathsf{id}}/u]) \end{aligned}$$

In particular, in the case for meta-variables, a meta-substitution triggers a local substitution as well, so the action of regular substitutions must be defined prior to that of meta-substitutions. The properties of regular substitutions must also be proved first.

The meta-substitution lemma confirms that terms and regular substitutions are stable under meta-substitutions. Since meta-substitutions also replace context variables, they propagate under local contexts and types.

Lemma 4.7 (Meta-substitutions)**.**

- *If* $\Psi \vdash_i \Gamma$ *and* $\Psi' \vdash \sigma : \Psi$*, then* $\Psi' \vdash_i \Gamma[\sigma]$*.*

- *If* $\Psi \vdash_i T$ *and* $\Psi' \vdash \sigma : \Psi$*, then* $\Psi' \vdash_i T[\sigma]$*.*

- *If* $\Psi; \Gamma \vdash_i t : T$ *and* $\Psi' \vdash \sigma : \Psi$*, then* $\Psi'; \Gamma[\sigma] \vdash_i t[\sigma] : T[\sigma]$*.*

- *If* $\Psi; \Gamma \vdash_i \delta : \Delta$ *and* $\Psi' \vdash \sigma : \Psi$*, then* $\Psi'; \Gamma[\sigma] \vdash_i \delta[\sigma] : \Delta[\sigma]$*.*

Meta-and regular substitutions interact with each other. The following lemma states that meta-substitutions distribute under terms and local substitutions.

Lemma 4.8 (Distributivity of meta-substitutions)**.**

- *If* $\Psi; \Gamma \vdash_i t : T$*,* $\Psi; \Delta \vdash_i \delta : \Gamma$ *and* $\Phi \vdash \sigma : \Psi$*, then* $t[\delta][\sigma] = (t[\sigma][\delta[\sigma]])$*.*
- *If* $\Psi; \Gamma \vdash_i \delta : \Delta$*,* $\Psi; \Gamma' \vdash_i \delta' : \Gamma$ *and* $\Phi \vdash \sigma : \Psi$*, then* $(\delta \circ \delta')[\sigma] = (\delta[\sigma]) \circ (\delta'[\sigma])$*.*

Similar lemmas are also proved for equivalence judgments. First, presupposition confirms the syntactic validity of equivalence judgments.

Lemma 4.9 (Presupposition of equivalence)**.**

- *If* $\Psi; \Gamma \vdash_1 t \approx t' : T$*, then* $\Psi; \Gamma \vdash_1 t : T$ *and* $\Psi; \Gamma \vdash_1 t' : T$*.*
- *If* $\Psi; \Gamma \vdash_1 \delta \approx \delta' : \Delta$*, then* $\Psi; \Gamma \vdash_1 \delta : \Delta$ *and* $\Psi; \Gamma \vdash_1 \delta' : \Delta$*.*

Then, equivalence judgments are stable under both regular and meta-substitutions. Note that meta-substitutions still propagate under regular contexts and types.

Lemma 4.10 (Regular substitutions)**.**

- *If* $\Psi; \Gamma \vdash_1 t \approx t' : T$ *and* $\Psi; \Delta \vdash_1 \delta \approx \delta' : \Gamma$*, then* $\Psi; \Delta \vdash_1 t[\delta] \approx t'[\delta'] : T$*.*
- *If* $\Psi; \Gamma \vdash_1 \delta'' \approx \delta''' : \Delta'$ *and* $\Psi; \Delta \vdash_1 \delta \approx \delta' : \Gamma$*, then* $\Psi; \Delta \vdash_1 \delta'' \circ \delta \approx \delta''' \circ \delta' : \Delta'$*.*

Lemma 4.11 (Meta-substitutions)**.**

- *If* $\Psi; \Gamma \vdash_1 t \approx t' : T$ *and* $\Phi \vdash \sigma : \Psi$*, then* $\Phi; \Gamma[\sigma] \vdash_1 t[\sigma] \approx t'[\sigma] : T[\sigma]$*.*
- *If* $\Psi; \Gamma \vdash_1 \delta \approx \delta' : \Delta$ *and* $\Phi \vdash \sigma : \Psi$*, then* $\Phi; \Gamma[\sigma] \vdash_1 \delta[\sigma] \approx \delta'[\sigma] : \Delta[\sigma]$*.*

4.5 Weak-head Reduction

In the previous sections, I have finished the discussion about the syntax and judgments of 2-layered modal type theory. From this section, I begin the discussion about weak normalization and the decidability of convertibility. In this chapter, instead of untyped domain models used in Part I, I follow Abel et al. (2018) and use Kripke reducibility predicates to prove weak normalization. The weak normalization algorithm is implemented by a series of weak-head reductions, which are defined in this section. The Kripke reducibility predicates model terms at each layer (Sec. 4.7 and 4.8), where the

$\boxed{t \rightsquigarrow t'}$ t reduces to t'.

$$\frac{t \rightsquigarrow t'}{t\ s \rightsquigarrow t'\ s} \qquad \frac{s \rightsquigarrow s'}{\texttt{letbox}\ u \leftarrow s\ \texttt{in}\ t \rightsquigarrow \texttt{letbox}\ u \leftarrow s'\ \texttt{in}\ t} \qquad \frac{t \rightsquigarrow t'}{t\ \$\ \Delta \rightsquigarrow t'\ \$\ \Delta}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b} \rightsquigarrow \mathsf{match}\ t'\ \mathsf{with}\ \overrightarrow{b}} \qquad \frac{}{(\lambda x.t)\ s \rightsquigarrow t[s/x]} \qquad \frac{}{(\Lambda g.t)\ \$\ \Delta \rightsquigarrow t[\Delta/g]}$$

$$\frac{}{\texttt{letbox}\ u \leftarrow \texttt{box}\ s\ \texttt{in}\ t \rightsquigarrow t[s/u]} \qquad \frac{\overrightarrow{b}(x) = \mathsf{var}_x \Rightarrow t}{\mathsf{match}\ \texttt{box}\ x\ \mathsf{with}\ \overrightarrow{b} \rightsquigarrow t}$$

$$\frac{\overrightarrow{b}(\mathsf{zero}) = \mathsf{zero} \Rightarrow t}{\mathsf{match}\ \texttt{box}\ \mathsf{zero}\ \mathsf{with}\ \overrightarrow{b} \rightsquigarrow t} \qquad \frac{\overrightarrow{b}(\mathsf{succ}\ t) = \mathsf{succ}\ ?u \Rightarrow t'}{\mathsf{match}\ \texttt{box}\ (\mathsf{succ}\ t)\ \mathsf{with}\ \overrightarrow{b} \rightsquigarrow t'[t/u]}$$

$$\frac{\overrightarrow{b}(\lambda x.t) = \lambda x.?u \Rightarrow t'}{\mathsf{match}\ \texttt{box}\ (\lambda x.t)\ \mathsf{with}\ \overrightarrow{b} \rightsquigarrow t'[t/u]} \qquad \frac{\overrightarrow{b}(t\ s) = ?u\ ?u' \Rightarrow t'}{\mathsf{match}\ \texttt{box}\ (t\ s)\ \mathsf{with}\ \overrightarrow{b} \rightsquigarrow t'[t/u, s/u']}$$

$\boxed{\Psi; \Gamma \vdash_1 t \rightsquigarrow t' : T}$ t of type T reduces to t'.

$$\frac{\Psi; \Gamma \vdash_1 t : T \qquad t \rightsquigarrow^* t'}{\Psi; \Gamma \vdash_1 t \rightsquigarrow t' : T}$$

Figure 4.7: One-step reduction

logical relations at layer 1 capture structure of syntax semantically. In the Kripke reducibility predicates, the semantics of terms are given by what weak-head normal forms they reduce to. The predicates relate two terms, if they reduce to weak-head normal forms, the sub-terms of which are recursively related. This setup allows us to prove theorems like syntactically equivalent terms must reduce to related weak-head normal forms. The matryoshka principle in the semantics is characterized by the layering restriction lemma (Lemma 4.21), which relates the logical relations at both layers. The layering restriction lemma is what distinguishes the layered style from the heterogeneous style, where models of different languages may not necessarily find definitive relations. Finally the fundamental theorem (Sec. 4.9) confirms that well-typed terms always reduce to weak-head normal forms. Applying the fundamental theorem to the convertibility algorithm (Sec. 4.10) proves the decidability of convertibility. This process not only concludes the investigation on 2-layered modal type theory, but also is steps to follow in the next chapter, where dependent types are handled.

The following define the syntax of weak-head normal and neutral forms:

$$\begin{array}{rll} w := & v \mid \mathsf{zero} \mid \mathsf{succ}\ t \mid \mathsf{box}\ t \mid \lambda x.t \mid \Lambda g.t & \text{(Weak-head normal form)} \\ v := & x \mid u^{\delta} \mid v\ t \mid \mathtt{letbox}\ u \leftarrow v\ \mathtt{in}\ t \mid v\ \$\ \Gamma & \text{(Neutral form)} \\ & \mid \mathsf{match}\ v\ \mathsf{with}\ \overrightarrow{b} \mid \mathsf{match}\ \mathsf{box}\ u^{\delta}\ \mathsf{with}\ \overrightarrow{b} & \end{array}$$

The definition of weak-head reductions is given in Fig. 4.7. Weak-head reduction only considers terms at layer 1 as this is the only layer for computation. One-step reductions generalize to multi-step naturally by taking reflexive-transitive closures. The multi-step variants are denoted as

$$t \rightsquigarrow^* t'$$

and

$$\Psi; \Gamma \vdash_1 t \rightsquigarrow^* t' : T$$

The syntactic properties for reductions are simple to prove as reductions are just a sub-relation of the equivalence judgment for terms.

Theorem 4.12 (Preservation). *If $\Psi; \Gamma \vdash_1 t \rightsquigarrow t' : T$, then $\Psi; \Gamma \vdash_1 t' : T$.*

Lemma 4.13 (Regular substitutions). *If $\Psi; \Gamma \vdash_1 t \rightsquigarrow t' : T$ and $\Psi; \Delta \vdash_1 \delta : \Gamma$, then $\Psi; \Delta \vdash_1 t[\delta] \rightsquigarrow t'[\delta] : T$.*

Lemma 4.14 (Meta-substitutions). *If $\Psi; \Gamma \vdash_1 t \rightsquigarrow t' : T$ and $\Phi \vdash \sigma : \Psi$, then $\Phi; \Gamma[\sigma] \vdash_1 t[\sigma] \rightsquigarrow t'[\sigma] : T[\sigma]$.*

The uniqueness lemma proves that the one-step reduction relation is functional and thus deterministic, i.e. the same input must lead to the same output.

Lemma 4.15 (Uniqueness). *If $t \rightsquigarrow t'$ and $t \rightsquigarrow t''$, then $t' = t''$.*

4.6 Generic Equivalence

In this section, I follow Abel et al. (2018) to define a modular generic equivalence, which will be instantiated by the convertibility checking derivation later (Sec. 4.10). In

$\boxed{\gamma : \Psi \Longrightarrow_g \Phi}$ γ is a meta-weakening.

$$\frac{\vdash \Psi}{\mathsf{id} : \Psi \Longrightarrow_g \Psi} \qquad \frac{\gamma : \Psi \Longrightarrow_g \Phi \qquad \Psi \vdash B}{p(\gamma) : \Psi, B \Longrightarrow_g \Phi} \qquad \frac{\gamma : \Psi \Longrightarrow_g \Phi \qquad \Phi \vdash B}{q(\gamma) : \Psi, B \Longrightarrow_g \Phi, B}$$

$\boxed{\tau : \Psi; \Gamma \Longrightarrow_i \Delta}$ τ is a regular weakening from Γ to Δ.

$$\frac{\Psi \vdash_i \Gamma}{\mathsf{id} : \Psi; \Gamma \Longrightarrow_i \Gamma} \qquad \frac{\tau : \Psi; \Gamma \Longrightarrow_i \Delta \qquad \Psi \vdash_i T}{p(\tau) : \Psi; \Gamma, x : T \Longrightarrow_i \Delta} \qquad \frac{\tau : \Psi; \Gamma \Longrightarrow_i \Delta \qquad \Psi \vdash_i T}{q(\tau) : \Psi; \Gamma, x : T \Longrightarrow_i \Delta, x : T}$$

Figure 4.8: Judgments for meta-and regular weakenings

principle, this generic equivalence is not exactly needed here: multiple instantiations are only needed for dependent types. I still choose to do it here to set up a framework for dependent types. There are two kinds of generic equivalence over terms:

- $\Psi; \Gamma \vdash_1 t \simeq t' : T$ describes a generic equivalence between two terms, and
- $\Psi; \Gamma \vdash_1 v \sim v' : T$ describes a generic equivalence between two neutral terms.

The subscript is fixed to be 1 because this is the only layer where computation occurs. Furthermore, $\Psi; \Gamma \vdash_1 t \simeq t' : T$ is generalized to $\Psi; \Gamma \vdash_1 \delta \simeq \delta' : \Delta$, which denotes a generic equivalence between regular substitutions. The definition is effectively just congruence:

$$\frac{\Psi \vdash_1 \Gamma}{\Psi; \Gamma \vdash_1 \cdot \simeq \cdot : \cdot} \qquad \frac{\Psi \vdash_1 g, \Gamma}{\Psi; g, \Gamma \vdash_1 \mathsf{wk} \simeq \mathsf{wk} : g} \qquad \frac{\Psi; \Gamma \vdash_1 \delta \simeq \delta' : \Delta \qquad \Psi; \Gamma \vdash_1 t \simeq t' : T}{\Psi; \Gamma \vdash_1 \delta, t/x \simeq \delta', t'/x : \Delta, x : T}$$

The generic equivalences and hence the Kripke reducibility predicates depend on weakenings of meta-and regular contexts. They are defined as follows:

$$\begin{array}{lll} \gamma & := \mathsf{id} \mid q(\gamma) \mid p(\gamma) & \text{(Meta-weakenings)} \\ \tau & := \mathsf{id} \mid q(\tau) \mid p(\tau) & \text{(Regular weakenings)} \end{array}$$

Their definitions are virtually identical and are routine. There are three cases, identity (id), dropping (p) and keeping (q). Technically, I should define how they react on types, terms, etc, but I do not focus too much on the actions of weakenings in this thesis as handling them are not interesting. Types, terms, etc. are automatically and implicitly weakened if their contexts are changed to weakened contexts. It should be obvious from the textual context. The typing judgments for weakenings are defined in Fig. 4.8. I also write meta-and regular weakenings as pairs when I weaken both meta-and local contexts at the same time:

$$\gamma; \tau :: \Psi; \Gamma \Longrightarrow \Phi; \Delta$$

The effect of a weakening pair is to apply γ first and then τ.

These two generic equivalences for terms must satisfy the following laws.

Law 4.1 (Subsumption).

- *If* $\Psi; \Gamma \vdash_1 v \sim v' : T$, *then* $\Psi; \Gamma \vdash_1 v \simeq v' : T$.
- *If* $\Psi; \Gamma \vdash_1 t \simeq t' : T$, *then* $\Psi; \Gamma \vdash_1 t \approx t' : T$.

The subsumption law of generic equivalence of terms implies the subsumption property of generic equivalence of regular substitutions:

Lemma 4.16 (Subsumption). *If* $\Psi; \Gamma \vdash_1 \delta \simeq \delta' : \Delta$, *then* $\Psi; \Gamma \vdash_1 \delta \approx \delta' : \Delta$.

Law 4.2 (PER). *Both generic equivalences for terms are PERs.*

Law 4.3 (Monotonicity). *Given* $\gamma; \tau :: \Psi; \Gamma \Longrightarrow \Phi; \Delta$, *if* $\Phi; \Delta \vdash_1 t \simeq t' : T$, *or* $\Phi; \Delta \vdash_1 v \sim v' : T$, *then* $\Psi; \Gamma \vdash_1 t \simeq t' : T$, *or* $\Psi; \Gamma \vdash_1 v \sim v' : T$, *respectively.*

Again monotonicity generalizes to regular substitutions:

Lemma 4.17 (Monotonicity). *Given* $\gamma; \tau :: \Psi; \Gamma \Longrightarrow \Phi; \Delta$, *if* $\Phi; \Delta \vdash_1 \delta \simeq \delta' : \Delta'$, *then* $\Psi; \Gamma \vdash_1 \delta \simeq \delta' : \Delta'$.

Law 4.4 (Weak-head closure). *If* $\Psi; \Gamma \vdash_1 t \rightsquigarrow^* w : T$, $\Psi; \Gamma \vdash_1 t' \rightsquigarrow^* w' : T$ *and* $\Psi; \Gamma \vdash_1 w \simeq w' : T$, *then* $\Psi; \Gamma \vdash_1 t \simeq t' : T$.

Law 4.5 (Congruence).

- *If* $\Psi \vdash_1 \Gamma$, *then* $\Psi; \Gamma \vdash_1 \textsf{zero} \simeq \textsf{zero} : \texttt{Nat}$.

- *If* $\Psi; \Gamma \vdash_1 t \simeq t' : \texttt{Nat}$*, then* $\Psi; \Gamma \vdash_1 \mathsf{succ}\ t \simeq \mathsf{succ}\ t' : \texttt{Nat}$*.*

- *If* $\Psi; \Gamma \vdash_1 t : S \longrightarrow T$*,* $\Psi; \Gamma \vdash_1 t' : S \longrightarrow T$ *and* $\Psi; \Gamma, x : S \vdash_1 t\ x \simeq t'\ x : T$*, then* $\Psi; \Gamma \vdash_1 t \simeq t' : S \longrightarrow T$*.*

- *If* $\Psi \vdash_1 \Gamma$ *and* $\Psi; \Delta \vdash_0 t : T$*, then* $\Psi; \Gamma \vdash_1 \texttt{box}\ t \simeq \texttt{box}\ t : T$*.*

- *If* $\Psi; \Gamma \vdash_1 t : (g : \mathsf{Ctx}) \Rightarrow T$*,* $\Psi; \Gamma \vdash_1 t' : (g : \mathsf{Ctx}) \Rightarrow T$ *and* $\Psi, g : \mathsf{Ctx}; \Gamma \vdash_1 t\ \$\ g \simeq t'\ \$\ g : T$*, then* $\Psi; \Gamma \vdash_1 t \simeq t' : (g : \mathsf{Ctx}) \Rightarrow T$*.*

Law 4.6 (Congruence of neutrals)**.**

- *If* $\Psi; \Gamma \vdash_1 x : T$*, then* $\Psi; \Gamma \vdash_1 x \sim x : T$*.*

- *If* $u : (\Delta \vdash T) \in \Psi$ *and* $\Psi; \Gamma \vdash_1 \delta \simeq \delta' : \Delta$*, then* $\Psi; \Gamma \vdash_1 u^{\delta} \sim u^{\delta'} : T$*.*

- *If* $\Psi; \Gamma \vdash_1 v \sim v' : S \longrightarrow T$ *and* $\Psi; \Gamma \vdash_1 t \simeq t' : S$*, then* $\Psi; \Gamma \vdash_1 v\ t \sim v\ t' : T$*.*

- *If* $\Psi \vdash_1 T'$*,* $\Psi; \Gamma \vdash_1 v \sim v' : \square(\Delta \vdash T)$ *and* $\Psi, u : (\Delta \vdash T); \Gamma \vdash_1 t \simeq t' : T'$*, then* $\Psi; \Gamma \vdash_1 \texttt{letbox}\ u \leftarrow v\ \texttt{in}\ t \sim \texttt{letbox}\ u \leftarrow v'\ \texttt{in}\ t' : T'$*.*

- *If* $\Psi; \Gamma \vdash_1 v \sim v' : (g : \mathsf{Ctx}) \Rightarrow T$ *and* $\Psi \vdash_0 \Delta$*, then* $\Psi; \Gamma \vdash_1 v\ \$\ \Delta \sim v'\ \$\ \Delta : T[\Delta/g]$

Law 4.7 (Congruence of neutral pattern matching on code)**.**

- *If* $\Psi \vdash_1 T'$*,* $\Psi; \Gamma \vdash_1 v \sim v' : \square(\Delta \vdash T)$ *and* $\Psi; \Gamma \vdash_1 \overrightarrow{b} \simeq \overrightarrow{b}' : \Delta \vdash T \Rightarrow T'$*, then* $\Psi; \Gamma \vdash_1 \mathsf{match}\ v\ \mathsf{with}\ \overrightarrow{b} \sim \mathsf{match}\ v'\ \mathsf{with}\ \overrightarrow{b}' : T'$*.*

- *If* $\Psi \vdash_1 T'$*,* $u \in (\Delta \vdash T) \in \Psi$*,* $\Psi; \Gamma \vdash_0 \delta : \Delta$ *and* $\Psi; \Gamma \vdash_1 \overrightarrow{b} \simeq \overrightarrow{b}' : \Delta \vdash T \Rightarrow T'$*, then* $\Psi; \Gamma \vdash_1 \mathsf{match}\ \texttt{box}\ u^{\delta}\ \mathsf{with}\ \overrightarrow{b} \sim \mathsf{match}\ \texttt{box}\ u^{\delta}\ \mathsf{with}\ \overrightarrow{b}' : T'$*.*

Here, $\Psi; \Gamma \vdash_1 \overrightarrow{b} \simeq \overrightarrow{b}' : \Delta \vdash T \Rightarrow T'$ *propagates generic equivalence between terms to matching branches. This judgment also requires coverage of branches.*

The congruence law of regular variables implies that regular weakening substitutions, specifically, regular identity substitutions, are reflexive in the generic equivalence:

Lemma 4.18 (Reflexivity of regular weakening substitutions)**.** *If* $\Psi \vdash_1 \Delta, \Gamma$*, then* $\Psi; \Delta, \Gamma \vdash_1 \mathsf{wk}_{\Delta} \simeq \mathsf{wk}_{\Delta} : \Delta$*.*

Lemma 4.19 (Reflexivity of regular identity substitutions). *If* $\Psi \vdash_1 \Gamma$*, then* $\Psi; \Gamma \vdash_1 \mathit{id}_\Gamma \simeq \mathit{id}_\Gamma : \Gamma$.

This further implies

Lemma 4.20 (Congruence of meta-variables). *If* $\vdash \Psi$ *and* $u : (\Gamma \vdash T) \in \Psi$*, then* $\Psi; \Gamma \vdash_1 u^{\mathit{id}_\Gamma} \sim u^{\mathit{id}_\Gamma} : T$.

4.7 Reducibility Predicates at Layer 0

In this section, I define multiple reducibility predicates for $i \in [0, 1]$:

- $\Psi \Vdash^i T$ denotes reducible type T at layer i.
- $\Psi; \Gamma \Vdash^i_1 t \approx t' : T$ relates two reducible terms t and t' at layer i.
- $\Psi \Vdash^i \Gamma$ extends reducible types to reducible contexts at layer i.
- $\Psi; \Gamma \Vdash^i_1 \delta \approx \delta' : \Delta$ relates two reducible regular substitutions t and t' at layer i.

The reducibility predicates are defined in two passes. In the first pass, I define the reducibility predicates of types, terms, local contexts and regular substitutions at layer 0 (this section). Note that the reducibility predicate of types does not really characterize reduction on the type level, as there are none. Its only purpose is to handle meta-substitutions of context variables. In the second pass, I define the predicates at layer 1 (Sec. 4.8), which extend those at layer 0 with meta-functions and contextual types, so the predicates in this section are states parametrically.

Reducibility of terms is defined by recursion on reducible types and is parameterized by layers. However, unlike syntactic judgments, where terms at layer 0 are from STLC, a reducible term at layer 0 may include meta-programs as sub-terms, though it must have types from layer 0 (i.e. STLC). In other words, reducible terms at layer 0 do not have to live at layer 0. This distinction is crucial especially when handling lifting semantically. For this reason, I put layering indices for the reducibility predicate of terms as superscripts to avoid confusions with subscripts, which specify where types and terms live in.

To understand this somewhat strange setup, consider the following code of the identity function for natural numbers:

$$\texttt{box}\ (\lambda x.x) : \Box(\cdot \vdash \texttt{Nat} \longrightarrow \texttt{Nat})$$

Since the identity function is constructed at layer 0, it is code of STLC. Nevertheless, the result of running it should yield the identity function at layer 1 due to the lifting lemma.

$$\texttt{letbox}\ u \leftarrow \texttt{box}\ (\lambda x.x)\ \texttt{in}\ u \approx \lambda y.y : \texttt{Nat} \longrightarrow \texttt{Nat}$$

Variables are renamed for the convenience of discussion. Something in this program is off: x comes from STLC so it is meant to be substituted by an STLC term, but due to lifting, it becomes y and now must also handle meta-programs, which clearly is not included in STLC. To illustrate,

$$\begin{aligned} & \texttt{letbox}\ u \leftarrow \texttt{box}\ (\lambda x.x)\ \texttt{in}\ u\ (\texttt{letbox}\ u' \leftarrow \texttt{box}\ 0\ \texttt{in}\ u') \\ \approx\ & (\lambda y.y)\ (\texttt{letbox}\ u' \leftarrow \texttt{box}\ 0\ \texttt{in}\ u') \\ \approx\ & \texttt{letbox}\ u' \leftarrow \texttt{box}\ 0\ \texttt{in}\ u' \\ \approx\ & 0 \end{aligned}$$

Thinking about semantics naively, the reducibility semantics of $\lambda x.x$ should be "the function takes a normalizing STLC term and computes to a normalizing STLC term", but normalizing or not, $\texttt{letbox}\ u' \leftarrow \texttt{box}\ 0\ \texttt{in}\ u'$ is clearly not an STLC term! The whole program, however, still computes as expected. To explain this mismatch in the semantics, the setup of the reducibility predicates described above becomes necessary. Though $\texttt{letbox}\ u' \leftarrow \texttt{box}\ 0\ \texttt{in}\ u'$ is a term from layer 1, its type Nat can live at layer 0. Therefore, I use *the layering restriction lemma* (Lemma 4.21) to confirm that arguments from layer 1 can be safely applied as long as they have types from STLC, which is definitely the case due to well-typedness. In other words, the layering restriction lemma is the semantic counterpart of the lifting lemma and is how layering provides code running for all code.

How the semantics applies functions from layer 0 is illustrated in Fig. 4.9. Assume $\lambda x.t$ is a function lifted from layer 0 and s is some term from layer 1. Then the

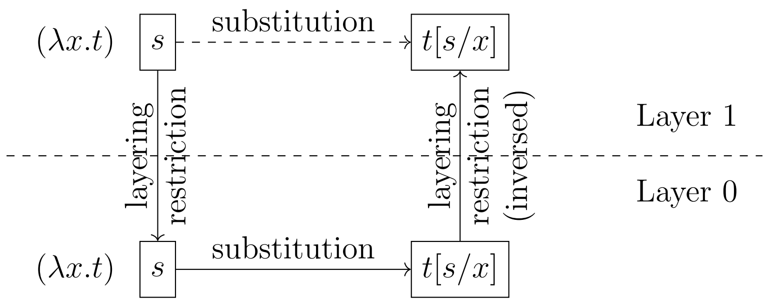


Figure 4.9: Illustration of *semantic* action on function applications

application at layer 1 (the dotted arrow) is semantically decomposed into three steps:

1. In the first step, layering restriction brings the semantics of s from layer 1 back to layer 0. This is possible only because s must have some type from STLC.

2. In the second step, the function application occurs. s replaces x in t everywhere. This step is the typical action for function applications.

3. In the third step, the inverse of layering restriction brings the result of the application from layer 0 to layer 1.

These three steps are exactly how the meta-variable case proceeds in the proof of the fundamental theorems (c.f. Theorem 4.28). Clearly, layering restriction is crucial to interpret the behaviors of lifted functions semantically and distinguishes the layered style from the heterogeneous style, where formal relations between sub-languages are intentionally avoided.

Having a conceptual grasp of how the semantics is organized, I move on to the first pass of defining the reducibility predicates. In this pass, I define the predicates for layer 0. Since the same predicates are also included at layer 1, they are defined parametrically in i. I first define reducibility of types as follows:

$$\frac{\vdash \Psi}{\Psi \Vdash^i \texttt{Nat}} \qquad \frac{\forall\ \gamma : \Phi \Longrightarrow_g \Psi\ .\ \Phi \Vdash^i S \qquad \forall\ \gamma : \Phi \Longrightarrow_g \Psi\ .\ \Phi \Vdash^i T}{\Psi \Vdash^i S \longrightarrow T}$$

Reducible types are almost the same as well-formed types, except that reducibility respects meta-weakenings, hence forming a Kripke model. In the function case, $S \longrightarrow T$

is semantically well-formed, if both S and T are required to be semantically well-formed under all meta-weakenings. Reducible meta-functions and contextual types at layer 1 use the reducible predicates at layer 0, so their definitions are postponed to Sec. 4.8. Keeping the predicates parameterized is convenient to see that layering restriction does hold. In particular, the predicates of natural numbers and functions do not depend on i.

Now I define the reducibility predicate for terms $\Psi; \Gamma \Vdash^i_1 t \approx t' : T$. The predicate is defined by recursion on reducible type $\Psi \Vdash^i T$. I first define the semantic natural numbers:

$$\frac{\Psi; \Gamma \vdash_1 t' \rightsquigarrow^* w' : \texttt{Nat} \qquad \begin{array}{c}\Psi; \Gamma \vdash_1 t \rightsquigarrow^* w : \texttt{Nat} \\ \Psi; \Gamma \vdash_1 w \simeq w' : \texttt{Nat}\end{array} \qquad \Psi; \Gamma \Vdash^{\textsf{Nf}} w \approx w' : \texttt{Nat}}{\Psi; \Gamma \Vdash t \approx t' : \texttt{Nat}}$$

$$\frac{}{\Psi; \Gamma \Vdash^{\textsf{Nf}} \textsf{zero} \approx \textsf{zero} : \texttt{Nat}} \qquad \frac{\Psi; \Gamma \Vdash t \approx t' : \texttt{Nat}}{\Psi; \Gamma \Vdash^{\textsf{Nf}} \textsf{succ}\ t \approx \textsf{succ}\ t' : \texttt{Nat}} \qquad \frac{\Psi; \Gamma \vdash_1 v \sim v' : \texttt{Nat}}{\Psi; \Gamma \Vdash^{\textsf{Nf}} v \approx v' : \texttt{Nat}}$$

Then $\Psi; \Gamma \Vdash^i_1 t \approx t' : \texttt{Nat} := \Psi; \Gamma \Vdash t \approx t' : \texttt{Nat}$. One can see clearly that the semantics of natural numbers does not rely on i at all.

Next, I define the case for function. $\Psi; \Gamma \Vdash^i_1 t \approx t' : S \longrightarrow T$ holds iff

- $\Psi; \Gamma \vdash_1 t \rightsquigarrow^* w : S \longrightarrow T$, and
- $\Psi; \Gamma \vdash_1 t' \rightsquigarrow^* w' : S \longrightarrow T$, and
- $\Psi; \Gamma \vdash_1 w \simeq w' : S \longrightarrow T$, and
- for any $\gamma; \tau : \Phi; \Delta \Longrightarrow \Psi; \Gamma$ and $\Phi; \Delta \Vdash^i_1 s \approx s' : S$, $\Phi; \Delta \Vdash^i_1 w\ s \approx w'\ s' : T$ holds.

It means that t and t' reduce to some weak-head normal forms, and the results of applying the weak-head normal forms to reducible terms remain reducible.

There are more cases for $i = 1$, which will be defined in Sec. 4.8. That said, the fact that the cases for natural numbers and functions can be given altogether, suggests that the semantics of these two types is irrelevant to whether $i = 0$ or $i = 1$. Indeed, this is the intuition for the layering restriction lemma.

Next, I generalize the reducibility predicates of types and terms to regular contexts and regular substitutions. Reducibility of regular contexts is defined inductively:

$$\frac{\vdash \Psi}{\Psi \Vdash^i \cdot} \qquad \frac{\vdash \Psi \qquad g : \mathsf{Ctx} \in \Psi}{\Psi \Vdash^i g} \qquad \frac{\Psi \Vdash^i \Delta \qquad \Psi \Vdash^i T}{\Psi \Vdash^i \Delta, x : T}$$

It simply requires all types within a regular context are semantically well-formed.

Then Reducibility of regular substitutions are defined by recursion on reducibility of regular contexts and simply applies the reducibility predicates of terms pointwise.

- $\Psi; \Gamma \Vdash^i_1 \delta \approx \delta' : \cdot$ holds iff $\Psi \vdash_i \Gamma$ and $\delta = \delta' = \cdot$.
- $\Psi; \Gamma \Vdash^i_1 \delta \approx \delta' : g$ holds iff $\Psi \vdash_i \Gamma$ and $\Gamma = g, \Gamma'$ for some Γ' and $\delta = \delta' = \mathsf{wk}$.
- $\Psi; \Gamma \Vdash^i_1 \delta \approx \delta' : \Delta, x : T$ holds iff
  - $\Psi; \Gamma \Vdash^i_1 \delta \approx \delta' : \Delta$,
  - $\Psi; \Gamma \Vdash^i_1 t \approx t' : T$.

Technically, this definition should be considered as two separate predicates, one for $i = 0$ and one for $i = 1$, because $\Psi; \Gamma \Vdash^1_1 t \approx t' : T$ will need to refer to $\Psi; \Gamma \Vdash^0_1 \delta \approx \delta' : \Delta$, so the whole definition cannot be defined parametrically. However, keeping them parameterized on paper helps to state the layering restriction lemma.

Lemma 4.21 (Layering Restriction)**.**

- *If $\Psi \Vdash^0 T$, then If $\Psi \Vdash^1 T$.*
- *If $\Psi \Vdash^0 T$, then $\Psi; \Gamma \Vdash^1_1 t \approx t' : T$ is equivalent to $\Psi; \Gamma \Vdash^0_1 t \approx t' : T$.*
- *If $\Psi \Vdash^0 \Delta$, then $\Psi \Vdash^1 \Delta$.*
- *If $\Psi \Vdash^0 \Delta$, then $\Psi; \Gamma \Vdash^1_1 \delta \approx \delta' : \Delta$ is equivalent to $\Psi; \Gamma \Vdash^0_1 \delta \approx \delta' : \Delta$.*

The more interesting direction is the one going from layer 1 to layer 0, which brings the semantics of terms and regular substitutions from layer 1 to layer 0. This lemma is crucial in the meta-variable case in the proof of the fundamental theorems (c.f. Theorem 4.28). Interestingly, in the syntax, the lifting lemma monotonically brings terms from layer 0 to layer 1, while in the semantics, layering restriction is an equivalence relation.

Another important lemma is monotonicity, which holds by design:

Lemma 4.22 (Monotonicity)**.**

- *If* $\Psi \Vdash^0 T$ *and* $\gamma : \Phi \Longrightarrow_g \Psi$*, then* $\Phi \Vdash^0 T$*.*
- *If* $\Psi \Vdash^0 T$ *and* $\Psi; \Gamma \Vdash^0_1 t \approx t' : T$*, given* $\gamma; \tau : \Phi; \Delta \Longrightarrow \Psi; \Gamma$*, then* $\Phi; \Delta \Vdash^0_1 t \approx t' : T$*.*
- *If* $\Psi \Vdash^0 \Gamma$ *and* $\gamma : \Phi \Longrightarrow_g \Psi$*, then* $\Phi \Vdash^0 \Gamma$*.*
- *If* $\Psi \Vdash^0 \Delta'$ *and* $\Psi; \Gamma \Vdash^0_1 \delta \approx \delta' : \Delta'$*, given* $\gamma; \tau : \Phi; \Delta \Longrightarrow \Psi; \Gamma$*, then* $\Phi; \Delta \Vdash^0_1 \delta \approx \delta' : \Delta'$*.*

The lemma says that all reducibility predicates are stable under weakenings.

4.8 Semantic Pattern Matching And Reducibility Predicates at Layer 1

In the previous section, I fully define reducibility predicates at layer 0 and partly for layer 1. The defined cases are for STLC, which are also included at layer 1. In this section, I complete the missing cases of reducibility predicates at layer 1. First, the remaining cases for reducible types at layer 1 are:

$$\frac{\Psi \Vdash^0 \Delta \qquad \Phi \Vdash^0 T}{\Psi \Vdash^1 \square(\Delta \vdash T)} \qquad\qquad \frac{\forall\, \gamma : \Phi \Longrightarrow_g \Psi \text{ and } \Phi \Vdash^0 \Gamma\ .\ \Phi \Vdash^1 T[\Gamma/g]}{\Psi \Vdash^1 (g : \mathsf{Ctx}) \Rightarrow T}$$

Reducible contextual types refer to reducible regular contexts and reducible types at layer 0. This is why reducibility must be defined by recursion on layers. Reducible meta-functions require codomain types to be stable under all meta-weakenings and meta-substitutions of context variables.

Now let us consider how to define reducible terms of contextual types. The guiding lemmas, lifting and static code, suggest that there are two different uses of a term of contextual type. It can be run, or be analyzed, or both. Therefore, the definition of

$\boxed{\Psi; \Gamma \Vdash^0_0 t : T}$ Code t of type T and all its sub-terms are reducible.

$$\frac{\Psi \Vdash^0 \Gamma \qquad x : T \in \Gamma \qquad \Psi; \Gamma \vDash^0_1 x : T}{\Psi; \Gamma \Vdash^0_0 x : T} \qquad \frac{\Psi; \Gamma \Vdash^0_0 \delta : \Delta \qquad u : (\Delta \vdash T) \in \Psi}{\Psi; \Gamma \Vdash^0_0 u^\delta : T} \qquad \frac{\Psi \Vdash^0 \Gamma \qquad \Psi; \Gamma \vDash^0_1 \mathsf{zero} : \mathtt{Nat}}{\Psi; \Gamma \Vdash^0_0 \mathsf{zero} : \mathtt{Nat}}$$

$$\frac{\Psi; \Gamma \Vdash^0_0 t : \mathtt{Nat} \qquad \Psi; \Gamma \vDash^0_1 \mathsf{succ}\ t : \mathtt{Nat}}{\Psi; \Gamma \Vdash^0_0 \mathsf{succ}\ t : \mathtt{Nat}} \qquad \frac{\Psi; \Gamma, x : S \Vdash^0_0 t : T \qquad \Psi; \Gamma \vDash^0_1 \lambda x.t : S \longrightarrow T}{\Psi; \Gamma \Vdash^0_0 \lambda x.t : S \longrightarrow T} \qquad \frac{\Psi; \Gamma \Vdash^0_0 t : S \longrightarrow T \qquad \Psi; \Gamma \Vdash^0_0 s : S \qquad \Psi; \Gamma \vDash^0_1 t\ s : T}{\Psi; \Gamma \Vdash^0_0 t\ s : T}$$

$$\Psi; \Gamma \Vdash^0_0 t \approx t' : T := \Psi; \Gamma \Vdash^0_0 t : T \text{ and } t = t'$$

$\boxed{\Psi; \Gamma \Vdash^0_0 \delta : \Delta}$ Regular substitution δ and all its sub-terms are reducible.

$$\frac{\Psi \vdash_i \Gamma}{\Psi; \Gamma \Vdash^0_0 \cdot : \cdot} \qquad \frac{\Psi \vdash_i g, \Gamma}{\Psi; g, \Gamma \vdash_i \mathsf{wk} : g} \qquad \frac{\Psi; \Gamma \Vdash^0_0 \delta : \Delta \qquad \Psi; \Gamma \Vdash^0_0 t : T}{\Psi; \Gamma \Vdash^0_0 \delta, t/x : \Delta, x : T}$$

$$\Psi; \Gamma \Vdash^0_0 \delta \approx \delta' : \Delta := \Psi; \Gamma \Vdash^0_0 \delta : \Delta \text{ and } \delta = \delta'$$

Figure 4.10: Semantic judgment for code

reducible terms of contextual types must carry both semantic and syntactic information for both uses. The semantic information handles code running and is characterized by the following definition: $\Psi; \Gamma \vDash^0_1 t : T$ holds iff

- $\Psi \Vdash^0 \Gamma$,
- $\Psi \Vdash^0 T$, and
- if $\gamma : \Phi \Longrightarrow_g \Psi$ and $\Phi; \Delta \Vdash^0_1 \delta \approx \delta' : \Gamma$, then $\Phi; \Delta \Vdash^0_1 t[\delta] \approx t[\delta'] : T$.

$\Psi; \Gamma \vDash^0_1 t : T$ is the standard semantic judgment for STLC, which characterizes the stability of reducibility under regular substitutions. Note that the superscripts in the last condition are 0's. They mean that given related regular substitutions (which might contain terms from layer 1) replacing variables of types from STLC (layer 0), the results are still reducible.

Following Hu and Pientka (2024b), the syntactic information handles pattern matching on code and is given by an inductive predicate, which is basically just the typing rules, with additional $\Psi; \Gamma \vDash_1^0 t : T$ to also keep tracking of the semantic information. This predicate is given in Fig. 4.10. For mnemonic purposes and the convenience to state semantic judgments, the predicate is $\Psi; \Gamma \Vdash_0^0 t : T$, with a subscript 0, denoting that it is a semantic judgment for code, i.e. terms from layer 0. This judgment remembers reducibility of all sub-terms, so all sub-terms can be run and be analyzed at any time. There is also a symmetrized version $\Psi; \Gamma \Vdash_0^0 t \approx t' : T$ by asking $t = t'$. The following semantic lifting and escape lemmas state that $\Psi; \Gamma \Vdash_0^0 t : T$ maintains semantic and syntactic information, respectively.

Lemma 4.23 (Semantic lifting)**.**

- *If* $\Psi; \Gamma \Vdash_0^0 t : T$*, then* $\Psi; \Gamma \vDash_1^0 t : T$*.*
- *If* $\Psi; \Gamma \Vdash_0^0 \delta : \Delta$*, then* $\Psi; \Gamma \vDash_1^0 \delta : \Delta$*.*

The semantic lifting lemma says that the semantic judgments for code maintain the semantic information that can be used for code running.

Lemma 4.24 (Escape)**.**

- *If* $\Psi; \Gamma \Vdash_0^0 t : T$*, then* $\Psi; \Gamma \vdash_0 t : T$*.*
- *If* $\Psi; \Gamma \Vdash_0^0 \delta : \Delta$*, then* $\Psi; \Gamma \vdash_0 \delta : \Delta$*.*

The escape lemma says that the semantic judgments for code can recover the syntactic judgments. In other words, the semantic judgments for code also maintain the syntactic information. Both lemmas combined imply that the semantic judgments for code maintain both necessary information for code running and pattern matching for all sub-structures.

Since $\Psi; \Gamma \vDash_1^0 t : T$ is stable under regular substitutions by definition, $\Psi; \Gamma \Vdash_0^0 t : T$ should also be stable under regular substitutions as it only adds syntactic information, which is clearly stable under regular substitutions.

Lemma 4.25 (Regular substitutions)**.**

- *If* $\Psi; \Gamma \Vdash_0^0 t : T$ *and* $\Psi; \Delta \Vdash_0^0 \delta : \Gamma$*, then* $\Psi; \Delta \Vdash_0^0 t[\delta] : T$*.*
- *If* $\Psi; \Gamma \Vdash_0^0 \delta' : \Delta'$ *and* $\Psi; \Delta \Vdash_0^0 \delta : \Gamma$*, then* $\Psi; \Delta \Vdash_0^0 \delta' \circ \delta : \Delta'$*.*

The semantic lifting and escape lemmas also confirm that now the setup is sufficient to give the definition of reducible terms at layer 1. First, I define the reducibility predicate of $\square(\Delta \vdash T)$:

$$\frac{\begin{array}{cc}\Psi; \Gamma \vdash_1 t \rightsquigarrow^* w : \square(\Delta \vdash T) & \Psi; \Gamma \vdash_1 t' \rightsquigarrow^* w' : \square(\Delta \vdash T) \\ \Psi; \Gamma \vdash_1 w \simeq w' : \square(\Delta \vdash T) & \Psi; \Gamma \Vdash^{\mathsf{Nf}} w \approx w' : \square(\Delta \vdash T)\end{array}}{\Psi; \Gamma \Vdash^1_1 t \approx t' : \square(\Delta \vdash T)}$$

$$\frac{\Psi; \Delta \Vdash^0_0 t : T}{\Psi; \Gamma \Vdash^{\mathsf{Nf}} \texttt{box}\ t \approx \texttt{box}\ t : \square(\Delta \vdash T)} \qquad \frac{\Psi; \Gamma \vdash_1 v \sim v' : \square(\Delta \vdash T)}{\Psi; \Gamma \Vdash^{\mathsf{Nf}} v \approx v' : \square(\Delta \vdash T)}$$

When t and t' reduce to some $\texttt{box}\ t''$, the reducibility predicate maintains the semantics of t'' as code, so that it can subsequently be run and analyzed.

The last case is for meta-functions. $\Psi; \Gamma \Vdash^1_1 t \approx t' : (g : \mathsf{Ctx}) \Rightarrow T$ holds iff

- $\Psi; \Gamma \vdash_1 t \rightsquigarrow^* w : (g : \mathsf{Ctx}) \Rightarrow T$, and
- $\Psi; \Gamma \vdash_1 t' \rightsquigarrow^* w' : (g : \mathsf{Ctx}) \Rightarrow T$, and
- $\Psi; \Gamma \vdash_1 w \simeq w' : (g : \mathsf{Ctx}) \Rightarrow T$, and
- for any $\gamma; \tau : \Phi; \Delta \Longrightarrow \Psi; \Gamma$, and given $\Phi \Vdash^0 \Delta'$, then we have $\Phi; \Delta \Vdash^1_1 w\ \$\ \Delta' \approx w'\ \$\ \Delta' : T[\Delta'/g]$.

This case is basically the same as regular functions.

The reducibility predicates at layer 1 are Kripke in the sense that they are monotonic. The monotonicity lemma is just like the monotonicity lemma for layer 0 except the change of layer from 0 to 1.

4.9 Semantic Judgments And Fundamental Theorems

Due to meta-functions, the following semantic soundness lemma is not immediately provable by induction:

$$\text{If } \Psi \vdash_i T, \text{ then } \Psi \Vdash^1 T.$$

where reducibility of meta-functions is given by the following rule:

$$\frac{\forall\ \gamma : \Phi \Longrightarrow_g \Psi \text{ and } \Phi \Vdash^0 \Gamma\ .\ \Phi \Vdash^1 T'[\Gamma/g]}{\Psi \Vdash^1 (g : \mathsf{Ctx}) \Rightarrow T'}$$

The induction hypothesis only gives

$$\Psi, g : \mathsf{Ctx} \Vdash^1 T'$$

which further implies the following by monotonicity:

$$\Phi, g : \mathsf{Ctx} \Vdash^1 T'$$

But then the proof is stuck because all assumptions do not say anything about meta-substitutions of g in T'.

The solution to this problem is to define the reducibility predicates for meta-contexts and meta-substitutions, following Abel et al. (2018). They are defined by induction-recursion on meta-contexts:

$$\frac{}{\Vdash \cdot}$$

Then $\Phi \vDash \sigma : \cdot$ holds iff $\vdash \Phi$ and $\sigma = \cdot$.

$$\frac{\Vdash \Psi}{\Vdash \Psi, g : \mathsf{Ctx}}$$

Then $\Phi \vDash \sigma : \Psi, g : \mathsf{Ctx}$ holds iff

- $\sigma = \sigma_1, \Gamma/g$,
- $\Phi \Vdash^0 \Gamma$, and+
- $\Phi \vDash \sigma_1 : \Psi$.

$$\frac{\Vdash \Psi \qquad \Psi \Vdash^0 \Gamma \qquad \Psi \Vdash^0 T}{\Vdash \Psi, u : (\Gamma \vdash T)}$$

$\Phi \vDash \sigma : \Psi, u : (\Gamma \vdash T)$ holds iff

- $\sigma = \sigma_1, t/u$,
- $\Phi \vDash \sigma_1 : \Psi$, and
- $\Psi; \Gamma[\sigma_1] \Vdash^0_0 t : T[\sigma_1]$.

where

- $\Psi \Vdash^i \Gamma$ is defined as given $\Phi \vDash \sigma : \Psi$, $\Phi \Vdash^i \Gamma[\sigma]$ holds.
- $\Psi \Vdash^i T$ is defined as given $\Phi \vDash \sigma : \Psi$, $\Phi \Vdash^i T[\sigma]$ holds.

In a meta-substitution, there are only two kinds of components: regular contexts and code of STLC. The latter is given above by the semantic judgment of code.

Semantic meta-substitutions are also monotonic.

Lemma 4.26 (Monotonicity)**.** *If $\Phi \vDash \sigma : \Psi$ and $\gamma : \Phi' \Longrightarrow_g \Phi$, then $\Phi' \vDash \sigma : \Psi$.*

Then the semantic judgments for terms and regular substitutions are stated as follows:

- $\Psi; \Gamma \Vdash_i t \approx t' : T$ iff $\Psi \Vdash^i \Gamma$ and $\Psi \Vdash^i T$ and for any $\Phi \vDash \sigma : \Psi$, $j \in [i, 1]$, and $\Phi; \Delta \Vdash^i_j \delta \approx \delta' : \Gamma[\sigma]$, then $\Phi; \Delta \Vdash^i_j t[\sigma][\delta] \approx t'[\sigma][\delta'] : T[\sigma]$ holds.
- $\Psi; \Gamma \Vdash_i \delta \approx \delta' : \Delta$ iff $\Psi \Vdash^i \Gamma$ and $\Psi \Vdash^i \Delta$ for any $\Phi \vDash \sigma : \Psi$, $j \in [i, 1]$, and $\Phi; \Delta' \Vdash^i_j \delta'' \approx \delta''' : \Gamma[\sigma]$, then $\Phi; \Delta' \Vdash^i_j \delta[\sigma] \circ \delta'' \approx \delta'[\sigma] \circ \delta''' : \Delta[\sigma]$ holds.
- $\Psi; \Gamma \Vdash_i t : T$ iff $\Psi; \Gamma \Vdash_i t \approx t : T$.
- $\Psi; \Gamma \Vdash_i \delta : \Delta$ iff $\Psi; \Gamma \Vdash_i \delta \approx \delta : \Delta$.

Note how the semantic judgments introduce another index parameter j. When $i = 1$, then necessarily $j = 1$. To prove $\Psi; \Gamma \Vdash_1 t \approx t' : T$, the goal is to prove $\Phi; \Delta \Vdash^1_1 t[\sigma][\delta] \approx t'[\sigma][\delta'] : T[\sigma]$ given $\Phi; \Delta \Vdash^1_1 \delta \approx \delta' : \Gamma[\sigma]$. However, when $i = 0$, then $j \in [0, 1]$, so the goal is in fact to prove two statements:

- $\Phi; \Delta \Vdash^0_1 t[\sigma][\delta] \approx t'[\sigma][\delta'] : T[\sigma]$ given $\Phi; \Delta \Vdash^0_1 \delta \approx \delta' : \Gamma[\sigma]$ and

- $\Phi; \Delta \Vdash^0_0 t[\sigma][\delta] \approx t'[\sigma][\delta'] : T[\sigma]$ given $\Phi; \Delta \Vdash^0_0 \delta \approx \delta' : \Gamma[\sigma]$. This is the same as proving $\Phi; \Delta \Vdash^0_0 t[\sigma][\delta] : T[\sigma]$ given $\Phi; \Delta \Vdash^0_0 \delta : \Gamma[\sigma]$.

These two statements are not independent. Combining with Lemma 4.25, the second statement is effectively to prove $\Phi; \Gamma[\sigma] \Vdash^0_0 t[\sigma] : T[\sigma]$, without the effect of regular substitutions. Further, as (main) part of the proof obligations, $\Phi; \Gamma[\sigma] \Vdash^0_0 t[\sigma] : T[\sigma]$ requires a proof that $\Phi; \Gamma[\sigma] \Vdash^0_1 t[\sigma] \approx t[\sigma] : T[\sigma]$ is stable under regular substitutions, which is already given by the first statement. Therefore, even though there are two statements to prove, the second statement is just an encapsulation of the first statement. The purpose of the second statement is to repackage semantic and syntactic information as the semantic judgment for code. This pattern is common in layered systems; the lower the layer is, the more information its semantics accumulates. In other words, a term from a lower layer carries information from all higher layers. The same pattern reoccurs in dependent types as well (c.f. Theorem 5.31).

The fundamental theorems are proved in two parts. The first part establishes the semantic soundness for meta-and regular contexts, and types. Most cases are straightforward. Since the semantic judgment of types respects meta-substitutions, the case for meta-functions goes through.

Theorem 4.27 (Fundamental)**.**

- *If* $\vdash \Psi$*, then* $\Vdash \Psi$*.*
- *If* $\Psi \vdash_i T$*, then* $\Psi \Vdash^i T$*.*
- *If* $\Psi \vdash_i \Gamma$*, then* $\Psi \Vdash^i \Gamma$*.*

The second part establishes the semantic soundness for terms and local substitutions. It says that given a syntactic judgment, its corresponding semantic judgment also holds at the same layer. Though in this thesis, I usually do not show detailed proof steps, but I do think that showing a few cases here is very important for readers to understand the pattern in simple types, as the same pattern is directly carried over to dependent types.

Theorem 4.28 (Fundamental)**.**

- *If* $\Psi; \Gamma \vdash_i t \approx t' : T$, *then* $\Psi; \Gamma \Vdash_i t \approx t' : T$.
- *If* $\Psi; \Gamma \vdash_i \delta \approx \delta' : \Delta$, *then* $\Psi; \Gamma \Vdash_i \delta \approx \delta' : \Delta$.
- *If* $\Psi; \Gamma \vdash_i t : T$, *then* $\Psi; \Gamma \Vdash_i t : T$.
- *If* $\Psi; \Gamma \vdash_i \delta : \Delta$, *then* $\Psi; \Gamma \Vdash_i \delta : \Delta$.

Proof. By a mutual induction on $\Psi; \Gamma \vdash_i t \approx t' : T$ and $\Psi; \Gamma \vdash_i \delta \approx \delta' : \Delta$. The third and fourth statements are results of the first two statements.

- $$\frac{\Psi; \Gamma \vdash_i \delta \approx \delta' : \Delta \qquad u : (\Delta \vdash T) \in \Psi}{\Psi; \Gamma \vdash_i u^{\delta} \approx u^{\delta'} : T}$$

  By a case analysis on i.

When $i = 1$ This is the case for code running.

$$\begin{aligned}
H_0 :& \Psi; \Gamma \Vdash_1 \delta \approx \delta' : \Delta && \text{(by IH)}\\
H_1 :& \Phi \vDash \sigma : \Psi && \text{(by assumption)}\\
& \Phi; \Delta' \Vdash\!\!\vDash_1^1 \delta'' \approx \delta''' : \Gamma[\sigma] && \text{(by assumption)}\\
H_2 :& \Phi; \Delta' \Vdash\!\!\vDash_1^1 \delta[\sigma] \circ \delta'' \approx \delta'[\sigma] \circ \delta''' : \Delta[\sigma] && \text{(by } H_0 \text{ and } H_1\text{)}\\
H_3 :& \Phi; \Delta' \Vdash\!\!\vDash_1^0 \delta[\sigma] \circ \delta'' \approx \delta'[\sigma] \circ \delta''' : \Delta[\sigma] &&\\
& && \text{(by layering restriction and } \Psi \Vdash^0 \Delta\text{)}\\
& \Phi; \Delta[\sigma] \Vdash\!\!\vDash_0^0 \sigma(u) : T[\sigma] && \text{(by lookup)}\\
H_4 :& \Phi; \Delta[\sigma] \vDash_1^0 \sigma(u) : T[\sigma] && \text{(by escape)}\\
& \Phi; \Delta' \Vdash\!\!\vDash_1^0 \sigma(u)[\delta[\sigma] \circ \delta''] \approx \sigma(u)[\delta'[\sigma] \circ \delta'''] : T[\sigma] && \text{(by } H_3 \text{ and } H_4\text{)}\\
& \Phi; \Delta' \Vdash\!\!\vDash_1^1 \sigma(u)[\delta[\sigma] \circ \delta''] \approx \sigma(u)[\delta'[\sigma] \circ \delta'''] : T[\sigma] &&\\
& && \text{(by inversed layering restriction)}
\end{aligned}$$

These steps are exactly those depicted in Fig. 4.9 and its ambient discussion.

When $i = 0$ Then in this case, there are two statements to prove depending on j. When $j = 1$, then the proof is almost the same as the proof of $i = 1$, but simpler, because there is no need to apply layering restriction as this is the case for code composition. When $j = 0$, the proof is to re-package the case for $j = 1$ into the semantic judgment for code.

$$\begin{aligned} H_0 :& \Psi; \Gamma \Vdash_0 \delta \approx \delta' : \Delta && \text{(by IH)} \\ H_1 :& \Phi \vDash \sigma : \Psi && \text{(by assumption)} \\ & \Phi; \Delta[\sigma] \Vdash\!\!=^0_0 \sigma(u) : T[\sigma] && \text{(by lookup)} \end{aligned}$$

The goal is to prove $\Phi; \Gamma[\sigma] \Vdash\!\!=^0_0 \sigma(u)[\delta[\sigma]] : T[\sigma]$. This is immediate as the semantic judgment of code is stable under local substitutions, together with $\delta[\sigma] = \delta'[\sigma]$ by H_0.

- $$\frac{\Psi \vdash_1 \Gamma \qquad \Psi; \Delta \vdash_0 t \approx t' : T}{\Psi; \Gamma \vdash_1 \texttt{box}\ t \approx \texttt{box}\ t' : \Box(\Delta \vdash T)}$$

$$\begin{aligned} H_0 :& \Psi; \Delta \Vdash_0 t \approx t' : T && \text{(by IH)} \\ H_1 :& \Phi \vDash \sigma : \Psi && \text{(by assumption)} \\ H_2 :& \Phi; \Delta' \Vdash\!\!=^1_1 \delta \approx \delta' : \Gamma[\sigma] && \text{(by assumption)} \\ & \Phi; \Delta[\sigma] \Vdash\!\!=^0_0 t[\sigma] : T[\sigma] \text{ and } t[\sigma] = t'[\sigma] && \text{(by } H_0 \text{ and } H_1\text{)} \\ & \Phi; \Delta[\sigma] \Vdash\!\!=^{\mathsf{Nf}} \texttt{box}\ t[\sigma] \approx \texttt{box}\ t'[\sigma] : \Box(\Delta[\sigma] \vdash T[\sigma]) && \text{(by definition)} \\ & \Phi; \Delta' \Vdash\!\!=^1_1 \texttt{box}\ t[\sigma][\delta] \approx \texttt{box}\ t'[\sigma][\delta'] : \Box(\Delta \vdash T)[\sigma] \end{aligned}$$

In the last step, regular substitutions do not propagate under `box`.

- $$\frac{\Psi \vdash_1 T' \qquad \Psi; \Gamma \vdash_1 s \approx s' : \Box(\Delta \vdash T) \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} \approx \overrightarrow{b}' : \Delta \vdash T \Rightarrow T'}{\Psi; \Gamma \vdash_1 \mathsf{match}\ s\ \mathsf{with}\ \overrightarrow{b} \approx \mathsf{match}\ s'\ \mathsf{with}\ \overrightarrow{b}' : T'}$$

In this case, I show how exactly the semantic judgment for code enables pattern matching on code.

$$\begin{aligned}
&H_0 : \Psi; \Gamma \Vdash_1 s \approx s' : \Box(\Delta \vdash T) && \text{(by IH)}\\
&H_1 : \Phi \vDash \sigma : \Psi && \text{(by assumption)}\\
&H_2 : \Phi; \Delta' \Vdash_1^1 \delta \approx \delta' : \Gamma[\sigma] && \text{(by assumption)}\\
&H_3 : \Phi; \Delta' \Vdash_1^1 s[\sigma][\delta] \approx s'[\sigma][\delta'] : \Box(\Delta \vdash T)[\sigma] && \text{(by } H_0\text{, } H_1 \text{ and } H_2\text{)}
\end{aligned}$$

Analyzing H_3 yields that for some w and w',

$$\begin{aligned}
\Phi; \Delta' &\vdash_1 s[\sigma][\delta] \rightsquigarrow^* w : \Box(\Delta[\sigma] \vdash (T[\sigma]))\\
\Phi; \Delta' &\vdash_1 s'[\sigma][\delta] \rightsquigarrow^* w' : \Box(\Delta[\sigma] \vdash (T[\sigma]))\\
H_4 : \Phi; \Delta' &\Vdash^{\mathsf{Nf}} w \approx w' : \Box(\Delta[\sigma] \vdash (T[\sigma]))
\end{aligned}$$

In a further case analysis on H_4, I only consider the following case as an example. Other cases proceed in a similar principle. For some S and S', so that $T = S \longrightarrow S'$, then

$$\dfrac{\dfrac{\Phi; \Delta[\sigma], x : S[\sigma] \Vdash_0^0 t'' : S'[\sigma] \qquad \Phi; \Delta[\sigma] \vDash_1^0 \lambda x.t'' : (S \longrightarrow S')[\sigma]}{\Phi; \Delta[\sigma] \Vdash_0^0 \lambda x.t'' : (S \longrightarrow S')[\sigma]}}{\Phi; \Delta' \Vdash^{\mathsf{Nf}} \texttt{box}\ \lambda x.t'' \approx \texttt{box}\ \lambda x.t'' : \Box(\Delta \vdash S \longrightarrow S')[\sigma]}$$

and $\overrightarrow{b}(\lambda x.t'') = \lambda x.?u \Rightarrow t$ and $\overrightarrow{b}'(\lambda x.t'') = \lambda x.?u \Rightarrow t'$.

$$\begin{aligned}
H_5 : &\Phi \vDash \sigma, t''/u : \Psi, u : (\Delta, x : S \vdash S') && \text{(by definition)}\\
&\Phi; \Delta' \Vdash_1^1 t[\sigma, t''/u][\delta] \approx t'[\sigma, t''/u][\delta'] : T'[\sigma, t''/u] && \text{(by IH and } H_5\text{)}\\
&T'[\sigma, t''/u] = T'[\sigma] && \text{(by computation)}\\
&\mathsf{match}\ s\ \mathsf{with}\ \overrightarrow{b}[\sigma][\delta]\\
= \ &\mathsf{match}\ s[\sigma][\delta]\ \mathsf{with}\ (\overrightarrow{b}[\sigma][\delta])\\
\rightsquigarrow^* \ &\mathsf{match}\ \texttt{box}\ \lambda x.t''\ \mathsf{with}\ (\overrightarrow{b}[\sigma][\delta])\\
\rightsquigarrow \ &t[q(\sigma)][\delta][t''/u]
\end{aligned}$$

$$
\begin{aligned}
&= t[q(\sigma) \circ (t''/u)][\delta[t''/u]] \\
&= t[\sigma, t''/u][\delta] && \text{(by computation)} \\
&\quad \mathsf{match}\ s\ \mathsf{with}'\ \overrightarrow{b}'[\sigma][\delta'] \rightsquigarrow^* t'[\sigma, t''/u][\delta'] && \text{(similarly)} \\
&\quad \Phi; \Delta' \Vdash^1_1 \mathsf{match}\ s\ \mathsf{with}\ \overrightarrow{b}[\sigma][\delta] \approx \mathsf{match}\ s'\ \mathsf{with}\ \overrightarrow{b}'[\sigma][\delta'] : T'[\sigma]
\end{aligned}
$$

Note that H_5 extends the meta-substitutions with t'' because $\Phi; \Delta[\sigma] \Vdash^0_0 \lambda x.t'' : (S \longrightarrow S')[\sigma]$ contains necessary syntactic information, so pattern matching can extract the syntactic information of the sub-term, i.e. t'' for arbitrary use in t and t'. Here $q(\sigma) := \sigma, u^{\mathsf{id}}/u$ and is also given in Appendix C.

□

4.10 Convertibility Checking

In this section, I give the conversion checking rules and instantiate the generic equivalence with it, proving that syntactic equivalence between terms is the same as convertibility. The conversion checking algorithms consist of four judgments:

- $\Psi; \Gamma \vdash_1 t \hat{\iff} t' : T$ checks the convertibility of two terms t and t'.
- $\Psi; \Gamma \vdash_1 \delta \hat{\iff} \delta' : \Delta$ is a pointwise generalization of $\Psi; \Gamma \vdash_1 t \hat{\iff} t' : T$.
- $\Psi; \Gamma \vdash_1 w \iff w' : T$ checks the convertibility of two normal forms w and w'. This operation is directed by types.
- $\Psi; \Gamma \vdash_1 v \longleftrightarrow v' : T$ checks the convertibility of two neutral forms v and v' and returns T. This operation is structural on the neutral forms.

The rules are given in Fig. 4.11. The conversion checking for neutral pattern matching on code follows the same principle as `letbox`, so I put the rules in Appendix E.

Finally, I instantiate the generic equivalence $\Psi; \Gamma \vdash_1 t \simeq t' : T$ with $\Psi; \Gamma \vdash_1 t \hat{\iff} t' : T$ and $\Psi; \Gamma \vdash_1 v \sim v' : T$ with $\Psi; \Gamma \vdash_1 v \longleftrightarrow v' : T$. Note that $\Psi; \Gamma \vdash_1 \delta \hat{\iff} \delta' : \Delta$ is equivalent to $\Psi; \Gamma \vdash_1 \delta \simeq \delta' : \Delta$.

The completeness theorem is proved from the fundamental theorems after the instantiation.

$$\frac{\Psi; \Gamma \vdash_1 t \rightsquigarrow^* w : T \qquad \Psi; \Gamma \vdash_1 t' \rightsquigarrow^* w' : T \qquad \Psi; \Gamma \vdash_1 w \Longleftrightarrow w' : T}{\Psi; \Gamma \vdash_1 t \mathrel{\hat{\Longleftrightarrow}} t' : T}$$

$$\frac{\Psi \vdash_1 \Gamma}{\Psi; \Gamma \vdash_1 \mathsf{zero} \Longleftrightarrow \mathsf{zero} : \mathtt{Nat}} \qquad \frac{\Psi; \Gamma \vdash_1 t \mathrel{\hat{\Longleftrightarrow}} t' : \mathtt{Nat}}{\Psi; \Gamma \vdash_1 \mathsf{succ}\ t \Longleftrightarrow \mathsf{succ}\ t' : \mathtt{Nat}} \qquad \frac{\Psi; \Gamma \vdash_1 v \longleftrightarrow v' : \mathtt{Nat}}{\Psi; \Gamma \vdash_1 v \Longleftrightarrow v' : \mathtt{Nat}}$$

$$\frac{\Psi; \Gamma, x : S \vdash_1 w\ x \mathrel{\hat{\Longleftrightarrow}} w'\ x : T}{\Psi; \Gamma \vdash_1 w \Longleftrightarrow w' : S \longrightarrow T} \qquad \frac{\Psi \vdash_1 \Gamma \qquad \Psi; \Delta \vdash_0 t : T}{\Psi; \Gamma \vdash_1 \mathtt{box}\ t \Longleftrightarrow \mathtt{box}\ t : \square(\Delta \vdash T)}$$

$$\frac{\Psi; \Gamma \vdash_1 v \longleftrightarrow v' : \square(\Delta \vdash T)}{\Psi; \Gamma \vdash_1 v \Longleftrightarrow v' : \square(\Delta \vdash T)} \qquad \frac{\Psi, g : \mathsf{Ctx}; \Gamma \vdash_1 w \ \$\ g \mathrel{\hat{\Longleftrightarrow}} w' \ \$\ g : T}{\Psi; \Gamma \vdash_1 w \Longleftrightarrow w' : (g : \mathsf{Ctx}) \Rightarrow T}$$

$$\frac{\Psi \vdash_1 \Gamma \qquad x : T \in \Gamma}{\Psi; \Gamma \vdash_1 x \longleftrightarrow x : T} \qquad \frac{\Psi; \Gamma \vdash_1 \delta \mathrel{\hat{\Longleftrightarrow}} \delta' : \Delta \qquad x : (\Delta \vdash T) \in \Psi}{\Psi; \Gamma \vdash_1 u^{\delta} \longleftrightarrow u^{\delta'} : T}$$

$$\frac{\begin{array}{c}\Psi; \Gamma \vdash_1 v \longleftrightarrow v' : S \longrightarrow T \\ \Psi; \Gamma \vdash_1 t \mathrel{\hat{\Longleftrightarrow}} t' : S\end{array}}{\Psi; \Gamma \vdash_1 v\ t \longleftrightarrow v'\ t' : T} \qquad \frac{\Psi; \Gamma \vdash_1 v \longleftrightarrow v' : (g : \mathsf{Ctx}) \Rightarrow T \qquad \Psi \vdash_0 \Delta}{\Psi; \Gamma \vdash_1 v \ \$\ \Delta \longleftrightarrow v' \ \$\ \Delta : T[\Delta/g]}$$

$$\frac{\Psi; \Gamma \vdash_1 v \longleftrightarrow v' : \square(\Delta \vdash T) \qquad \Psi \vdash_1 T' \qquad \Psi, u : (\Delta \vdash T); \Gamma \vdash_1 t \mathrel{\hat{\Longleftrightarrow}} t' : T'}{\Psi; \Gamma \vdash_1 \mathtt{letbox}\ u \leftarrow v\ \mathtt{in}\ t \longleftrightarrow \mathtt{letbox}\ u \leftarrow v'\ \mathtt{in}\ t' : T'}$$

Figure 4.11: Conversion checking algorithm

Theorem 4.29 (Completeness)**.**

- *If* $\Psi; \Gamma \vdash_1 t \approx t' : T$, *then* $\Psi; \Gamma \vdash_1 t \mathrel{\hat{\Longleftrightarrow}} t' : T$.
- *If* $\Psi; \Gamma \vdash_1 \delta \approx \delta' : \Delta$, *then* $\Psi; \Gamma \vdash_1 \delta \mathrel{\hat{\Longleftrightarrow}} \delta' : \Delta$.

The completeness theorem says that if two terms (substitutions, resp.) are equivalent, then they are also checked convertible by the algorithm.

Combined with the soundness theorem, which says that two convertible terms (substitutions, resp.) admitted by the algorithm are syntactic equivalent, I prove that convertibility is equivalent to syntactic equivalence.

Theorem 4.30 (Soundness)**.**

- *If* $\Psi; \Gamma \vdash_1 t \mathrel{\hat{\Longleftrightarrow}} t' : T$, *then* $\Psi; \Gamma \vdash_1 t \approx t' : T$.

- *If* $\Psi; \Gamma \vdash_1 w \Longleftrightarrow w' : T$*, then* $\Psi; \Gamma \vdash_1 w \approx w' : T$.

- *If* $\Psi; \Gamma \vdash_1 v \longleftrightarrow v' : T$*, then* $\Psi; \Gamma \vdash_1 v \approx v' : T$.

- *If* $\Psi; \Gamma \vdash_1 \delta \mathrel{\hat{\Longleftrightarrow}} \delta' : \Delta$*, then* $\Psi; \Gamma \vdash_1 \delta \approx \delta' : \Delta$.

Proof. Mutual induction on the premises. □

Next, I prove the decidability of convertibility.

Lemma 4.31 (Decidability of conversion checking)**.**

- *If* $\Psi; \Gamma \vdash_1 t \mathrel{\hat{\Longleftrightarrow}} t : T$ *and* $\Psi; \Gamma \vdash_1 t' \mathrel{\hat{\Longleftrightarrow}} t' : T$*, then* $\Psi; \Gamma \vdash_1 t \mathrel{\hat{\Longleftrightarrow}} t' : T$ *is decidable.*

- *If* $\Psi; \Gamma \vdash_1 \delta \mathrel{\hat{\Longleftrightarrow}} \delta : \Delta$ *and* $\Psi; \Gamma \vdash_1 \delta' \mathrel{\hat{\Longleftrightarrow}} \delta' : \Delta$*, then* $\Psi; \Gamma \vdash_1 \delta \mathrel{\hat{\Longleftrightarrow}} \delta' : \Delta$ *is decidable.*

- *If* $\Psi; \Gamma \vdash_1 w \Longleftrightarrow w : T$ *and* $\Psi; \Gamma \vdash_1 w' \Longleftrightarrow w' : T$*, then* $\Psi; \Gamma \vdash_1 w \Longleftrightarrow w' : T$ *is decidable.*

- *If* $\Psi; \Gamma \vdash_1 v \longleftrightarrow v : T$ *and* $\Psi; \Gamma \vdash_1 v' \longleftrightarrow v' : T$*, then* $\Psi; \Gamma \vdash_1 v \longleftrightarrow v' : T$ *is decidable.*

Combining completeness, convertibility is decidable.

Theorem 4.32 (Decidability of convertibility)**.**

- *If* $\Psi; \Gamma \vdash_1 t : T$ *and* $\Psi; \Gamma \vdash_1 t' : T$*, then* $\Psi; \Gamma \vdash_1 t \approx t' : T$ *is decidable.*

- *If* $\Psi; \Gamma \vdash_1 \delta : \Delta$ *and* $\Psi; \Gamma \vdash_1 \delta' : \Delta$*, then* $\Psi; \Gamma \vdash_1 \delta \approx \delta' : \Delta$ *is decidable.*

4.11 Comparison with Homogeneous and Heterogeneous Styles

In Sec. 4.7, I discussed the setup of the reducibility predicates and why I put i as a superscript in $\Psi; \Gamma \Vdash^i_1 t \approx t' : T$. To briefly recapitulate, in order to handle lifting of functions from STLC semantically, the layer i marks which layer T comes from. If

$\Psi \Vdash^0 T$, then the layering restriction applies to bring arguments at layer 1 to layer 0, so that they can be applied to functions from STLC. The layering restriction lemma models lifting in the semantics. This is how layering is distinguished from regular heterogeneous systems, where relations among different sub-languages are intentionally avoided in the latter kind.

Layered modal type theory is also somewhat similar to the dual-context-style system of contextual $\lambda^{\square}$ by Nanevski et al. (2008). In contextual $\lambda^{\square}$, every $\square(\Gamma \vdash T)$ is valid, and `box` and `letbox` can be arbitrarily nested. More precisely, the typing judgment of $\lambda^{\square}$ is $\Psi; \Gamma \vdash t : T$, without the layering index at all. The introduction and elimination rules for $\square$ are:

$$\frac{\Psi; \Delta \vdash t : T}{\Psi; \Gamma \vdash \texttt{box}\ t : \square(\Delta \vdash T)} \qquad \frac{\Psi; \Gamma \vdash s : \square(\Delta \vdash T) \qquad \Psi, u : (\Delta \vdash T); \Gamma \vdash t : T'}{\Psi; \Gamma \vdash \texttt{letbox}\ u \leftarrow s\ \texttt{in}\ t : T'}$$

For convenience, in the example below I set $\Delta = \cdot$, so $\square(\cdot \vdash T)$ degenerates to $\square T$. The following program is valid in $\lambda^{\square}$ but is not in layered modal type theory:

$$\texttt{letbox}\ u \leftarrow \texttt{box}\ (\texttt{box}\ \textsf{zero})\ \texttt{in}\ u : \square\texttt{Nat}$$

This program has type $\square$`Nat` and computes to `box` zero. However, due to the two nested `box`'s, this program is not well-typed in layered modal type theory.

Following this thought, I give a program that is valid in $\lambda^{\square}$, but would never be valid even if layered modal type theory is generalized to arbitrary n layers:

$$\begin{aligned}
\textsf{lift}\ &: \texttt{Nat} \to \square\texttt{Nat} \\
\textsf{lift}(\textsf{zero})\ &:= \texttt{box}\ \textsf{zero} \\
\textsf{lift}(\textsf{succ}\ x)\ &:= \texttt{letbox}\ u \leftarrow \textsf{lift}(x)\ \texttt{in}\ \texttt{box}\ (\textsf{succ}\ u)
\end{aligned}$$

$$\begin{aligned}
\textsf{nest}\ &: \texttt{Nat} \to \square\texttt{Nat} \\
\textsf{nest}(\textsf{zero})\ &:= \texttt{box}\ \textsf{zero} \\
\textsf{nest}(\textsf{succ}\ x)\ &:= \texttt{letbox}\ u \leftarrow \textsf{lift}(x)\ \texttt{in}\ \texttt{box}\ (\texttt{letbox}\ u' \leftarrow \textsf{nest}\ u\ \texttt{in}\ u')
\end{aligned}$$

The lift function lifts a natural number into a `box`. That is, a number succ $(\cdots$ (succ zero)) becomes `box` (succ $(\cdots$ (succ zero))). This function is valid in both $\lambda^{\square}$ and layered modal type theory. The nest function is a bit trickier and is only valid in $\lambda^{\Box}$. The step case of nest makes a recursive call on the result u of $\mathsf{lift}(x)$ inside $\texttt{box}$. Given a concrete natural number m, $\mathsf{nest}(m)$ produces

$$\texttt{box}\ (\texttt{letbox}\ u_m \leftarrow \texttt{box}\ (\cdots(\texttt{letbox}\ u_1 \leftarrow \texttt{box}\ \mathsf{zero}\ \texttt{in}\ u_1)\cdots)\ \texttt{in}\ u_m)$$

Thus, $\mathsf{nest}(m)$ produces m nested layers of $\texttt{box}$'s and $\texttt{letbox}$'s. The computational behavior of this function is rather uninteresting. By giving $\mathsf{nest}(m)$ to a $\texttt{letbox}$, all the $\texttt{letbox}$'s inside of the outermost $\texttt{box}$ will eventually collapse and therefore the following computation always holds

$$\texttt{letbox}\ u \leftarrow \mathsf{nest}(m)\ \texttt{in}\ u \approx \mathsf{zero}$$

This nest function can be encoded in $\lambda^{\Box}$ because $\lambda^{\Box}$ is simply not concerned about the nesting at all. However, since m can be arbitrary, any fixed n-layered generalized modal type theory cannot encode this function. This observation suggests that a layered system is always strictly weaker than $\lambda^{\Box}$ in certain aspects. This loss of expressive power is the cost to open up a completely orthogonal direction, where covering pattern matching on code can be soundly added to the type theory. Supporting pattern matching on code compensates the expressive power in a different dimension, making the overall expressive power of layered modal type theory incomparable with that of $\lambda^{\Box}$.

4.12 Summary

In this section, I develop an almost simply typed layered modal type theory and show that layering is the right path to supporting intensional analysis. In particular, layered modal type theory retains confluence and normalization, and checking convertibility between well-typed terms is decidable. Supporting intensional analysis is important: it is a frequently used feature when writing tactics in proof assistants. Due to limitations from simple types, the core type theory cannot support a general recursion principle on code (special ones are possible, e.g. one in the examples), but this problem naturally goes away with dependent types. In the next chapter, I will show that layering and the matryoshka principle also scale naturally to dependent types, and demonstrate how intensional analysis can be supported coherently with dependent types.

5

DeLaM: Dependent Layered Modal Type Theory

In Chapter 4, I presented a simply typed layered modal type theory, which is based on the matryoshka principle: sub-languages at higher layers subsume those at lower layers. In layered modal type theory, users can not only compose and execute code as in the homogeneous $\lambda^{\Box}$ by Davies and Pfenning (2001); Pfenning and Davies (2001) investigated in Part I, but also pattern match on the syntactic structure of code. Layered model type theory is coherent, in the sense that it satisfies weak normalization and its syntactic convertibility is decidable.

Though layered modal type theory has provided a type-theoretic foundation for meta-programming with intensional analysis for simple types, whether the layered style and the matryoshka principle can be carried over to dependent types remains an interesting question to answer. In this chapter, this question is answered positively and I develop DeLaM, a **De**pendent **La**yered **M**odal type theory, which enables meta-programming in Martin-Löf type theory (MLTT) with recursion principles on open

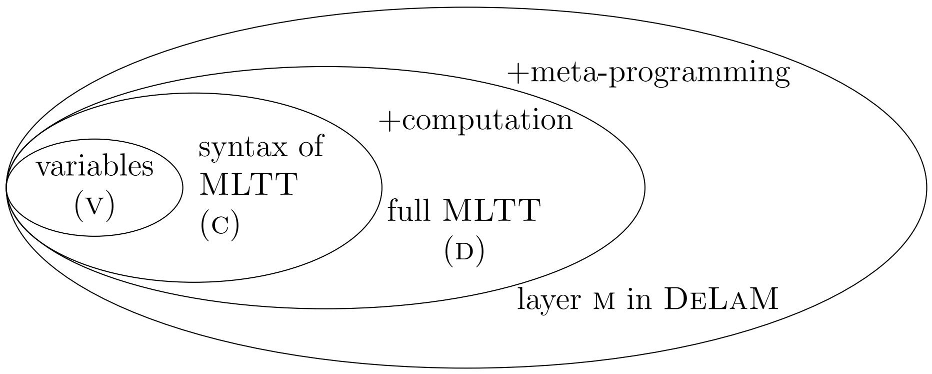


Figure 5.1: Layer hierarchy of DeLaM

code. The investigation in Chapter 4 scales surprisingly well both in syntax and in semantics. On the syntactic side, the matryoshka principle is formally captured by two *guiding lemmas*: even though *the static code lemma* proves that code objects do not compute, via *the lifting lemma*, code objects may compute after being lifted to higher layers. Both lemmas are very simple to check and are very helpful when designing the syntactic judgments. On the semantic side, I also construct the Kripke logical relations for all layers, and prove *the layering restriction lemma* to model code running in the semantics. Many aspects in DeLaM are natural extensions of layered modal type theory, with more technicalities to handle dependent types. Readers might find Chapter 4 simpler to digest for high-level ideas. This chapter is dedicated for more specific discussions on dependent types.

In contrast to layered modal type theory, where I name the layers by numbers, layers in DeLaM are named by letters for a better mnemonic: code objects for variables (V), code objects for MLTT (C), the dependent type theory of MLTT (D), and meta-programs (M), such that $\text{V} < \text{C} < \text{D} < \text{M}$. Syntactically, the layers V and C describe static code objects of MLTT with no computation; contrarily, the layer D corresponds to the dependent type theory of pure MLTT and allows computation. The meta-programming layer M extends MLTT with contextual types, code objects and recursion principles over them. Hence the expressive and the computational power of sub-languages strictly increases along the layers. The relation of the layers is depicted in Fig. 5.1. An outer layer is always an extension of its inner layer.

As proved by the static code lemma, code objects at layers V and C are static and do not compute, but their types do and live at the higher layer D. To illustrate, consider the constructor for an empty list `nil : (A : Ty) → List A`, where `Ty` is the type for the universe. Then `box (nil ((λ x. x) Nat))` and `box (nil Nat)` represent two distinct code objects at layer C, so they are not equivalent. In particular, the β redex in the first code object does not compute at layer C. However, both contextual types `□ ( ⊢ List ((λ x. x) Nat))` and `□ ( ⊢ List Nat)` are equivalent and are equally valid types for the code objects. The β redex in the first contextual type now appears on the type level due to dependent types and computes. This action of bringing code (C) to the type level (D) is a special instance of the lifting lemma, which I call *code promotion*. Note that the types of code living at a separate layer and code promotion do not exist in layered modal type theory and are unique in DeLaM. They are introduced as one possible (probably the most natural) way to combine the matryoshka principle and dependent types. Finally, the meta-programming layer M extends MLTT with constructs to support an explicit way to execute code. Just like Chapter 4, a code object, e.g. `box ((λ x. x) 0)`, can be eliminated by `letbox` and run at layer M. The lifting lemma applies here to lift a term from layer C to layer M, so that a code object may compute.

In this chapter, I will first give two examples for meta-programming in DeLaM (Sec. 5.1). Due to recursion on code, meta-programs in DeLaM are able to analyze the structures of proof obligations and apply algorithms to solve them. In other words, DeLaM provides a type-safe way to write tactics and complements many existing tactic systems in proof assistants. Then, I will give the definition of DeLaM (Sec. 5.2 and 5.3). Formally, DeLaM is a dependent type theory with a non-cumulative, Tarski-style universe hierarchy. It notably has one uniform syntax for all layers, so users only need to be familiar with one language for programming, proving, and meta-programming. One major advantage of this syntactic design is that DeLaM allows users to access all definitions from MLTT freely at the meta-programming layer M without having to forgo critical meta-theoretic properties like normalization. This design allows users to use the same set of libraries for both programming and meta-programming, and hence improves users' time efficiency. Layers only have impact on the judgmental level, where layers distinguish valid objects and valid computational behaviors following the matryoshka principle. Next, I follow Abel et al. (2018) and extend the Kripke logical relations in

Chapter 4 to DeLaM (Sec. 5.4). Despite the increasing complexity, the logical relations for DeLaM follow the same skeleton as in Chapter 4. Roughly speaking, I first define the logical relations for types and terms for MLTT over a set of parametric generic equivalences. These logical relations capture how terms compute at both layers D and M. Then, the semantics of MLTT is encapsulated in a set of inductive semantic judgments to model code. These semantic judgments record the syntactic shape of code, so they are virtually just typing rules, but also carry extra semantics of MLTT to describe how code objects compute. Finally, I give the logical relations for types and terms for layer M, where the semantics of contextual types are given in terms of the semantic judgments of code. The fundamental theorems are instantiated accordingly to obtain the two conclusive theorems: weak normalization and the decidability of convertibility of DeLaM (Sec. 5.5).

The investigation presented in this chapter is very complex, so I only focus on the high-level ideas. This chapter is incremental on Chapter 4. Many lemmas and theorems in this chapter find simplified counterparts in Chapter 4. This chapter is published work (Hu and Pientka, 2025). Readers interested in full technical details might refer to the technical report (Hu and Pientka, 2024a).

5.1 DeLaM by Examples

DeLaM supports quotation of static MLTT code objects in `box` and composition of code objects. These code objects can be further analyzed by recursion and eventually be executed to obtain MLTT programs. These features allow us to write widely used tactics, and the generated proofs are guaranteed to be well-scoped and well-formed as ascribed by their types.

5.1.1 Recursion on Code Objects Describing MLTT Terms

As a first example, I implement a tactic, which checks whether two expressions are equal up to associativity and commutativity (AC). This functionality is frequently desired in a proof assistant. For example, both Isabelle/HOL (Wenzel et al., 2024, Sec. 9.3.3) and Lean provide such AC solvers.

To make the implementation concise, I concentrate on an AC checker for summations of expressions of natural numbers. In addition, I assume that the addition `+`, its

```
ac-check : (g : Ctx) ⇒ (a b : □ (g ⊢ Nat)) →
  letbox a' ← a; b' ← b in Option (□ (g ⊢ Eq Nat a' b'))
ac-check g (box (a1 + a2)) b =
  letbox b' ← b in search g (box a1) (box b') >>= λ (c, pf1).
  letbox c' ← c in ac-check g (box a2) (box c') >>= λ pf2.
  letbox pf1' ← pf1; pf2' ← pf2 in
  Some (box (trans (cong (a1 +_) pf2') (sym pf1')))
ac-check g (box a') b =
  letbox b' ← b in
  if eq? (box a') (box b') then Some (box refl) else None

search : (g : Ctx) ⇒ (a b : □ (g ⊢ Nat)) →
  letbox a' ← a; b' ← b in
  Option (Σ (c : □ (g ⊢ Nat)).
    letbox c' ← c in □ (g ⊢ Eq Nat b' (a' + c')))
search g a (box (b1 + b2)) =
  letbox a' ← a in
  if eq? (box b1) (box a')
  then Some (box b2, box refl)
  else if eq? (box b2) (box a')
  then Some (box b1, box (comm b1 b2))
  else search g (box a') (box b2) >>= λ (c, pf).
  letbox c' ← c; pf' ← pf in
  Some (box (b1 + c'), box (trans (cong (b1' +_) pf') (pull b1' a 'c')))
search g a (box b') = letbox a' ← a in None
```

Figure 5.2: An implementation of an equality checker modulo AC

associativity and commutativity, and transitivity and Leibniz substitution of equality have been defined at layer C, so I have access to these definitions in a code object. Finally, I assume that a summation is written in the right-associated form, i.e. of the form $a_1 + (a_2 + \ldots + (a_n + a_{n+1}))$ and a_i is an object of type `Nat`. In practice, this assumption can be ensured by a preprocessing phase, where associativity is repetitively applied.

The main idea of this tactic is simple: to compare $a_1 + \ldots + (a_n + a_{n+1})$ with `b`, I match all a_i's with addends in `b` until a_{n+1}, and then I just compare a_{n+1} with the rest of `b` for syntactic equality. The tactic is implemented as the `ac-check` function in Fig. 5.2 in an Agda-like surface syntax. In the type of `ac-check`, a fat arrow ⇒ denotes a *meta-function*. The `ac-check` function is polymorphic in the context `g`, so it applies for any regular context. Next, it takes two pieces of code `a` and `b` as inputs, with their open variables in `g`. The return type should express the equality between `a` and `b`.

To do that, I first use `letbox` to unpack `a` and `b` to obtain meta-variables `a'` and `b'`, so that I can subsequently use them in the contextual type `(□ (g ⊢ Eq Nat a' b'))`. [15] This contextual type describes the proof that `a'` and `b'` are equal where `Eq` refers to propositional equality. The actual return type of `ac-check` is an `Option`, as `ac-check` might not be able to prove the equality.

The `ac-check` function is defined by recursion on the input `a`.

1. If `a` is `box` (a_1 `+` a_2), then a_1 and a_2 are pattern variables representing open code of addends of `a` in the parametric context `g`. Then, I use the `search` function to look for a_1 in `b`. It optionally returns code `c` with an equality proof pf_1 of `b` `=` a_1 `+ c`. Hence `c` describes the remainder of `b` excluding a_1 and I recursively compare a_2 with `c`. Since the tactics return `Options`, I use the monadic bind operation (`>>=`) to short-circuit the uninteresting `None` case. If the recursion is successful, then I obtain two proofs at the end: pf_1 for `b` `=` a_1 `+ c` and pf_2 for a_2 `= c`. The proof obligation of `ac-check` requires a code object of a proof of a_1 `+` a_2 `= b`, which can be proved by following a_1 `+` a_2 `=` a_1 `+ c = b`. The proof term is written in an Agda convention. Inside of `box`, I use the transitivity of `Eq`, `trans`, which requires two further sub-goals a_1 `+` a_2 `=` a_1 `+ c` and a_1 `+ c = b`. The first sub-goal is established by the congruence of `+`, i.e. `cong (`a_1 `+_)`, to replace a_2 with `c` using pf_2`'`. The second sub-goal is immediate by pf_1`'` after symmetry.

Agda users might find the last step familiar. Indeed, providing the proof obligations in DELAM resembles filling in holes in Agda in many aspects. The critical difference however, is that in DELAM, proof obligations are fulfilled using (meta-)programs.

2. If `a` is not an addition at all, then the only possibility for it to be equal to `b` is that both syntactically describe the same code. This is tested by `eq?`.

The `search` function is the main driver of the algorithm. It recurses on `b` to look for an addend identical to `a`. If `b` has no addition, then `search` fails and returns `None`. If `b` is `box` (b_1 `+` b_2), the first two `if`s compare `a` with b_1 and b_2. If either comparison succeeds, then this addend have been found. Otherwise, in the last `else`, a recursive `search` continues to look for `a` in b_2. If successful, the recursion returns some `c` and a proof `pf`

[15]A practical front-end would insert `letbox`'s smartly, but for the purpose of this thesis, I would like to make the mechanism explicit.

of `b₂ = a + c`. The return value is `b₁ + c` together with a proof of `b₁ + b₂ = a + (b₁ + c)`. The proof obligation is established again by a transitivity. First, a congruence proves the first sub-goal of `b₁ + b₂ = b₁ + (a + c)`. Then, the second sub-goal requires to prove `b₁ + (a + c) = a + (b₁ + c)`, which is established by a call to `pull`: `pull b₁' a' c'`, which swaps `b₁` and `a`. This property is called *left commutativity*, and is often required in an AC solver, e.g. in Isabelle/HOL and in Lean. Left commutativity can be proved by a sequence of associativity, commutativity and again associativity.

The tactic `ac-check` can be used to algorithmically derive proofs for many tedious equations about natural numbers. The following lemma is one example:

```
lem : (x y z : Nat) → Eq Nat (x + (y + z)) (y + (z + x))
lem x y z =
  let Some pf ← ac-check (a : Nat, b : Nat, c : Nat) -- context
                         (box (a + (b + c))) (box (b + (c + a)))
  in letbox u ← pf in u[x/a,y/b,z/c]
```

The first argument to `ac-check` is the ambient context `a : Nat, b : Nat, c : Nat` and unrelated to the function arguments `x`, `y` and `z`. The next two arguments are code objects describing both sides of the equation in the ambient context above. Since the invocation of `ac-check` is closed, it will return `Some pf` by following a sequence of computations, where `pf` is a code object denoting the equality proof of the code type `□(a:Nat,b:Nat,c:Nat ⊢Eq Nat (a + (b + c)) (b + (c + a)))`. Note that `pf` is a well-formed equality proof of the given contextual type. To use the equality proof `pf` at the layer M, I use `letbox` to bind it to the meta-code variable `u`. Subsequently, in the body of `letbox`, I use `u` with a regular substitution to substitute `x`, `y` and `z` for `a`, `b` and `c`, respectively. This pattern of extracting a proof from code is an instance of code running, which is justified by the lifting lemma.

Finally, definitions like `Option`, `Eq` and `>>=` are defined in MLTT, but the meta-programming layer M also has access to them, due to the uniform syntax of DeLaM for all layers and the lifting lemma. Layers are only used in type-checking to control valid types, terms and computation. Hence, users of DeLaM only need to learn one language for programming, proving and meta-programming.

5.1.2 Recursion on Code Objects Describing MLTT Types

In DeLaM, code objects describe not only MLTT terms, but also MLTT types. A code object representing a MLTT type has a contextual type `□ (g ⊢ @l)` where `@l` denotes the universe level `l` of the type. Hence, DeLaM supports recursion on the shape of a code object of types. Being able to write meta-programs that recursively analyze the code describing MLTT types is key to implementing tactics.

To illustrate, consider a goal of the following form: it is either a universally quantified formula `(x : Nat) → F'`, a conjunction `F₁ ∧ F₂`, or an equality between arithmetic expressions. A quotation of a goal in this form yields a contextual code object of type `F : □ (g ⊢ @0)`. Below, I implement the `crush` tactic, which takes in `F` as an input and constructs a code object describing the proof for `F`, if `crush` finds it. The tactic is implemented as a meta-program in DeLaM by pattern matching on the shape of `F` and leverages the previous AC checker to crush equalities between arithmetic expressions.

```
crush : (g : Ctx) ⇒ (F : □ (g ⊢ @0)) →
  letbox F' ← F in Option (□ (g ⊢ F'))
crush g (box (Eq Nat a b))  = ac-check g (box a) (box b)
crush g (box (F₁ ∧ F₂))     = crush g (box F₁) >>= λ (r₁ : □ (g ⊢ F₁)).
                              crush g (box F₂) >>= λ (r₂ : □ (g ⊢ F₂)).
  letbox pf₁ ← r₁ ; pf₂ ← r₂ in Some (box (pf₁, pf₂))
crush g (box ((x:Nat) → F)) = crush (g, x:Nat) (box F) >>=
  λ (r : □ (g,x:Nat ⊢ F)). letbox pf ← r in Some (box (λ x. pf))
crush g (box _)             = None
```

In the first case, if the goal formula is simply the equality between arithmetic expressions `a` and `b`, the solving is delegated to `ac-check`. Otherwise, if the goal is a conjunction, then both components are crushed and their proofs are composed together if both `crush`ings are successful. I again use the bind operation (`>>=`) for convenience. If the goal is a universal quantification, then the regular context is extended with the parameter `x:Nat` and the recursion is invoked on `box F`. In general, abstracting over contexts is crucial for recursion on binders that extend regular contexts (see also (Pientka et al., 2019) for similar situations). If the recursive call on `F` is successful and returns a proof `r`, the proof obligation requires a proof for `(x : Nat) → F`, which is achieved by embedding `pf` in a `λ`. The last case captures all other shapes of formulas and returns

None. Now, crush can be used to solve more complex goals such as the following:

```
lem2 : (x y : Nat) → Eq Nat (x + y) (y + x) ∧
                ((z : Nat) → Eq Nat (x + (y + z)) (y + (z + x)))
lem2 =
  let Some pf ← crush ()
    (box ((x y : Nat) → Eq Nat (x + y) (y + x) ∧
           ((z : Nat) → Eq Nat (x + (y + z)) (y + (z + x)))))
  in letbox u ← pf in u
```

This tactic follows the same pattern as in the last example. The crush tactic automatically introduces the arguments to regular contexts and handles the conjunction. Eventually two invocations to ac-check will be successful as well as the overall call to crush, producing a proof of this lemma.

5.2 Syntax of DeLaM

Starting this section, I define DeLaM formally. DeLaM includes multiple layers: the layer c accommodates static code objects, whereas the layer d corresponds to the dependent type theory of MLTT. The topmost layer m extends MLTT with contextual types and other constructs for composition, execution, and recursion on code. While the syntax of DeLaM (see Fig. 5.3) is uniform and thus users only need to learn one language, layers are distinguished in the judgments, which define well-formed objects and valid computational bahaviors. I will dissect the syntax gradually and discuss my design decisions. Readers might find hyperlinks in the text convenient.

5.2.1 Explicit Universe Polymorphism

DeLaM supports universe polymorphism following Bezem et al. (2022). Universes (l) form an idempotent commutative monoid, where $l \sqcup l'$ takes the maximum of l and l'. The ω level is added to support universe-polymorphic functions. A universe level l is well-formed w.r.t. a universe context L, if all universe variables (ℓ) in l appear in L and l contains no ω. Similar to Agda, universes respect a number of equalities: identity ($0 \sqcup l = l$), distributivity ($1 + (l \sqcup l') = (1 + l) \sqcup (1 + l')$), absorption ($l \sqcup (1 + l) = 1 + l$), commutativity ($l \sqcup l' = l' \sqcup l$), associativity ($(l_1 \sqcup l_2) \sqcup l_3 = l_1 \sqcup (l_2 \sqcup l_3)$) and idempotence ($l \sqcup l = l$). Given two well-formed universes l and l', whether $l = l'$ is decidable as implemented in Agda. One possible algorithm is to compare the universe level

$$
\begin{array}{rll}
i, j, k \in & \{\textsc{v}, \textsc{c}, \textsc{d}, \textsc{m}\} & \text{(Layers, where } \textsc{v} < \textsc{c} < \textsc{d} < \textsc{m}\text{)} \\
x, \ell & & \text{(Regular, universe variables, resp.)} \\
l := & \ell \mid 0 \mid 1 + l \mid l \sqcup l' \mid \omega & \text{(Universe levels)} \\
g, U, u & & \text{(Meta-variables for regular contexts, types, terms, resp.)} \\
L := & \cdot \mid L, \ell & \text{(Universe contexts)} \\
B := & g : \mathsf{Ctx} \mid U : (\Gamma \vdash_i \ @\ l) \mid u : (\Gamma \vdash_i T\ @\ l) & \text{(Meta-bindings)} \\
\Phi, \Psi := & \cdot \mid \Phi, B & \text{(Meta-contexts)} \\
\Gamma, \Delta := & \cdot \mid g \mid \Gamma, x : T\ @\ l & \text{(Regular contexts)} \\
\delta := & \cdot \mid \mathsf{wk} \mid \delta, t/x & \text{(Regular substitutions)} \\
M, S, T := & \mathtt{Ty}_l \mid \mathtt{Nat} \mid \Pi^{l,l'}(x : S).T \mid U^{\delta} \mid \overrightarrow{\ell} \Rightarrow^l T \mid \mathtt{El}^l\ t & \text{(Types)} \\
& \mid (g : \mathsf{Ctx}) \Rightarrow^l T \mid (U : (\Gamma \vdash_{\textsc{d}} \ @\ l)) \Rightarrow^{l'} T \mid \square(\Gamma \vdash_{\textsc{c}} \ @\ l) \mid \square(\Gamma \vdash_{\textsc{c}} T\ @\ l) & \\
s, t := & \mathtt{Ty}_l \mid \mathtt{Nat} \mid \Pi^{l,l'}(x : s).t \mid x \mid u^{\delta} \mid \mathsf{zero} \mid \mathsf{succ}\ t & \text{(Terms)} \\
& \mid \lambda^{l,l'}(x : S).t \mid (t : \Pi^{l,l'}(x : S).T)\ s \mid \Lambda^l\ \overrightarrow{\ell}.t \mid t\ \$\ \overrightarrow{l} & \\
& \mid \Lambda^l\ g.t \mid t\ \$\ \Gamma \mid \Lambda^{l,l'}_{\textsc{d}}\ U.t \mid t\ \$_{\textsc{d}}\ T \mid \mathtt{box}\ T \mid \mathtt{box}\ t & \\
& \mid \mathtt{letbox}^{l'}_{x.M}\ U \leftarrow (t : \square(\Gamma \vdash_{\textsc{c}} \ @\ l))\ \mathsf{in}\ t' & \\
& \mid \mathtt{letbox}^{l'}_{x.M}\ u \leftarrow (t : \square(\Gamma \vdash_{\textsc{c}} T\ @\ l))\ \mathsf{in}\ t' & \\
& \mid \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (t : \square(\Gamma \vdash_{\textsc{c}} \ @\ l)) \mid \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (t : \square(\Gamma \vdash_{\textsc{c}} T\ @\ l)) & \\
\overrightarrow{M} := & (\ell, g, x_T.M_{\mathsf{Typ}})\ (\ell, g, U_T, x_t.M_{\mathsf{Trm}}) & \text{(Two motives for recursion on code)} \\
\overrightarrow{b} := & \overrightarrow{b}_{\mathsf{Typ}}\ \overrightarrow{b}_{\mathsf{Trm}} & \text{(Branches for recursion on code)} \\
b_{\mathsf{Typ}} := & (\ell, g.t_{\mathtt{Ty}}) \mid (g.t_{\mathtt{Nat}}) \mid (\ell, \ell', g, U_S, U_T, x_S, x_T.t_{\Pi}) \mid (\ell, g, u_t, x_t.t_{\mathtt{El}}) & \\
& & \text{(Branches for code of MLTT types)} \\
b_{\mathsf{Trm}} := & (\ell, g.t'_{\mathtt{Ty}}) \mid (g.t'_{\mathtt{Nat}}) \mid (\ell, \ell', g, u_s, u_t, x_s, x_t.t'_{\Pi}) \mid (\ell, g, U_T, u_x.t_x) & \\
& \mid (g.t_{\mathsf{zero}}) \mid (g, u_t, x_t.t_{\mathsf{succ}}) \mid (\ell, \ell', g, U_S, U_T, u_t, x_S, x_t.t_{\lambda}) & \\
& \mid (\ell, \ell', g, U_S, U_T, u_t, u_s, x_S, x_T, x_t, x_s.t_{\mathsf{app}}) & \\
& & \text{(Branches for code of MLTT terms)}
\end{array}
$$

Figure 5.3: Syntax of DeLaM

associated with each universe variable in l and l'. Moreover, universes form a partial order ($l \leq l' := l' = l \sqcup l'$) and a strict order ($l < l' := 1 + l \leq l'$). The strict order is well-founded. This fact will be used to define the logical relations to prove weak normalization and decidability of convertibility.

DeLaM supports a universe polymorphic functions space ($\overrightarrow{\ell} \Rightarrow^l T$); this type is introduced by abstractions ($\Lambda^l\ \overrightarrow{\ell}.t$) and used by applications ($t\ \$\ \overrightarrow{l}$).

5.2.2 Variables, Contexts and Substitutions

Following Chapter 4, DeLaM distinguishes regular variables (x) and meta-variable, which represent holes in code. There are three different kinds of meta-variables: g denotes a hole for an MLTT context, U denotes a hole for an MLTT type, and u denotes a hole for an MLTT term. The meta-variables for MLTT types and terms U and u are associated with regular substitutions as superscripts. I may omit writing the regular substitution if it is the identity id. To support universe polymorphism, I use ℓ to range over all universe variables. Three kinds of variables are stored in three kinds of contexts: regular contexts (Γ), meta-contexts (Ψ), and universe contexts (L). Universe contexts are just collections of universe variables. Meta-contexts are lists of meta-bindings B, which bind meta-variables to matching objects in MLTT. For regular contexts, due to context polymorphism, there are two base cases for a regular context: empty ($\cdot$) or a context variable (g) bound in meta-contexts. Correspondingly, there are also two base cases for a regular substitution. The empty substitution $\cdot$ maps to an empty context, while the weakening wk is the base case for context variables (c.f. Sec. 5.3.1).

Under the hood, I consider all variables are encoded in de Bruijn indices, so I can avoid dealing with issues about α-renaming.

5.2.3 Non-cumulative Tarski-Style Universes and Types

Instead of a more common Russell-style universe hierarchy, DeLaM employs a Tarski-style hierarchy (Palmgren, 1998). The Tarski style brings the syntax closer to the semantics than the Russell style and simplifies the semantic development. In the Tarski-style formulation, types and terms belong to two different but mutually defined grammars. Terms can refer to types in type annotations, while types can only refer to terms in the decoder $\mathtt{El}$. Dependent types are achieved by computing an encoding of a type as a term and using $\mathtt{El}$ for decoding. For example, $\mathtt{El}^0\ \mathtt{Nat}$ decodes to the type $\mathtt{Nat}$ and $\mathtt{El}^0\ (\Pi(x : \mathtt{Nat}).\mathtt{Nat})$ decodes to the type $\Pi(x : \mathtt{Nat}).\mathtt{Nat}$. As a consequence, $\mathtt{Ty}$, $\mathtt{Nat}$, and Π types are overloaded both as types and as terms. In addition, types also include meta-variable for types (U^δ), universe-polymorphic functions ($\overrightarrow{\ell} \Rightarrow^l T$), meta-functions for regular contexts ($(g : \mathsf{Ctx}) \Rightarrow^l T$), meta-functions for MLTT types

$((U : (\Gamma \vdash_{\textsc{d}} @\, l)) \Rightarrow^{l'} T)$ and contextual types ($\square(\Gamma \vdash_{\textsc{c}} @\, l)$ and $\square(\Gamma \vdash_{\textsc{c}} T @\, l)$) describing well-typed open code. Note that meta-functions for MLTT types only abstract over types from layer D, which do have non-trivial computational behaviors. As shown in Sec. 5.1, meta-functions are typically used to set up preliminaries in the typing context in order to describe contextual types, which denotes the actual code to manipulate.

In addition, the universe hierarchy in DeLaM is non-cumulative. Non-cumulativity ensures that all terms have unique types up to syntactic equivalence. The uniqueness is particularly convenient for the recursion principles for code, where subtyping induced by cumulativity causes unwanted complications. Due to the non-cumulative universe hierarchy, following Pujet and Tabareau (2023), I use $@\, l$ to mark the universe levels in contextual kinds and other places explicitly.

5.2.4 Dissecting Types and Terms of DeLaM

Disregarding universe levels for a moment, terms in DeLaM include MLTT objects like the encodings of types (e.g. natural numbers (`Nat`) and function types ($\Pi(x : s).t$)), function abstractions ($\lambda(x : S).t$), and function applications ($(t : \Pi(x : S).T)\ s$). Note that a function application includes the type annotation of t. This is necessary for quotation of terms and obtaining sufficient typing information of t. Without this annotation, one cannot generally derive what T is given only the overall type of the application. A recursor for natural numbers can be added in the usual way (see (Hu and Pientka, 2024a) and also the addition of a recursor for natural numbers in Chapter 4 and appendix D), but I omit it here for a more compact presentation.

Inspired by Cocon (Pientka et al., 2019), I include abstractions over regular contexts ($\Lambda\ g.t$) and over MLTT types ($\Lambda_{\textsc{d}}\ U.T$) to introduce meta-functions and applications of meta-functions ($t\ \$\ \Gamma$ and $t\ \$_{\textsc{d}}\ T$). In addition, following Chapter 4 and Hu and Pientka (2024b), I add the capability to quote code (`box` T and `box` t), to compose and execute code using `letbox`, and to recurse over code objects using `elim`.

Both `letbox` and the recursor are defined to account for possible type dependencies. As a consequence, both constructs are annotated with the overall types of the expressions (a.k.a. the motive M). In particular, `letbox` carries a motive annotation $x.M$. In the case of `elim`, there are two motive annotations $\overrightarrow{M}$, as `elim` defines a mutual recursion principle over code objects of MLTT types and terms at the same time. The mutual recursion also leads to two sets of branches $\overrightarrow{b}_{\mathsf{Typ}}$ and $\overrightarrow{b}_{\mathsf{Trm}}$. It may be surprising to see that one recursor includes two mutually defined recursion principles: one for types and one for terms. This comes from the fact that types and terms in DeLaM are also mutually defined due to the Tarski-style universe hierarchy. Hence, when analyzing a function application $(t : \Pi(x : S).T)\ s$, we may recursively analyze not only the terms t and s, but also the types T and S. Similarly, when analyzing a type of the form $\texttt{El}^l\ t$, we may recursively analyze the term t. This mutual dependency is the source of the complication in the recursor.

Intuitively, the branches in $\overrightarrow{b}_{\mathsf{Typ}}$ cover all possible MLTT types and those in $\overrightarrow{b}_{\mathsf{Trm}}$ cover all possible MLTT terms. $\overrightarrow{b}$ then collects both kinds of branches. The branches in $\overrightarrow{b}_{\mathsf{Typ}}$ cover the following cases: when elim encounters the code of a universe, it chooses branch $t_{\texttt{Ty}}$; when elim encounters the type $\texttt{Nat}$, it chooses branch $t_{\texttt{Nat}}$; when elim encounters a Π type, it chooses the branch t_{Π}; and when elim encounters a decoder $\texttt{El}$, it chooses the branch $t_{\texttt{El}}$. In $\overrightarrow{b}_{\mathsf{Trm}}$, there are cases for variables (t_x), natural numbers (t_{zero} and t_{succ}), function abstractions (t_λ), function applications (t_{app}), and the encodings of types ($t'_{\texttt{Ty}}$, $t'_{\texttt{Nat}}$ and t'_{Π}). In each branch, there are three groups of variables. The first group is the pattern variables of the pattern being considered. For example, in the branch t_{succ}, the pattern variable u_t describes the pattern variable in the pattern $\mathsf{succ}\ u_t$. Similarly, in the branch t_{app}, the pattern would be $(u_t : \Pi(x : u_S).u_T)\ u_s$, so there are pattern variables for the function u_t and the argument u_s, as well as for the type annotations u_S and u_T. Since the regular context might grow in the Π case and in the λ case, each branch maintains a pattern variable g for the regular context. As the second group of variables, each branch includes recursion variables x for recursive calls. In the succ case, x_t refers to the recursive call over u_t when elim encounters the pattern $\mathsf{succ}\ u_t$. In the application case, there are four recursive calls: x_t corresponds to the recursive call on the function u_t; x_s corresponds to the recursive call on the argument u_s; and x_T and x_S correspond to the recursive calls on the types u_S and u_T respectively. At last, each branch might introduce universe variables as the third group of variables to quantify the universe levels of pattern variables. The variables in the branches $\overrightarrow{b}_{\mathsf{Typ}}$ are organized in a similar principle.

Layer i	variables ($\textsc{v}$)	code object ($\textsc{c}$)	MLTT ($\textsc{d}$)	meta-program ($\textsc{m}$)
Syntax	Variables only	MLTT	MLTT	MLTT + meta-programming
Computation	No	No	Yes	Yes
Meta-programming capabilities	No	No	No	Yes
$\Uparrow(i)$	$\textsc{d}$	$\textsc{d}$	$\textsc{d}$	$\textsc{m}$

Table 5.1: Characteristics of each layer

5.3 Syntactic Judgments in DeLaM

One advantage of DeLaM is its uniform syntax. The distinction of valid types and terms is only made in the judgments by layers. Most syntactic judgments in DeLaM are parameterized by a layer i. Through the layer i, I define rules that generically hold at multiple layers and rules that only exist at a specific layer. Layers control not only the validity of objects, but also the computational behaviors. In this way, I cleanly distinguish the layers $\textsc{v}$ and $\textsc{c}$ for code objects, the layer $\textsc{d}$ for the dependent type theory of MLTT, and the layer $\textsc{m}$ of meta-programming. Following Pientka et al. (2019); Cave and Pientka (2012), the layer $\textsc{v}$ is introduced to only describe static code objects for MLTT variables. This layer only appears in the pattern variable when the recursor on code of terms hits the case for variables (t_x) (c.f. Appendix F.2.1). Based on the matryoshka principle, these layers form a strict order: $\textsc{v} < \textsc{c} < \textsc{d} < \textsc{m}$. As explained at the beginning of this chapter, the computational power strictly increases as the layer increases. In particular, code objects at layer $\textsc{c}$ do not compute, so that the recursion principles are applied to the syntactic shapes of code. On the other hand, code promotion may bring a code object to layer $\textsc{d}$, where computation may occur. The topmost layer $\textsc{m}$ further extends the layer $\textsc{d}$ with computational constructs that composes, executes and does recursion on code.

Most syntactic judgments are defined parametrically in layer i. Here the layer i refers to the layer which the *principal object* lives at. For example, $L \mid \Psi; \Gamma \vdash_i t : T \,@\, l$ defines that t is well-typed at layer i. In this case, the regular context Γ and the type

T live at a higher layer than i. For example, if $i = \textsc{c}$, then t is a code object, and therefore its type T lives at layer $\textsc{d}$. I define the function $\Uparrow(i)$ to compute the layer of the surrounding typing environment when the principle object lives at layer i:

$$\Uparrow(\textsc{m}) := \textsc{m} \qquad\qquad \Uparrow(i) := \textsc{d} \quad \text{if } i \neq \textsc{m}$$

Presupposition formally captures the purpose of $\Uparrow(i)$ (c.f. Lemma 5.3). The relation between layers is summarized in Table 5.1. In this section, I discuss below a few selected rules for each judgment in DeLaM. I highlight the principal object in the informal explanation for each judgment in $\boxed{\text{shades}}$. For conciseness, I assume all parameterized universes l to be well-formed.

5.3.1 Well-Formed Regular and Meta-Context

Let us begin with the discussion on the well-formedness of meta-and regular contexts w.r.t. a universe context L. A meta-context Ψ is well-formed, if every meta-binding is well-formed. In particular, $U : (\Gamma \vdash_i \ @\, l)$ is well-formed, if the regular context Γ is well-formed at the higher layer $\Uparrow(i)$ and $i \in \{\textsc{c}, \textsc{d}\}$; $u : (\Gamma \vdash_i \ @\, T) l$ is well-formed, if the type T is well-formed at layer $\Uparrow(i)$ and $i \in \{\textsc{v}, \textsc{c}\}$. Note that the value of i has a fixed range in both cases. I could have written $\textsc{d}$ instead of $\Uparrow(i)$, but the current formulation is very convenient for understanding the presupposition lemma (Lemma 5.3). A regular context Γ is well-formed, if every type declaration $x : T \,@\, l$ in Γ is well formed.

$\boxed{L \vdash \Psi}$ Meta-Context $\boxed{\Psi}$ is wf

$$\frac{}{L \vdash \cdot} \qquad \frac{L \vdash \Psi}{L \vdash \Psi, g : \mathsf{Ctx}} \qquad \frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma \quad i \in \{\textsc{c}, \textsc{d}\}}{L \vdash \Psi, U : (\Gamma \vdash_i \ @\, l)} \qquad \frac{L \mid \Psi; \Gamma \vdash_{\Uparrow(i)} T \,@\, l \quad i \in \{\textsc{v}, \textsc{c}\}}{L \vdash \Psi, u : (\Gamma \vdash_i T \,@\, l)}$$

$\boxed{L \mid \Psi \vdash_i \Gamma}$ At layer i the regular context $\boxed{\Gamma}$ is wf

$$\frac{L \vdash \Psi}{L \mid \Psi \vdash_i \cdot} \qquad \frac{L \vdash \Psi \quad g : \mathsf{Ctx} \in \Psi}{L \mid \Psi \vdash_i g} \qquad \frac{L \mid \Psi \vdash_i \Gamma \quad L \mid \Psi; \Gamma \vdash_i T \,@\, l}{L \mid \Psi \vdash_i \Gamma, x : T \,@\, l}$$

A regular substitution δ is well-formed, if all terms within are well-formed.

$\boxed{L \mid \Psi; \Gamma \vdash_i \delta : \Delta}$ At layer i regular substitution $\boxed{\delta}$ substitutes variables in Δ with terms in Γ

$$\frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma}{L \mid \Psi; \Gamma \vdash_i \cdot : \cdot} \qquad \frac{L \mid \Psi \vdash_{\Uparrow(i)} (g, \Gamma)}{L \mid \Psi; g, \Gamma \vdash_i \mathsf{wk} : g} \qquad \frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_i \delta : \Delta \\ L \mid \Psi; \Delta \vdash_{\Uparrow(i)} T \mathbin{@} l \quad L \mid \Psi; \Gamma \vdash_i t : T[\delta] \mathbin{@} l \end{array}}{L \mid \Psi; \Gamma \vdash_i \delta, t/x : \Delta, x : T \mathbin{@} l}$$

Note that for wk to be well-formed, it can only weaken the context g, Γ to the same context variable g. The symmetrized variants $\boxed{L \mid \Psi \vdash_i \Gamma \approx \Delta}$ and $\boxed{L \mid \Psi; \Gamma \vdash_i \delta \approx \delta' : \Delta}$ are pointwise generalizations of $L \mid \Psi \vdash_i \Gamma$ and $L \mid \Psi; \Gamma \vdash_i \delta : \Delta$.

5.3.2 Types and Terms

Now let us consider the judgments for types and terms. I first define the well-formedness of types T at layer i. By controlling layer i, the well-formedness judgment controls what types are available. For example, contextual types $\square(\Gamma \vdash_{\text{C}} \mathbin{@} l)$ and $\square(\Gamma \vdash_{\text{C}} T \mathbin{@} l)$ are only available at layer M and hence all lower layers (V, C, D) only have access to MLTT terms and types.

In the rules for types, if a rule is parameterized by a layer i, then this rule is available at all *possible* layers. For example, the meta-and regular variable rules are available at all four layers. The Π rule is also parameterized by i, but since Π is clearly not a variable, Π types are available as code objects at layer C, as types in MLTT at layer D, and as types at layer M. Due to Tarski universes à la Palmgren (1998), $\mathtt{El}^l\ t$ decodes the encoding $t : \mathtt{Ty}_l$ to an actual type. If U is a meta-variable bound to a type at layer i' in the meta-context Ψ, then U^δ is a valid type at any layer $i \geq i'$. The condition $i \geq i'$ encodes the lifting lemma (c.f. Lemma 5.1), which allows an object from a lower layer to be lifted to any higher layer freely. The associated regular substitution δ is needed to instantiate open variables associated with U. On the other hand, rules for types that provide meta-programming facilities, like contextual types and meta-functions, are only available at layer M. Note that contextual types $\square(\Gamma \vdash_{\text{C}} \mathbin{@} l)$ and $\square(\Gamma \vdash_{\text{C}} T \mathbin{@} l)$ live on universe level 0 regardless of the universe level l. This is because contextual types encode intrinsically typed syntax trees of MLTT objects at layer M, which corresponds to a logically stronger sub-language than MLTT, so universe level 0 is large enough to encode all well-formed types and terms of MLTT. This observation is modeled in the semantics (c.f. Sec. 5.4.6), where contextual types need not refer to other semantics at layer $\textsc{m}$, so its semantics can be placed on level 0. Finally, for universe-polymorphic functions $\overrightarrow{\ell} \Rightarrow^l T$, they can only live on universe level ω, similar to Agda, and the universe level l may refer to $\overrightarrow{\ell}$ and is the universe level of T. This fact is explicitly encoded by the condition $L, \overrightarrow{\ell} \vdash l : \mathsf{Level}$, which is the well-formedness judgment for universe levels. Due to this condition, l cannot contain ω, so universe-polymorphic functions must introduce all universe variables at once.

$\boxed{L \mid \Psi; \Gamma \vdash_i T \,@\, l}$ At layer i type $\boxed{T}$ is wf on universe level l

$$\frac{L \mid \Psi; \Gamma \vdash_i S \,@\, l \qquad L \mid \Psi; \Gamma, x : S \,@\, l \vdash_i T \,@\, l'}{L \mid \Psi; \Gamma \vdash_i \Pi^{l,l'}(x : S).T \,@\, l \sqcup l'} \qquad \frac{L \mid \Psi; \Gamma \vdash_i t : \mathtt{Ty}_l \,@\, 1 + l}{L \mid \Psi; \Gamma \vdash_i \mathtt{El}^l\ t \,@\, l}$$

$$\frac{U : (\Delta \vdash_{i'} \,@\, l) \in \Psi \qquad i' \in \{\textsc{c}, \textsc{d}\} \qquad i' \leq i \qquad L \mid \Psi; \Gamma \vdash_i \delta : \Delta}{L \mid \Psi; \Gamma \vdash_i U^\delta \,@\, l}$$

$$\frac{L \mid \Psi \vdash_{\textsc{m}} \Gamma \qquad L \mid \Psi \vdash_{\textsc{d}} \Delta}{L \mid \Psi; \Gamma \vdash_{\textsc{m}} \square(\Delta \vdash_{\textsc{c}} \,@\, l) \,@\, 0} \qquad \frac{L \mid \Psi \vdash_{\textsc{m}} \Gamma \qquad L \mid \Psi; \Delta \vdash_{\textsc{d}} T \,@\, l}{L \mid \Psi; \Gamma \vdash_{\textsc{m}} \square(\Delta \vdash_{\textsc{c}} T \,@\, l) \,@\, 0}$$

$$\frac{L \mid \Psi, g : \mathsf{Ctx}; \Gamma \vdash_{\textsc{m}} T \,@\, l}{L \mid \Psi; \Gamma \vdash_{\textsc{m}} (g : \mathsf{Ctx}) \Rightarrow^l T \,@\, l} \qquad \frac{L \mid \Psi \vdash_{\textsc{d}} \Delta \qquad L \mid \Psi, U : (\Delta \vdash_{\textsc{d}} \,@\, l); \Gamma \vdash_{\textsc{m}} T \,@\, l'}{L \mid \Psi; \Gamma \vdash_{\textsc{m}} (U : (\Delta \vdash_{\textsc{d}} \,@\, l)) \Rightarrow^{l'} T \,@\, l}$$

$$\frac{L, \overrightarrow{\ell} \mid \Psi; \Gamma \vdash_{\textsc{m}} T \,@\, l \qquad |\overrightarrow{\ell}| > 0 \qquad L, \overrightarrow{\ell} \vdash l : \mathsf{Level}}{L \mid \Psi; \Gamma \vdash_{\textsc{m}} \overrightarrow{\ell} \Rightarrow^l T \,@\, \omega}$$

Following the same principle, I define the typing rules in Fig. 5.4. Well-typed terms in vanilla MLTT are defined parametrically in layer i. These terms include encodings of types, natural numbers, functions, and variables. The same as U^δ, a meta-variable u is also associated with a regular substitution δ to fill in the open variables. The premise $i' \leq i$ also builds the lifting lemma into the typing rule. Terms related to meta-programming are only available at layer $\textsc{m}$, e.g. meta-functions, the `box` constructor and `letbox`. For example, to introduce a meta-function for regular contexts, I use Λ to introduce a context variable g. I use \$ to apply a meta-function to a regular context. Note that the regular context Δ lives at layer $\textsc{d}$, so it is ensured a context in MLTT. The core syntax of `letbox` requires a motive $x.M$, which computes the result type. The

$\boxed{L \mid \Psi; \Gamma \vdash_i t : T \,@\, l}$ At layer i term t has type T at universe level l

$$\frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma}{L \mid \Psi; \Gamma \vdash_i \mathtt{Ty}_l : \mathtt{Ty}_{1+l} \,@\, 2+l} \qquad \frac{u : (\Delta \vdash_{i'} T \,@\, l) \in \Psi \qquad i' \leq i \qquad L \mid \Psi; \Gamma \vdash_i \delta : \Delta}{L \mid \Psi; \Gamma \vdash_i u^{\delta} : T[\delta] \,@\, l}$$

$$\frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma \qquad x : T \,@\, l \in \Gamma}{L \mid \Psi; \Gamma \vdash_i x : T \,@\, l} \qquad \frac{L \mid \Psi; \Gamma \vdash_i S \,@\, l \qquad L \mid \Psi; \Gamma, x : S \,@\, l \vdash_i t : T \,@\, l'}{L \mid \Psi; \Gamma \vdash_i \lambda^{l,l'}(x : S).t : \Pi^{l,l'}(x : S).T \,@\, l \sqcup l'}$$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_i S \,@\, l \\ L \mid \Psi; \Gamma, x : S \,@\, l \vdash_i T \,@\, l' \qquad L \mid \Psi; \Gamma \vdash_i t : \Pi^{l,l'}(x : S).T \,@\, l \sqcup l' \qquad L \mid \Psi; \Gamma \vdash_i s : S \,@\, l \end{array}}{L \mid \Psi; \Gamma \vdash_i (t : \Pi^{l,l'}(x : S).T)\ s : T[s/x] \,@\, l'}$$

$$\frac{\begin{array}{c} L \mid \Psi \vdash_{\mathrm{M}} \Gamma \\ L \mid \Psi, g : \mathsf{Ctx}; \Gamma \vdash_{\mathrm{M}} t : T \,@\, l \end{array}}{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} \Lambda^l\ g.t : (g : \mathsf{Ctx}) \Rightarrow^l T \,@\, l} \qquad \frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_{\mathrm{M}} t : (g : \mathsf{Ctx}) \Rightarrow^l T \,@\, l \\ L \mid \Psi \vdash_{\mathrm{D}} \Delta \end{array}}{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} t \ \$\ \Delta : T[\Delta/g] \,@\, l}$$

$$\frac{L \mid \Psi \vdash_{\mathrm{M}} \Gamma \qquad L \mid \Psi; \Delta \vdash_{\mathrm{C}} T \,@\, l}{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} \mathtt{box}\ T : \square(\Delta \vdash_{\mathrm{C}} \,@\, l) \,@\, 0} \qquad \frac{L \mid \Psi \vdash_{\mathrm{M}} \Gamma \qquad L \mid \Psi; \Delta \vdash_{\mathrm{C}} t : T \,@\, l}{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} \mathtt{box}\ t : \square(\Delta \vdash_{\mathrm{C}} T \,@\, l) \,@\, 0}$$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_{\mathrm{M}} t : \square(\Delta \vdash_{\mathrm{C}} \,@\, l) \,@\, 0 \qquad L \mid \Psi; \Gamma, x : \square(\Delta \vdash_{\mathrm{C}} \,@\, l) \,@\, 0 \vdash_{\mathrm{M}} M \,@\, l' \\ L \mid \Psi, U : (\Delta \vdash_{\mathrm{C}} \,@\, l); \Gamma \vdash_{\mathrm{M}} t' : M[\mathtt{box}\ U^{\mathsf{id}}/x] \,@\, l' \end{array}}{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} \mathtt{letbox}^{l'}_{x.M}\ U \leftarrow (t : \square(\Delta \vdash_{\mathrm{C}} \,@\, l))\ \mathsf{in}\ t' : M[t/x] \,@\, l'}$$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_{\mathrm{M}} t : \square(\Delta \vdash_{\mathrm{C}} T \,@\, l) \,@\, l \qquad L \mid \Psi; \Gamma, x : \square(\Delta \vdash_{\mathrm{C}} T \,@\, l) \,@\, 0 \vdash_{\mathrm{M}} M \,@\, l' \\ L \mid \Psi, u : (\Delta \vdash_{\mathrm{C}} T \,@\, l); \Gamma \vdash_{\mathrm{M}} t' : M[\mathtt{box}\ u^{\mathsf{id}}/x] \,@\, l' \end{array}}{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} \mathtt{letbox}^{l'}_{x.M}\ u \leftarrow (t : \square(\Delta \vdash_{\mathrm{C}} T \,@\, l))\ \mathsf{in}\ t' : M[t/x] \,@\, l'}$$

$$\frac{L, \overrightarrow{\ell} \mid \Psi; \Gamma \vdash_{\mathrm{M}} t : T \,@\, l \qquad |\overrightarrow{\ell}| > 0 \qquad L, \overrightarrow{\ell} \vdash l : \mathsf{Level}}{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} \Lambda^l\ \overrightarrow{\ell}.t : \overrightarrow{\ell} \Rightarrow^l T \,@\, \omega}$$

$$\frac{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} t : \overrightarrow{\ell} \Rightarrow^l T \,@\, \omega \qquad |\overrightarrow{\ell}| = |\overrightarrow{l}| > 0 \qquad \forall\, l' \in \overrightarrow{l}\ .\ L \vdash l' : \mathsf{Level}}{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} t \ \$\ \overrightarrow{l} : T[\overrightarrow{l}/\overrightarrow{\ell}] \,@\, l[\overrightarrow{l}/\overrightarrow{\ell}]}$$

Figure 5.4: Typing rules for terms

`letbox` construct eliminates a term of a contextual type and binds it to a meta-variable. If this term has a contextual type for types, then `letbox` introduces a new meta-variable for types U; if this term has a contextual type for terms, then `letbox` introduces a new meta-varible for terms u. The `letbox` body t lives in an extended meta-context with this new meta-variable and has type M with x substituted with it. At last, to eliminate a universe-polymorphic functions, I pass in a list of universes as $\overrightarrow{l}$. In addition to the type, the result universe level is also substituted, i.e. becomes $l[\overrightarrow{l}/\overrightarrow{\ell}]$. Since l and all universe levels in $\overrightarrow{l}$ are well-formed, the result of this substitution is also well-formed.

Finally, I discuss some selected equivalence rules. I focus here on term equivalences and omit the equivalences for types $\boxed{L \mid \Psi; \Gamma \vdash_i T \approx T' \mathbin{@} l}$, where the only non-trivial rules are decoding rules. All equivalence judgments include symmetry, transitivity, and naturally derived congruence rules at all layers. This follows exactly the same principle as outlined in Chapter 1. I show here two β rules and one η rule. In DeLaM, no computation rule is available at layers $\textsc{v}$ and $\textsc{c}$. Computation rules for terms in MLTT like the β rule and the η rule for functions are available at both layers $\textsc{d}$ and $\textsc{m}$ to handle both code promotion and code execution. Rules for meta-programs like the β rule for `letbox` and for recursors are only available at layer $\textsc{m}$. Meta-functions and universe-polymorphic functions also enjoy η equivalence, which is quite natural so I omit the rules here.

$\boxed{L \mid \Psi; \Gamma \vdash_i t \approx t' : T \mathbin{@} l}$ At layer i the term $\boxed{t}$ is equivalent to the term $\boxed{t'}$

$$\frac{i \in \{\textsc{d}, \textsc{m}\} \qquad L \mid \Psi; \Gamma \vdash_i S \mathbin{@} l \qquad L \mid \Psi; \Gamma, x : S \mathbin{@} l \vdash_i t : T \mathbin{@} l' \qquad L \mid \Psi; \Gamma \vdash_i s : S \mathbin{@} l}{L \mid \Psi; \Gamma \vdash_i (\lambda^{l,l'}(x : S).t : \Pi^{l,l'}(x : S).T)\ s \approx t[s/x] : T[s/x] \mathbin{@} l'}$$

$$\frac{i \in \{\textsc{d}, \textsc{m}\} \qquad L \mid \Psi; \Gamma \vdash_i t : \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l'}{L \mid \Psi; \Gamma \vdash_i t \approx \lambda^{l,l'}(x : S).(t : \Pi^{l,l'}(x : S).T)\ x : \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l'}$$

$$\frac{\begin{array}{c} L \mid \Psi \vdash_{\textsc{m}} \Gamma \qquad L \mid \Psi; \Delta \vdash_{\textsc{c}} t : T \mathbin{@} l \qquad L \mid \Psi; \Gamma, x_t : \Box(\Delta \vdash_{\textsc{c}} T \mathbin{@} l) \mathbin{@} 0 \vdash_{\textsc{m}} M \mathbin{@} l' \\ L \mid \Psi, u : (\Delta \vdash_{\textsc{c}} T \mathbin{@} l); \Gamma \vdash_{\textsc{m}} t' : M[\mathtt{box}\ u^{\mathsf{id}}/x_t] \mathbin{@} l' \end{array}}{L \mid \Psi; \Gamma \vdash_{\textsc{m}} \mathtt{letbox}^{l'}_{x_t.M}\ u \leftarrow ((\mathtt{box}\ t) : \Box(\Delta \vdash_{\textsc{c}} T \mathbin{@} l))\ \mathsf{in}\ t' \approx t'[t/u] : M[\mathtt{box}\ t/x_t] \mathbin{@} l'}$$

5.3.3 Static Code and Lifting Lemma

Similar to Chapter 4, layered modal type theories have two guiding lemmas: the *lifting lemma* and the *static code lemma*. The guiding lemmas are syntactic lemmas that can be checked at the early stage of the technical investigation, and serve the purpose of guiding the design of the type theory. The lifting lemma says that a well-typed term at a lower layer is also well-typed at higher layers. The static code lemma states that equivalence between code objects is syntactic equality.

Lemma 5.1 (Lifting). *If $i \leq i'$, and*

- $L \mid \Psi; \Gamma \vdash_i T \mathbin{@} l$, *then* $L \mid \Psi; \Gamma \vdash_{i'} T \mathbin{@} l$;
- $L \mid \Psi; \Gamma \vdash_i T \approx T' \mathbin{@} l$, *then* $L \mid \Psi; \Gamma \vdash_{i'} T \approx T' \mathbin{@} l$;
- $L \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l$, *then* $L \mid \Psi; \Gamma \vdash_{i'} t : T \mathbin{@} l$;
- $L \mid \Psi; \Gamma \vdash_i t \approx t' : T \mathbin{@} l$, *then* $L \mid \Psi; \Gamma \vdash_{i'} t \approx t' : T \mathbin{@} l$;
- $L \mid \Psi; \Gamma \vdash_i \delta : \Delta$, *then* $L \mid \Psi; \Gamma \vdash_{i'} \delta : \Delta$;
- $L \mid \Psi; \Gamma \vdash_i \delta \approx \delta' : \Delta$, *then* $L \mid \Psi; \Gamma \vdash_{i'} \delta \approx \delta' : \Delta$.

Lemma 5.2 (Static Code). *If $i \in \{\textsc{v}, \textsc{c}\}$, and*

- $L \mid \Psi; \Gamma \vdash_i T \approx T' \mathbin{@} l$, *then* $T = T'$;
- $L \mid \Psi; \Gamma \vdash_i t \approx t' : T \mathbin{@} l$, *then* $t = t'$;
- $L \mid \Psi; \Gamma \vdash_i \delta \approx \delta' : \Delta$, *then* $\delta = \delta'$.

A consequence of the lifting lemma is that code objects at layer c can be lifted to layers d and m for free. It justifies *code execution* by using the lifting lemma to execute a code object of layer c at layer m. It also enables *code promotion*, which promotes the syntactic representation of a code object at layer c to layer d to capture computation of MLTT on the type level at layer d.

One consequence of the static code lemma is no interesting computational behavior at layers v and c, so terms at both layers describe the static syntax of code objects. This is a necessary condition to intensionally analyze code from layers c and v.

Presupposition is another important syntactic property for DeLaM. It says that if the principal object is being defined at layer i, regular contexts and types live at layer $\Uparrow(i)$.

Lemma 5.3 (Presupposition)**.**

- *If* $L \mid \Psi; \Gamma \vdash_i T \mathbin{@} l$*, then* $L \mid \Psi \vdash_{\Uparrow(i)} \Gamma$ *and either*
  - $L \vdash l : \mathsf{Level}$ *or*
  - $i = \textsc{m}$ *and* $l = \omega$.
- *If* $L \mid \Psi; \Gamma \vdash_i T \approx T' \mathbin{@} l$*, then* $L \mid \Psi \vdash_{\Uparrow(i)} \Gamma$*,* $L \mid \Psi; \Gamma \vdash_i T \mathbin{@} l$*,* $L \mid \Psi; \Gamma \vdash_i T' \mathbin{@} l$ *and either*
  - $L \vdash l : \mathsf{Level}$ *or*
  - $i = \textsc{m}$ *and* $l = \omega$.
- *If* $L \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l$*, then* $L \mid \Psi \vdash_{\Uparrow(i)} \Gamma$*,* $L \mid \Psi; \Gamma \vdash_{\Uparrow(i)} T \mathbin{@} l$ *and either*
  - $L \vdash l : \mathsf{Level}$ *or*
  - $i = \textsc{m}$ *and* $l = \omega$.
- *If* $L \mid \Psi; \Gamma \vdash_i t \approx t' : T \mathbin{@} l$*, then* $L \mid \Psi \vdash_{\Uparrow(i)} \Gamma$*,* $L \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l$*,* $L \mid \Psi; \Gamma \vdash_i t' : T \mathbin{@} l$*,* $L \mid \Psi; \Gamma \vdash_{\Uparrow(i)} T \mathbin{@} l$ *and either*
  - $L \vdash l : \mathsf{Level}$ *or*
  - $i = \textsc{m}$ *and* $l = \omega$.
- *If* $L \mid \Psi; \Gamma \vdash_i \delta : \Delta$*, then* $L \mid \Psi \vdash_{\Uparrow(i)} \Gamma$ *and* $L \mid \Psi \vdash_{\Uparrow(i)} \Delta$.
- *If* $L \mid \Psi; \Gamma \vdash_i \delta \approx \delta' : \Gamma'$*, then* $L \mid \Psi \vdash_{\Uparrow(i)} \Gamma$*,* $L \mid \Psi \vdash_{\Uparrow(i)} \Delta$*,* $L \mid \Psi; \Gamma \vdash_i \delta : \Delta$ *and* $L \mid \Psi; \Gamma \vdash_i \delta' : \Delta$.

5.3.4 Universe, Regular and Meta-Substitutions

Other than regular substitutions, which substitute regular variables with terms, DeLaM also needs universe and meta-substitutions to substitute universe and meta-variables. A universe substitution ϕ replaces universe variables in one universe context with universe levels in another, where ϕ is effectively just a list of universe levels. The well-formedness judgment $\boxed{L \vdash \phi : L'}$ requires all universe levels in ϕ are well-formed. Various kinds of substitutions are defined in the usual way. Since they are very long and tedious, I omit the definitions here. Full definitions can be found in (Hu and Pientka, 2024a, Sec. 5.1). The universe substitution lemma says that syntactic judgments are closed under universe substitutions:

Lemma 5.4 (Universe Substitutions)**.**

- *If* $L' \mid \Psi; \Gamma \vdash_i T \mathbin{@} l$ *and* $L \vdash \phi : L'$*, then* $L \mid \Psi[\phi]; \Gamma[\phi] \vdash_i T[\phi] \mathbin{@} l[\phi]$*.*
- *If* $L' \mid \Psi; \Gamma \vdash_i T \approx T' \mathbin{@} l$ *and* $L \vdash \phi : L'$*, then* $L \mid \Psi[\phi]; \Gamma[\phi] \vdash_i T[\phi] \approx T'[\phi] \mathbin{@} l[\phi]$*.*
- *If* $L' \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l$ *and* $L \vdash \phi : L'$*, then* $L \mid \Psi[\phi]; \Gamma[\phi] \vdash_i t[\phi] : T[\phi] \mathbin{@} l[\phi]$*.*
- *If* $L' \mid \Psi; \Gamma \vdash_i t \approx t' : T \mathbin{@} l$ *and* $L \vdash \phi : L'$*, then*
  $L \mid \Psi[\phi]; \Gamma[\phi] \vdash_i t[\phi] \approx t'[\phi] : T[\phi] \mathbin{@} l[\phi]$*.*
- *If* $L' \mid \Psi; \Gamma \vdash_i \delta : \Gamma'$ *and* $L \vdash \phi : L'$*, then* $L \mid \Psi[\phi]; \Gamma[\phi] \vdash_i \delta[\phi] : \Gamma'[\phi]$*.*
- *If* $L' \mid \Psi; \Gamma \vdash_i \delta \approx \delta' : \Gamma'$ *and* $L \vdash \phi : L'$*, then* $L \mid \Psi[\phi]; \Gamma[\phi] \vdash_i \delta[\phi] \approx \delta'[\phi] : \Gamma'[\phi]$*.*

The regular substitution lemma is standard. It only changes the regular context.

Lemma 5.5 (Regular Substitutions)**.**

- *If* $L \mid \Psi; \Gamma' \vdash_i T \mathbin{@} l$ *and* $L \mid \Psi; \Gamma \vdash_i \delta : \Gamma'$*, then* $L \mid \Psi; \Gamma \vdash_i T[\delta] \mathbin{@} l$*.*
- *If* $L \mid \Psi; \Gamma' \vdash_i T \approx T' \mathbin{@} l$ *and* $L \mid \Psi; \Gamma \vdash_i \delta : \Gamma'$*, then* $L \mid \Psi; \Gamma \vdash_i T[\delta] \approx T'[\delta] \mathbin{@} l$*.*
- *If* $L \mid \Psi; \Gamma' \vdash_i t : T \mathbin{@} l$ *and* $L \mid \Psi; \Gamma \vdash_i \delta : \Gamma'$*, then* $L \mid \Psi; \Gamma \vdash_i t[\delta] : T[\delta] \mathbin{@} l$*.*
- *If* $L \mid \Psi; \Gamma' \vdash_i t \approx t' : T \mathbin{@} l$ *and* $L \mid \Psi; \Gamma \vdash_i \delta : \Gamma'$*, then*
  $L \mid \Psi; \Gamma \vdash_i t[\delta] \approx t'[\delta] : T[\delta] \mathbin{@} l$*.*

- *If* $L \mid \Psi; \Gamma' \vdash_i \delta' : \Gamma''$ *and* $L \mid \Psi; \Gamma \vdash_i \delta : \Gamma'$, *then* $L \mid \Psi; \Gamma \vdash_i \delta' \circ \delta : \Gamma''$.
- *If* $L \mid \Psi; \Gamma' \vdash_i \delta' \approx \delta'' : \Gamma''$ *and* $L \mid \Psi; \Gamma \vdash_i \delta : \Gamma'$, *then* $L \mid \Psi; \Gamma \vdash_i \delta' \circ \delta \approx \delta'' \circ \delta : \Gamma''$.

Meta-substitutions in DeLaM is very similar to Sec. 4.4. They replace MLTT objects accordingly. The base cases are:

$$\begin{aligned} g[\sigma] &:= \sigma(g) \\ (\Gamma, x : T \mathbin{@} l)[\sigma] &:= \Gamma[\sigma], x : T[\sigma] \mathbin{@} l \\ U^\delta[\sigma] &:= \sigma(U)[\delta[\sigma]] \\ u^\delta[\sigma] &:= \sigma(u)[\delta[\sigma]] \end{aligned}$$

In the cases of U^δ and u^δ, the action of σ triggers the action of the regular substitution $\delta[\sigma]$. Therefore, the action of regular substitutions and its properties must be reasoned prior to those of meta-substitutions. The well-formedness of meta-substitutions is defined by the judgment $\boxed{L \mid \Psi \vdash \sigma : \Phi}$ as:

$$\frac{L \vdash \Psi}{L \mid \Psi \vdash \cdot : \cdot} \qquad \frac{L \mid \Psi \vdash \sigma : \Phi \qquad L \mid \Psi \vdash_{\text{D}} \Gamma}{L \mid \Psi \vdash \sigma, \Gamma/g : \Phi, g : \mathsf{Ctx}} \qquad \frac{\begin{array}{c} L \mid \Psi \vdash \sigma : \Phi \qquad L \mid \Phi \vdash_{\text{D}} \Gamma \\ i \in \{\text{C}, \text{D}\} \qquad L \mid \Psi; \Gamma[\sigma] \vdash_i T \mathbin{@} l \end{array}}{L \mid \Psi \vdash \sigma, T/U : \Phi, u : (\Gamma \vdash_i \mathbin{@} l)}$$

$$\frac{L \mid \Psi \vdash \sigma : \Phi \qquad L \mid \Phi; \Gamma \vdash_{\text{D}} T \mathbin{@} l \qquad i \in \{\text{V}, \text{C}\} \qquad L \mid \Psi; \Gamma[\sigma] \vdash_i t : T[\sigma] \mathbin{@} l}{L \mid \Psi \vdash \sigma, t/u : \Phi, u : (\Gamma \vdash_i T \mathbin{@} l)}$$

The equivalence between meta-substitutions simply takes the congruence of these rules. The meta-substitution lemma says that syntactic judgments are closed under meta-substitutions:

Lemma 5.6 (Meta-substitutions)**.**

- *If* $L \mid \Phi; \Gamma \vdash_i T \mathbin{@} l$ *and* $L \mid \Psi \vdash \sigma : \Phi$, *then* $L \mid \Psi; \Gamma[\sigma] \vdash_i T[\sigma] \mathbin{@} l$.
- *If* $L \mid \Phi; \Gamma \vdash_i T \approx T' \mathbin{@} l$ *and* $L \mid \Psi \vdash \sigma : \Phi$, *then* $L \mid \Psi; \Gamma[\sigma] \vdash_i T[\sigma] \approx T'[\sigma] \mathbin{@} l$.
- *If* $L \mid \Phi; \Gamma \vdash_i t : T \mathbin{@} l$ *and* $L \mid \Psi \vdash \sigma : \Phi$, *then* $L \mid \Psi; \Gamma[\sigma] \vdash_i t[\sigma] : T[\sigma] \mathbin{@} l$.

- *If* $L \mid \Phi; \Gamma \vdash_i t \approx t' : T \mathbin{@} l$ *and* $L \mid \Psi \vdash \sigma : \Phi$*, then* $L \mid \Psi; \Gamma[\sigma] \vdash_i t[\sigma] \approx t'[\sigma] : T[\sigma] \mathbin{@} l$*.*

- *If* $L \mid \Phi; \Gamma \vdash_i \delta : \Delta$ *and* $L \mid \Psi \vdash \sigma : \Phi$*, then* $L \mid \Psi; \Gamma[\sigma] \vdash_i \delta[\sigma] : \Delta[\sigma]$*.*

- *If* $L \mid \Phi; \Gamma \vdash_i \delta \approx \delta' : \Delta$ *and* $L \mid \Psi \vdash \sigma : \Phi$*, then* $L \mid \Psi; \Gamma[\sigma] \vdash_i \delta[\sigma] \approx \delta'[\sigma] : \Delta[\sigma]$*.*

It is worth noting that the process of proving the meta-substitution lemma is actually quite cumbersome. In the case for meta-variables, this lemma requires the regular substitution lemma as a prerequisite. However, the proof of the regular substitution lemma also requires reasoning about meta-substitutions in multiple cases, e.g. in the meta-function application rule in Fig. 5.4, where a meta-substitution is applied to the type, before the meta-substitution lemma is established. This requirement is eventually reflected as a demand of an agreement of "biases" in equivalence rules. One immediate way to arrive at such an agreement is to swap both sides of the computational equivalence rules of terms above. This treatment is not a common pattern in other type theories, and it is only required here because of dependent types and that meta-substitutions trigger regular substitutions.[16]

5.3.5 Weak-Head Reductions

I follow Abel et al. (2018) to establish the proofs of weak normalization and the decidability of convertibility, so I need a description of weak-head normal forms (WHNFs) and a notion of weak-head reductions. Their definitions are entirely standard. WHNFs for types (W) are either type constructors, or neutral types (V), which are meta-variables (u^δ) or a decoding of neutral terms ($\texttt{El}^l\ \nu$). WHNFs for terms (w) are either in introduction forms, or neutral terms (ν), which are either variables or elimination forms blocked by other neutrals. Weak normalization then proves that all well-formed types and terms must reach their WHNFs in finite steps of reductions. WHNFs and neutral forms are captured by the following grammar:

$$\begin{aligned} W :=\ & \texttt{Nat} \mid \Pi^{l,l'}(x : S).T \mid \texttt{Ty}_l \mid \overrightarrow{\ell} \Rightarrow^l T && \text{(WHNFs for types)} \\ & \mid (g : \mathsf{Ctx}) \Rightarrow^l T \mid (U : (\Gamma \vdash_{\text{D}} \mathbin{@} l)) \Rightarrow^{l'} T \mid \square(\Gamma \vdash_{\text{C}} \mathbin{@} l) \mid \square(\Gamma \vdash_{\text{C}} T \mathbin{@} l) \end{aligned}$$

[16]Readers may find concrete proof steps of Lemma 5.15 and a more detailed discussion in Remark after it in the technical report (Hu and Pientka, 2024a).

$$
\begin{array}{rcll}
V & := & U^\delta \mid \texttt{El}^l\ \nu & \text{(Neutral forms for types)} \\
w & := & \nu \mid \texttt{Nat} \mid \Pi^{l,l'}(x : s).t \mid \mathsf{Ty}_l \mid \mathsf{zero} \mid \mathsf{succ}\ t \mid \lambda^{l,l'}(x : S).t & \text{(WHNFs for terms)} \\
 & & \mid \Lambda^l\ \overrightarrow{\ell}.t \mid \Lambda^l\ g.t \mid \Lambda^{l,l'}_{\text{D}}\ U.t \mid \mathtt{box}\ T \mid \mathtt{box}\ t & \\
\nu & := & x \mid u^\delta \mid (\nu : \Pi^{l,l'}(x : S).T)\ s \mid \nu\ \$\ \overrightarrow{l} & \text{(Neutral forms for terms)} \\
 & & \mid \nu\ \$\ \Gamma \mid \nu\ \$_{\text{D}}\ T \mid \mathtt{letbox}^{l'}_{x_T.M}\ U \leftarrow (\nu : \square(\Gamma \vdash_{\text{C}} @\, l))\ \mathsf{in}\ t' & \\
 & & \mid \mathtt{letbox}^{l'}_{x_t.M}\ u \leftarrow (\nu : \square(\Gamma \vdash_{\text{C}} T\, @\, l))\ \mathsf{in}\ t' & \\
 & & \mid \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\nu : \square(\Gamma \vdash_{\text{C}} @\, l)) \mid \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ ((\mathtt{box}\ U^\delta) : \square(\Gamma \vdash_{\text{C}} @\, l)) & \\
 & & \mid \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\nu : \square(\Gamma \vdash_{\text{C}} T\, @\, l)) \mid \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ ((\mathtt{box}\ u^\delta) : \square(\Gamma \vdash_{\text{C}} T\, @\, l)) &
\end{array}
$$

The untyped one-step reductions for types ($T \rightsquigarrow T'$) and terms ($t \rightsquigarrow t'$) are also standard. In general, reductions are divided into two groups, one for reductions in head positions, and the other for actual computation. For types, head reductions only occur for `El`:

$$
\texttt{El}^0\ \texttt{Nat} \rightsquigarrow \texttt{Nat} \qquad \texttt{El}^{1+l}\ \texttt{Ty}_l \rightsquigarrow \texttt{Ty}_l \qquad \texttt{El}^{l \sqcup l'}\ \Pi^{l,l'}(x : s).t \rightsquigarrow \Pi^{l,l'}(x : \texttt{El}^l\ s).\texttt{El}^{l'}\ t
$$

$$
\frac{t \rightsquigarrow t'}{\texttt{El}^l\ t \rightsquigarrow \texttt{El}^l\ t'}
$$

Reductions for terms also follow the same principle. They follow the same line as in Sec. 4.5. One-step reductions enjoy typical properties:

Lemma 5.7 (Soundness). *Given $i \in \{\text{D}, \text{M}\}$,*

- *if $L \mid \Psi; \Gamma \vdash_i T @ l$ and $T \rightsquigarrow T'$, then $L \mid \Psi; \Gamma \vdash_i T \approx T' @ l$;*
- *if $L \mid \Psi; \Gamma \vdash_i t : T @ l$ and $t \rightsquigarrow t'$, then $L \mid \Psi; \Gamma \vdash_i t \approx t' : T @ l$.*

Lemma 5.8 (Preservation). *Given $i \in \{\text{D}, \text{M}\}$,*

- *if $L \mid \Psi; \Gamma \vdash_i T @ l$ and $T \rightsquigarrow T'$, then $L \mid \Psi; \Gamma \vdash_i T' @ l$;*
- *if $L \mid \Psi; \Gamma \vdash_i t : T @ l$ and $t \rightsquigarrow t'$, then $L \mid \Psi; \Gamma \vdash_i t' : T @ l$.*

Lemma 5.9 (Universe Substitutions). *Given $i \in \{\text{D}, \text{M}\}$,*

- *if $L' \mid \Psi; \Gamma \vdash_i T @ l$, $T \rightsquigarrow T'$ and $L \vdash \phi : L'$, then $T[\phi] \rightsquigarrow T'[\phi]$;*

- *if* $L' \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l$, $t \rightsquigarrow t'$ *and* $L \vdash \phi : L'$, *then* $t[\phi] \rightsquigarrow t'[\phi]$.

Lemma 5.10 (Regular Substitutions). *Given* $i \in \{\textsc{d}, \textsc{m}\}$,

- *if* $L \mid \Psi; \Delta \vdash_i T \mathbin{@} l$, $T \rightsquigarrow T'$ *and* $L \mid \Psi; \Gamma \vdash_i \delta : \Delta$, *then* $T[\delta] \rightsquigarrow T'[\delta]$;

- *if* $L \mid \Psi; \Delta \vdash_i t : T \mathbin{@} l$, $t \rightsquigarrow t'$ *and* $L \mid \Psi; \Gamma \vdash_i \delta : \Delta$, *then* $t[\delta] \rightsquigarrow t'[\delta]$.

Lemma 5.11 (Meta-Substitutions). *Given* $i \in \{\textsc{d}, \textsc{m}\}$,

- *if* $L \mid \Phi; \Gamma \vdash_i T \mathbin{@} l$, $T \rightsquigarrow T'$ *and* $L \mid \Psi \vdash \sigma : \Phi$, *then* $T[\sigma] \rightsquigarrow T'[\sigma]$;

- *if* $L \mid \Phi; \Gamma \vdash_i t : T \mathbin{@} l$, $t \rightsquigarrow t'$ *and* $L \mid \Psi \vdash \sigma : \Phi$, *then* $t[\sigma] \rightsquigarrow t'[\sigma]$.

Lemma 5.12 (Determinacy).

- *If* $T \rightsquigarrow T'$ *and* $T \rightsquigarrow T'$, *then* $T' = T''$.

- *If* $t \rightsquigarrow t'$ *and* $t \rightsquigarrow t''$, *then* $t' = t''$.

One-step reductions are generalized to multi-step reductions for types ($T \rightsquigarrow^* T'$) and for terms ($t \rightsquigarrow^* t'$). If multi-step reductions step to WHNFs, then determinacy ensures that WHNFs are unique.

Lemma 5.13 (Determinacy).

- *If* $T \rightsquigarrow^* W$ *and* $T \rightsquigarrow^* W'$, *then* $W = W'$.

- *If* $t \rightsquigarrow^* w$ *and* $t \rightsquigarrow^* w'$, *then* $w = w'$.

Typed reductions $L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow T' \mathbin{@} l$, $L \mid \Psi; \Gamma \vdash_i t \rightsquigarrow t' : T \mathbin{@} l$ and their multi-step variants are obtained by pairing untyped reductions with corresponding typing judgments.

$$\frac{L \mid \Psi; \Gamma \vdash_i T \mathbin{@} l \qquad T \rightsquigarrow T'}{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow T' \mathbin{@} l} \qquad \frac{L \mid \Psi; \Gamma \vdash_i T \mathbin{@} l \qquad T \rightsquigarrow^* T'}{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* T' \mathbin{@} l}$$

$$\frac{L \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l \qquad t \rightsquigarrow t'}{L \mid \Psi; \Gamma \vdash_i t \rightsquigarrow t' : T \mathbin{@} l} \qquad \frac{L \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l \qquad t \rightsquigarrow^* t'}{L \mid \Psi; \Gamma \vdash_i t \rightsquigarrow^* t' : T \mathbin{@} l}$$

5.3.6 Recursion on Code

Finally, I describe the rules for the recursors for code ($\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (t : \Box E)$). I have listed the syntax towards the end of Fig. 5.3. As mentioned before, this term encompasses in fact two mutual recursion principles: one for code of types and the other for code of terms. Therefore, there are two motives M_{Typ} and M_{Trm} in $\overrightarrow{M}$. They describe the return types of the recursion principles on code of types and of terms respectively. The branches also fall into two categories: the branches for recursively analyzing types (b_{Typ}) and for terms (b_{Trm}).

As the recursor describes two recursion principles, they give rise to two typing rules. I focus on the rule for the recursor for code of terms. The recursor for code of types is slightly simpler in the scrutinee. I abbreviate the well-formedness of branches as $\overrightarrow{b}$ wf and concentrate on checking well-formedness of motives M_{Typ} and M_{Trm} and the scrutinee t.

$$\frac{\begin{array}{c} L, \ell \mid \Psi, g : \mathsf{Ctx}; \Gamma, x_T : \Box(g \vdash_{\mathrm{C}} @\, \ell) @\, 0 \vdash_{\mathrm{M}} M_{\mathsf{Typ}} @\, l_1 \\ L, \ell \mid \Psi, g : \mathsf{Ctx}, u_T : (g \vdash_{\mathrm{D}} @\, \ell); \Gamma, x_t : \Box(g \vdash_{\mathrm{C}} u_T @\, \ell) @\, 0 \vdash_{\mathrm{M}} M_{\mathsf{Trm}} @\, l_2 \\ \overrightarrow{b}\ \mathsf{wf} \qquad L \mid \Psi; \Gamma \vdash_{\mathrm{M}} t : \Box(\Delta \vdash_{\mathrm{C}} T @\, l) @\, 0 \qquad L \mid \Psi; \Delta \vdash_{\mathrm{D}} T @\, l \end{array}}{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (t : \Box(\Delta \vdash_{\mathrm{C}} T @\, l)) : M_{\mathsf{Trm}}[l/\ell, \Delta/g, T/u_T, t/x_t] @\, l_2}$$

The motives M_{Typ} and M_{Trm} abstract over the context variable g, which might change during recursion. As the return type might also depend on the scrutinee, the motives also abstract over x_T (the scrutinee as code of type) or x_t (the scrutinee as code of term). In the latter case, I also keep track of the type of the scrutinee, denoted by u_T. The overall type of the recursion is $M_{\mathsf{Trm}}[l/\ell, \Delta/g, T/u_T, t/x_t]$ where I replace g with the concrete regular context Δ of the scrutinee, u_T with the type T of the scrutinee, and x_t with the scrutinee t itself in M_{Trm}.

Note that T lives at layer D, which supports computation, while the term eventually computed by t will be a static contextual code object describing the syntax of an MLTT term. As a consequence, if T is for example $(\lambda x.x)$ `Nat`, it is equivalent to `Nat`. In other words, it captures the fact that we are only interested in analyzing a code object of type `Nat` and the exact shape of the type T is unimportant. On the other hand, if the code object t contains $(\lambda x.x)$ `Nat` as a sub-code, the redex would remain, because this code object represents a different static syntax tree from that of code `Nat` due to the static code lemma.

Since contextual types also compute, to make sure the determinacy of one-step reduction, I choose to always reduce this T before reducing the scrutinee. Other strategies could have been taken; but this particular one is the simplest to ensure determinacy.

$$\frac{T \rightsquigarrow T'}{\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (t : \square(\Delta \vdash_{\mathsf{c}} T \mathbin{@} l)) \rightsquigarrow \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (t : \square(\Delta \vdash_{\mathsf{c}} T' \mathbin{@} l))}$$

$$\frac{t \rightsquigarrow t'}{\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (t : \square(\Delta \vdash_{\mathsf{c}} W \mathbin{@} l)) \rightsquigarrow \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (t' : \square(\Delta \vdash_{\mathsf{c}} W \mathbin{@} l))}$$

Note that t is only reduced after the contextual type has reached a WHNF W.

Now let us consider the well-formedness of branches. As described earlier in Sec. 5.2.4, I distinguish branches for types (b_{Typ}) and terms (b_{Trm}). I use the branch for function applications t_{app} as a running example and other branches can be derived naturally following the same principle (see all the branches in Appendix F). To facilitate the discussion, I use colors to differentiate contexts, universe variables, types and terms in the pattern and the scrutinee. For more readability, I simply write u for u^{id} when a meta-variable is associated with the identity substitution. In this branch, the scrutinee is the code of $(t : \Pi^{l,l'}(x : S).T)\ s$ and is matched against the pattern $(u_t : \Pi^{\ell,\ell'}(x : u_S).u_T)\ u_s$. Each sub-structure in the scrutinee is matched by a pattern variable. These pattern variables are meta-code variables u extended to the meta-context Ψ and with matching subscripts of the sub-structures. The well-formedness conditions of the pattern variables record the ambient contexts, and the types if the pattern variables are for code of terms. When lining up the pattern variables and the sub-structures, there is a correspondence not only between them, but also between their well-formedness conditions. For example, the typing of the pattern variable u_s encodes the well-formedness of its matching sub-structure s.

	Regular Context	Code Object	Well-formedness
Sub-structures	Γ	s	$L \mid \Psi; \Gamma \vdash_{\mathsf{c}} s : S \mathbin{@} l$
Pattern variables	g	u_s	$u_s : (g \vdash_{\mathsf{c}} u_S \mathbin{@} \ell)$

In addition to the pattern variables, there are two new universe variables ℓ and ℓ' to capture the universes of S and T. Finally, each meta-code variable gives rise to a recursive call. The recursive calls are regular variables x extended to the regular context with matching subscripts of the sub-structures. The recursive calls on code of MLTT types such as u_S and u_T have the corresponding type M_{Typ} appropriately refined. In particular, the scrutinee x_T in M_{Typ} is instantiated with $\texttt{box}\ u_S$ and $\texttt{box}\ u_T$, respectively. Since u_T has a longer regular context than u_S, this fact is reflected in the variable g in M_{Typ}. Recursive calls on code of terms such as u_t and u_s have the corresponding type M_{Trm} appropriately instantiated. Here I replace the scrutinee x_t in M_{Trm} with $\texttt{box}\ u_s$ and $\texttt{box}\ u_t$, respectively. In addition to the regular context for each of u_t and u_s, I also refine the type u_T in M_{Trm} with a Π type and u_S, respectively. Collecting all additional assumptions in the contexts gives rise to the following well-formedness condition for the branch t_{app} for function applications:

$$\begin{array}{ll}
L, \ell, \ell' \mid \Psi, g : \mathsf{Ctx} & \\
\quad , u_S : (g \vdash_{\mathrm{C}} \ @\ \ell),\ \ u_T : (g, x : u_S\ @\ \ell \vdash_{\mathrm{C}} \ @\ \ell') & \\
\quad , u_t : (g \vdash_{\mathrm{C}} \Pi^{\ell,\ell'}(x : u_S).u_T\ @\ \ell \sqcup \ell'),\ \ u_s : (g \vdash_{\mathrm{C}} u_S\ @\ \ell) & \left.\vphantom{\begin{array}{l}a\\a\\a\end{array}}\right\}\text{Pattern variables} \\
\quad ; \Gamma, x_S : M_{\mathsf{Typ}}[\ell/\ell, g/g, \texttt{box}\ u_S/x_T]\ @\ l_1 & \\
\quad , x_T : M_{\mathsf{Typ}}[\ell'/\ell, (g, x : u_S\ @\ \ell)/g, \texttt{box}\ u_T/x_T]\ @\ l_1 & \\
\quad , x_t : M_{\mathsf{Trm}}[\ell \sqcup \ell'/\ell, g/g, \Pi^{\ell,\ell'}(x : u_S).u_T/u_T, \texttt{box}\ u_t/x_t]\ @\ l_2 & \\
\quad , x_s : M_{\mathsf{Trm}}[\ell/\ell, g/g, u_S/u_T, \texttt{box}\ u_s/x_t]\ @\ l_2 & \left.\vphantom{\begin{array}{l}a\\a\\a\\a\end{array}}\right\}\text{Recursive Calls} \\
\vdash_{\mathrm{M}} t_{\mathsf{app}} : M_{\mathsf{Trm}}[\ell'/\ell, g/g, {u_T}^{\mathsf{id}, u_s/x}/u_T, \texttt{box}\ ((u_t : \Pi^{\ell,\ell'}(x : u_S).u_T)\ u_s)/x_t]\ @\ l_2 &
\end{array}$$

Since recursions only occur for syntactic sub-terms, it informally justifies the recursors and their termination. This informal observation is made precise in the semantics of code given in Sec. 5.4.4. The reduction rule is straightforward. I replace each component with the appropriate instantiation. Here I focus on the rule for one-step reductions to save space; the equivalence rule is naturally derived from the reduction rule.

$$\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\texttt{box}\ ((t : \Pi^{l,l'}(x : S).T)\ s) : \square(\Gamma \vdash_{\mathrm{C}} W\ @\ l)) \rightsquigarrow t_{\mathsf{app}}[l/\ell, l'/\ell', \sigma, \delta]$$

where $\sigma = \Gamma/g, S/u_S, T/u_T, t/u_t, s/u_s$ is the meta-substitution which instantiates all pattern variables, and δ builds all recursive calls. It is defined as:

$$\begin{array}{ll} \delta = \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ S : \Box(\Gamma \vdash_{\text{C}}\ @\, l)) & /x_S \\ \quad, \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ T : \Box(\Gamma, x : S\,@\,l \vdash_{\text{C}}\ @\, l')) & /x_T \\ \quad, \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ t : \Box(\Gamma \vdash_{\text{C}} \Pi^{l,l'}(x : S).T\,@\,l \sqcup l')) & /x_t \\ \quad, \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ s : \Box(\Gamma \vdash_{\text{C}} S\,@\,l)) & /x_s \end{array}$$

To end the discussion on syntax, I recapitulate the difference between `letbox` and elim. Based on Hu and Pientka (2024b) and Chapter 4, the former is responsible for code composition and running, while the latter is for intensional analysis. They differ specifically in their computational behavior. In particular, $\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ u^{\delta} : \Box E)$ is neutral, because u^{δ} simply does not find a branch in $\overrightarrow{b}$. Meanwhile, $\mathtt{letbox}^{l}_{x.M}\ u' \leftarrow (\mathtt{box}\ u^{\delta} : \Box E)\ \mathtt{in}\ t$ reduces to $t[u^{\delta}/u']$. This distinction is further revealed in the logical relations, where semantics for terms at layer C must carry two kinds of information: syntactic information about their shapes and semantic information about how they run (c.f. Sec. 5.4.4). In the next section, I will describe how to model features in DeLaM semantically.

5.4 Kripke Logical Relations

The Kripke logical relations of DeLaM follow the same outline as in Abel et al. (2018) and are extensions of those in Chapter 4: the logical relations are parameterized by generic equivalences. To derive the fundamental theorems, I instantiate these generic equivalences. However, DeLaM is quite complex due to the presence of dependent types, Tarski-style universes, and the distinction between MLTT (C and D layers) and meta-programming (M layer) which enables quoting, evaluating, and recursively analyzing code. To shorten the proofs as much as possible, I simplify the logical relations by adopting PER-style definitions. In this way, I can define the logical relations with only two predicates – in contrast, (Abel et al., 2018) requires four predicates for types and terms. This improvement further allows more compact statements of lemmas and proofs by reducing their number to approximately half. This significantly eases the meta-theoretic development. Nevertheless, the complete proofs remain very verbose and therefore, in this thesis, I only focus on the main idea and refer the interested

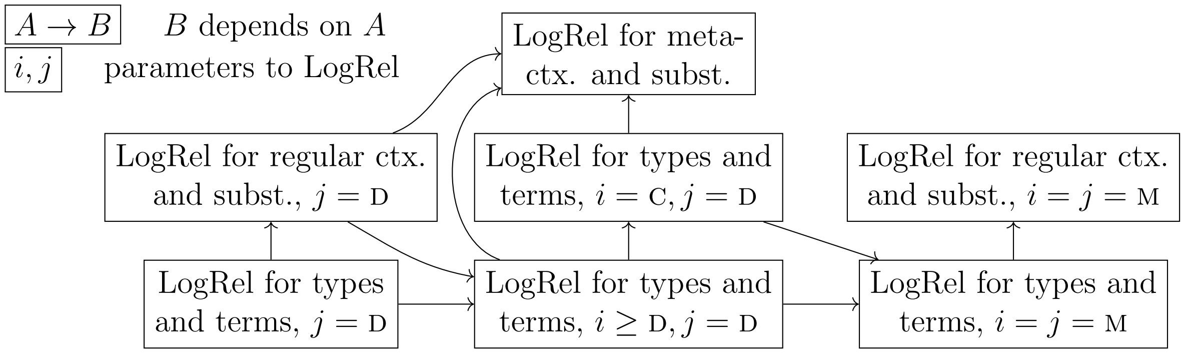

$A \to B$ B depends on A
i, j parameters to LogRel
LogRel for meta-ctx. and subst.
LogRel for regular ctx. and subst., $j = \textsc{d}$
LogRel for types and terms, $i = \textsc{c}, j = \textsc{d}$
LogRel for regular ctx. and subst., $i = j = \textsc{m}$
LogRel for types and terms, $j = \textsc{d}$
LogRel for types and terms, $i \geq \textsc{d}, j = \textsc{d}$
LogRel for types and terms, $i = j = \textsc{m}$

Figure 5.5: Structure of logical relations

readers to the technical report (Hu and Pientka, 2024a). The structure of the logical relations is depicted in Fig. 5.5, where the nodes are clickable in a PDF viewer. The logical relations are parameterized by i, the layer for terms, and j, the layer for types. I will give more explanations on i and j in Sec. 5.4.2.

5.4.1 Generic Equivalences

First, I define the generic equivalences for types and terms. The generic equivalences are parameters of judgments with laws to the logical relations, which I eventually instantiate with the conversion checking algorithm to obtain its completeness proof. They follow the same pattern as in Sec. 4.6. There are four generic equivalence judgments: $\boxed{L \mid \Psi; \Gamma \vdash_i V \sim V' \mathbin{@} l}$ and $\boxed{L \mid \Psi; \Gamma \vdash_i T \simeq T' \mathbin{@} l}$ are equivalences for neutral types and any types, and $\boxed{L \mid \Psi; \Gamma \vdash_i \nu \sim \nu' : T \mathbin{@} l}$ and $\boxed{L \mid \Psi; \Gamma \vdash_i t \simeq t' : T \mathbin{@} l}$ are those for neutral terms and any terms. The layer i only takes $\textsc{d}$ or $\textsc{m}$ because they are the only layers with computation. Their laws are formulated following closely the recipe by Abel et al. (2018) and are similar to Sec. 4.6. These judgments are subsumed by syntactic equivalence at their layers and form PERs. They respect equivalence of contexts and types, and weakenings of contexts. The rest of the laws are to capture how equivalences are propagated under WHNFs. For example, for t and t' of type $\Pi^{l,l'}(x : S).T$ to be equivalent, a law requires $t\ x$ and $t'\ x$ to be equivalent on type T for $x : S$. Two neutrals are generically equivalent if all sub-components are generically equivalent pointwise.[17]

[17]A complete list of laws is available in the technical report (Hu and Pientka, 2024a, Sec. 7.1).

5.4.2 Logical Relations for Types and Terms in MLTT

The logical relations are defined on top of generic equivalence. They are Kripke in their stability under weakenings. Since there are three different contexts, there are three corresponding kinds of weakenings in DeLaM. In particular, $\theta :: L \Longrightarrow L'$ is a universe weakening, $\gamma :: L \mid \Psi \Longrightarrow_g \Phi$ is a meta-weakening, and $\tau :: L \mid \Psi; \Gamma \Longrightarrow_i \Delta$ is a regular weakening. The subscript i denotes the layer at which the regular contexts Γ and Δ live. Combining different weakenings is to simultaneously apply them at the same time. Weakenings are applied in the order of θ, γ and at last τ. I let ψ range over simultaneous weakenings of all three contexts:

$$\psi := (\theta, \gamma, \tau) :: L \mid \Psi; \Gamma \Longrightarrow_i L' \mid \Phi; \Delta$$

where

$$\begin{aligned} &\theta :: L \Longrightarrow L' \\ &\gamma :: L' \mid \Psi[\theta] \Longrightarrow_g \Phi \\ &\tau :: L' \mid \theta; \Gamma[\theta][\gamma] \Longrightarrow_i \Delta \end{aligned}$$

Similarly, there are times when I need to only weaken the universe and meta-contexts. At this time, I use α to range over these weakenings:

$$\alpha := (\theta, \gamma) :: L \mid \Psi \Longrightarrow L' \mid \Phi$$

Moreover, following Chapter 4, the actions of weakenings are implicit in DeLaM, to avoid clutters. That is, if T is well-formed in $L' \mid \Phi; \Delta$, I directly say that T is weakened and is also well-typed in $L \mid \Psi; \Gamma$.

As the first step, I define the logical relations for types and terms: $\boxed{\mathcal{D} :: L \mid \Psi; \Gamma \Vdash^j_i T \approx T' \mathbin{@} l}$ and $\boxed{L \mid \Psi; \Gamma \Vdash^j_i t \approx t' : \mathtt{El}(\mathcal{D})}$, where $(i, j) \in \{(\textsc{d}, \textsc{d}), (\textsc{m}, \textsc{d}), (\textsc{m}, \textsc{m})\}$ or equivalently $i \geq j \geq \textsc{d}$. As I have set up in Sec. 1.4, I assume an informal meta-type theory across the whole thesis, so I use $\mathcal{D}$ to refer to the judgment $L \mid \Psi; \Gamma \Vdash^j_i T \approx T' \mathbin{@} l$ in an inductive-recursive definition. The judgments say that T and T' are related (resp. t and t') as types at layer j and as terms at layer i. When $j = \textsc{d}$, it means that T and T' are related types in MLTT after reductions, regardless of which layer they and their terms actually live at. This separation of con-

$$\frac{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* \texttt{Nat} \mathbin{@} 0 \quad L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* \texttt{Nat} \mathbin{@} 0}{L \mid \Psi; \Gamma \Vdash\!\vDash^j_i T \approx T' \mathbin{@} 0}$$

$$\frac{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* \texttt{Ty}_l \mathbin{@} 1+l \quad L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* \texttt{Ty}_l \mathbin{@} 1+l}{L \mid \Psi; \Gamma \Vdash\!\vDash^j_i T \approx T' \mathbin{@} 1+l}$$

$$\frac{L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* V \mathbin{@} l \quad L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* V' \mathbin{@} l \quad L \mid \Psi; \Gamma \vdash_i V \sim V' \mathbin{@} l}{L \mid \Psi; \Gamma \Vdash\!\vDash^j_i T \approx T' \mathbin{@} l}$$

$$\frac{\begin{array}{l} L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* \Pi^{l,l'}(x : S_1).T_1 \mathbin{@} l \sqcup l' \qquad L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* \Pi^{l,l'}(x : S_2).T_2 \mathbin{@} l \sqcup l' \\ \mathcal{E} :: \forall\ \psi :: L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma\ .\ L' \mid \Phi; \Delta \Vdash\!\vDash^j_i S_1 \approx S_2 \mathbin{@} l \\ \mathcal{F} :: \forall\ \psi :: L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma \text{ and } L' \mid \Phi; \Delta \Vdash\!\vDash^j_i s \approx s' : \texttt{El}(\mathcal{E}(\psi))\ . \\ \qquad L' \mid \Phi; \Delta \Vdash\!\vDash^j_i T_1[s/x] \approx T_2[s'/x] \mathbin{@} l' \end{array}}{L \mid \Psi; \Gamma \Vdash\!\vDash^j_i T \approx T' \mathbin{@} l \sqcup l'}$$

Figure 5.6: Logical relations for types in MLTT

sideration in j is particularly important in the layering restriction lemma (Lemma 5.14), which gives a semantic explanation for code execution. As a mnemonic, the number of vertical bars in the turnstile matches the number of contexts. As the thesis progresses, this number gradually decreases, so it also serves as a progress bar for the technical development.

The logical relations are defined by

1. first, recursion on j, which means that I first define the semantics for types of MLTT before I consider all types at layer $\textsc{m}$,
2. then a transfinite well-founded recursion on the universe levels,
3. at last, an induction-recursion (Dybjer and Setzer, 2003) to relate types and terms.

The recursion on j before universe levels allows to restart the universe level from 0 when referring to the logical relations for $j = \textsc{d}$ in those for $j = \textsc{m}$. Therefore, all contextual types can safely live on level 0. I will discuss more when I define the logical relations for $j = \textsc{m}$ in Sec. 5.4.6.

The logical relations for types are defined inductively in Fig. 5.6. The inductive definition is named $\mathcal{D}$, which is used in the recursion to relate terms. The four cases are natural numbers, universes, neutral types and Π types. The last case is the most complex one. First, T and T' reduce to their respective Π types. Then the third premise, named $\mathcal{E}$, relates input types for all weakenings, and the fourth premise $\mathcal{F}$ relates output types for all weakenings given related inputs s and s' given by $\mathcal{E}(\psi)$. The predicate $\mathcal{E}(\psi)$ effectively gives $L' \mid \Phi; \Delta \Vdash^{j}_{i} S_1 \approx S_2 \mathbin{@} l$, which is used by the recursive predicate to relate s and s'. This is exactly the place that requires an inductive-recursive definition. The logical relations for terms $L \mid \Psi; \Gamma \Vdash^{j}_{i} t \approx t' : \texttt{El}(\mathcal{D})$ are defined by recursion on $\mathcal{D}$. In the case of Π types, the logical relations for terms are defined as

- $L \mid \Psi; \Gamma \vdash_i t \leadsto^* w : \Pi^{l,l'}(x : S_1).T_1 \mathbin{@} l \sqcup l'$ and $L \mid \Psi; \Gamma \vdash_i t' \leadsto^* w' : \Pi^{l,l'}(x : S_1).T_1 \mathbin{@} l \sqcup l'$, i.e. t and t' reduce to WHNFs w and w', respectively;
- $L \mid \Psi; \Gamma \vdash_i w \simeq w' : \Pi^{l,l'}(x : S_1).T_1 \mathbin{@} l \sqcup l'$, saying that w and w' are generically equivalent,
- at last, for all $\psi :: L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma$ and related inputs $\mathcal{A} :: L' \mid \Phi; \Delta \Vdash^{j}_{i} s \approx s' : \texttt{El}(\mathcal{E}(\psi))$, the results of applying w and w' are related: $L' \mid \Phi; \Delta \Vdash^{j}_{i} w\ s \approx w'\ s' : \texttt{El}(\mathcal{F}(\psi, \mathcal{A}))$.

When $j = \textsc{d}$, I have given the complete definition of the logical relations for MLTT. Compared to the version by Abel et al. (2018), the number of definitions is reduced by half by defining the logical relations in a PER style. I still have to prove the same set of lemmas, but their statements and proofs are also halved in size. This makes the large semantic development of DeLaM more manageable.

When $j = \textsc{m}$, there still need more cases to handle types for meta-programming and universe-polymorphic functions (see Sec. 5.4.6). Nevertheless, the definitions given here still apply. In other words, for types shared between layers $\textsc{d}$ and $\textsc{m}$, their logical relations only differ in layers. This observation is captured by the layering restriction lemma:

Lemma 5.14 (Layering Restriction). *If $\mathcal{D} :: L \mid \Psi; \Gamma \Vdash^{\textsc{d}}_{\textsc{m}} T \approx T' \mathbin{@} l$,*

- *then $\mathcal{E} :: L \mid \Psi; \Gamma \Vdash^{\textsc{m}}_{\textsc{m}} T \approx T' \mathbin{@} l$;*
- *then $L \mid \Psi; \Gamma \Vdash^{\textsc{d}}_{\textsc{m}} t \approx t' : \texttt{El}(\mathcal{D})$ and $L \mid \Psi; \Gamma \Vdash^{\textsc{m}}_{\textsc{m}} t \approx t' : \texttt{El}(\mathcal{E})$ are equivalent statements.*

$$\frac{L \vdash \Psi}{L \mid \Psi \Vdash_i^j \cdot \approx \cdot} \qquad \frac{L \vdash \Psi \qquad u : \mathsf{Ctx} \in \Psi}{L \mid \Psi \Vdash_i^j u \approx u}$$

$$\frac{\begin{array}{l}\mathcal{E} :: \forall\ \alpha :: L' \mid \Phi \Longrightarrow L \mid \Psi\ .\ L' \mid \Phi \Vdash_i^j \Delta \approx \Delta' \\ \mathcal{F} :: \forall\ \alpha :: L' \mid \Phi \Longrightarrow L \mid \Psi \text{ and } L' \mid \Phi; \Gamma \Vdash_i^j \delta \approx \delta' : \mathcal{E}(\alpha)\ . \\ \qquad L' \mid \Phi; \Gamma \Vdash_i^j T[\delta] \approx T'[\delta'] \mathbin{@} l\end{array}}{L \mid \Psi \Vdash_i^j \Delta, x : T \mathbin{@} l \approx \Delta', x : T' \mathbin{@} l}$$

Figure 5.7: Logical relations for contexts

The most interesting direction in this lemma is M to D, i.e. that $L \mid \Psi; \Gamma \Vdash_{\text{M}}^{\text{D}} t \approx t' : \texttt{El}(\mathcal{D})$ implies $L \mid \Psi; \Gamma \Vdash_{\text{M}}^{\text{M}} t \approx t' : \texttt{El}(\mathcal{E})$, given that T and T' are related at layer D, i.e. in MLTT. Effectively, if we know that T and T' eventually reduce to some types from MLTT, then we can *refine* the relation between t and t' from $j = \text{M}$ to $j = \text{D}$. This lemma is crucial to model code execution semantically. The similar situation occurs in simple types too as described at the beginning of Sec. 4.7 and is formalized in the meta-variable case in the proof of the fundamental theorems (Theorem 4.28). The same reasoning is applied to DeLaM as well. Consider an identity function $\lambda(x : \texttt{Nat}).x$ for natural numbers from layer C. The semantics of this function says that given a normalizing input t in MLTT, $(\lambda x.x)\ t$ gives a normalizing output also in MLTT. However, after being lifted to layer M, the identity function can be applied to meta-programs like $\texttt{letbox}\ u \leftarrow \texttt{box zero in}\ u$, which is clearly not in MLTT. Layering restriction is here to rescue by saying that even though meta-programs are not from MLTT, since Nat is a type in MLTT, the semantics of every normalizing meta-program of type Nat can be refined/lowered to $j = \text{D}$ to be passed as arguments to the identity function. Note that this action has nothing to do with syntax. It does *not* actually lower the layer of terms but is purely semantic and just refines the information of the logical relations. Finally, the results are lifted again to $j = \text{M}$ in the reverse direction. In this way, the machinery in code execution is semantically explained. Interestingly, though lifting (Lemma 5.1) monotonically brings terms from a lower layer to a higher one syntactically, layering restriction does need to be expressed as an equivalence in the semantics.

The logical relations for regular contexts ($\mathcal{D} :: L \mid \Psi \Vdash_i^j \Gamma \approx \Delta$) and regular substitutions
($L \mid \Psi; \Gamma \Vvdash_i^j \delta \approx \delta' : \mathcal{D}$) are also defined inductive-recursively. The former is defined inductively in Fig. 5.7 and requires all relations between types to be stable under regular substitutions pointwise. The latter is defined recursively on the former, and is a generalization of the logical relations of terms. They are also defined in the PER style so that subsequent proofs are simplified. For the case of context extensions, $L \mid \Psi; \Gamma \Vvdash_i^j \delta \approx \delta' : \mathcal{D}$ is defined as

- $\delta = \delta_1, t/x$ and $\delta' = \delta_1', t'/x$, so substitutions are also extended;
- $\mathcal{C} :: \forall\ \alpha :: L' \mid \Phi \Longrightarrow L \mid \Psi\ .\ L' \mid \Phi; \Gamma \Vvdash_i^j \delta_1 \approx \delta_1' : \mathcal{E}(\alpha)$, which says that δ_1 and δ_1' are recursively related by $\mathcal{E}(\alpha)$;
- finally, $\forall\ \alpha :: L' \mid \Phi \Longrightarrow L \mid \Psi\ .\ L' \mid \Phi; \Gamma \Vvdash_i^j t \approx t' : \mathtt{El}(\mathcal{F}(\alpha, \mathcal{C}(\alpha)))$, which relates t and t'. Note that t and t' are related by $\mathcal{F}$, which requires two arguments, a weakening α, and a relation between two regular substitutions, which is given by $\mathcal{C}$.

The logical relations for regular contexts and regular substitutions also have a generalized version of layering restriction (Lemma 5.14).

Lemma 5.15 (Layering Restriction). *If $\mathcal{D} :: L \mid \Psi \Vdash_{\mathrm{M}}^{\mathrm{D}} \Delta \approx \Delta'$,*

- *then $\mathcal{E} :: L \mid \Psi \Vdash_{\mathrm{M}}^{\mathrm{M}} \Delta \approx \Delta'$;*
- *then $L \mid \Psi; \Gamma \Vvdash_{\mathrm{M}}^{\mathrm{D}} \delta \approx \delta' : \mathcal{D}$ and $L \mid \Psi; \Gamma \Vvdash_{\mathrm{M}}^{\mathrm{M}} \delta \approx \delta' : \mathcal{E}$ are equivalent.*

Proof. Induction on $\mathcal{D}$. The step case is very similar to the function case above. □

5.4.3 Properties of Logical Relations

Besides layering restriction, logical relations also have the following properties. First, the weakening lemma says that the logical relations are indeed Kripke: they are closed under weakenings.

Lemma 5.16 (Weakening). *If $\mathcal{D} :: L \mid \Psi; \Gamma \Vvdash_i^{\mathrm{D}} T \approx T' \mathbin{@} l$,*

- *and if* $\psi :: L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma$*, then* $\mathcal{E} :: L' \mid \Phi; \Delta \Vdash^{\mathrm{D}}_i T \approx T' \,@\, l$*;*

- *and if* $L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i t \approx t' : \mathtt{El}(\mathcal{D})$ *and* $\psi :: L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma$*, then* $L' \mid \Phi; \Delta \Vdash^{\mathrm{D}}_i t \approx t' : \mathtt{El}(\mathcal{E})$*.*

The escape lemma says that the logical relations at layer i implies generic equivalences at layer i.

Lemma 5.17 (Escape)**.** *If* $\mathcal{D} :: L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i T \approx T' \,@\, l$*,*

- *then* $L \mid \Psi; \Gamma \vdash_i T \simeq T' \,@\, l$*;*

- *and if* $L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i t \approx t' : \mathtt{El}(\mathcal{D})$*, then* $L \mid \Psi; \Gamma \vdash_i t \simeq t' : T \,@\, l$ *and* $L \mid \Psi; \Gamma \vdash_i t \simeq t' : T' \,@\, l$*.*

In the conclusion, the fact of $j = \mathrm{D}$ is not used anywhere. In particular, if $i = \mathrm{M}$, T and T' are types from layer M as well. This is not a problem, because the logical relation only requires them to be types from MLTT *after being reduced to WHNFs.*

Next, the irrelevance lemmas say that the exact relation between types is not relevant to relate terms.

Lemma 5.18 (Right Irrelevance)**.** *If* $\mathcal{D} :: L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i T \approx T' \,@\, l$*,* $\mathcal{E} :: L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i T \approx T'' \,@\, l$ *and* $L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i t \approx t' : \mathtt{El}(\mathcal{D})$*, then* $L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i t \approx t' : \mathtt{El}(\mathcal{E})$*.*

Lemma 5.19 (Left Irrelevance)**.** *If* $\mathcal{D} :: L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i T' \approx T \,@\, l$*,* $\mathcal{E} :: L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i T'' \approx T \,@\, l$ *and* $L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i t \approx t' : \mathtt{El}(\mathcal{D})$*, then* $L \mid \Psi; \Gamma \Vdash^{\mathrm{D}}_i t \approx t' : \mathtt{El}(\mathcal{E})$*.*

Due to irrelevance, it is often convenient to discuss only the relation between terms without having to specify the relation between types. In this case, I define $\boxed{L \mid \Psi; \Gamma \Vdash^{j}_i t \approx t' : T \,@\, l}$ as an abbreviation for some T', such that there is $\mathcal{D} :: L \mid \Psi; \Gamma \Vdash^{j}_i T \approx T' \,@\, l$ and $L \mid \Psi; \Gamma \Vdash^{j}_i t \approx t' : \mathtt{El}(\mathcal{D})$.

Finally, symmetry and transitivity prove that the logical relations indeed form PERs.

Lemma 5.20 (Symmetry)**.**

- *If* $\mathcal{D} :: L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i T \approx T' \,@\, l$*, then* $\mathcal{E} :: L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i T' \approx T \,@\, l$.

- *If* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i t \approx t' : \mathit{El}(\mathcal{D})$*, then* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i t' \approx t : \mathit{El}(\mathcal{E})$.

Lemma 5.21 (Transitivity)**.**

- *If* $\mathcal{D}_1 :: L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i T_1 \approx T_2 \,@\, l$ *and* $\mathcal{D}_2 :: L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i T_2 \approx T_3 \,@\, l$*, then* $\mathcal{D}_3 :: L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i T_1 \approx T_3 \,@\, l$.

- *If* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i t_1 \approx t_2 : \mathit{El}(\mathcal{D}_1)$ *and* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i t_2 \approx t_3 : \mathit{El}(\mathcal{D}_2)$*, then* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i t_1 \approx t_3 : \mathit{El}(\mathcal{D}_3)$.

The logical relations for regular contexts and regular substitutions also have similar properties. In general, the lemmas are just generalizations of corresponding lemmas above.

Lemma 5.22 (Weakening)**.**

- *If* $\mathcal{D} :: L \mid \Psi \Vdash\!\vDash^{\mathrm{D}}_i \Delta \approx \Delta'$ *and* $\alpha :: L' \mid \Phi \Longrightarrow L \mid \Psi$*, then* $\mathcal{E} :: L' \mid \Phi \Vdash\!\vDash^{\mathrm{D}}_i \Delta \approx \Delta'$.

- *If* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i \delta \approx \delta' : \mathcal{D}$*,* $\alpha :: L' \mid \Phi \Longrightarrow L \mid \Psi$ *and* $\tau :: L' \mid \Phi; \Gamma' \Longrightarrow_i \Gamma$*, then* $L' \mid \Phi; \Gamma' \Vdash\!\!\vDash^{\mathrm{D}}_i \delta \approx \delta' : \mathcal{E}$.

Lemma 5.23 (Escape)**.**

- *If* $\mathcal{D} :: L \mid \Psi \Vdash\!\vDash^{\mathrm{D}}_i \Delta_1 \approx \Delta_2$*, then* $L \mid \Psi \vdash_i \Delta_1 \simeq \Delta_2$.

- *If* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i \delta \approx \delta' : \mathcal{D}$*, then* $L \mid \Psi; \Gamma \vdash_i \delta \simeq \delta' : \Delta_1$ *and* $L \mid \Psi; \Gamma \vdash_i \delta \simeq \delta' : \Delta_2$.

Lemma 5.24 (Right Irrelevance)**.** *If* $\mathcal{D} :: L \mid \Psi \Vdash\!\vDash^{\mathrm{D}}_i \Delta_1 \approx \Delta_2$*,* $\mathcal{E} :: L \mid \Psi \Vdash\!\vDash^{\mathrm{D}}_i \Delta_1 \approx \Delta_3$ *and* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i \delta \approx \delta' : \mathcal{D}$*, then* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i \delta \approx \delta' : \mathcal{E}$.

Lemma 5.25 (Left Irrelevance)**.** *If* $\mathcal{D} :: L \mid \Psi \Vdash\!\vDash^{\mathrm{D}}_i \Delta_1 \approx \Delta_2$*,* $\mathcal{E} :: L \mid \Psi \Vdash\!\vDash^{\mathrm{D}}_i \Delta_3 \approx \Delta_2$ *and* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i \delta \approx \delta' : \mathcal{D}$*, then* $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{\mathrm{D}}_i \delta \approx \delta' : \mathcal{E}$.

Similarly, due to irrelevance, I define $\boxed{L \mid \Psi; \Gamma \Vdash\!\!\vDash^{j}_i \delta \approx \delta' : \Delta}$ as for some Δ', $\mathcal{E} :: L \mid \Psi \Vdash\!\vDash^{j}_i \Delta \approx \Delta'$ and $L \mid \Psi; \Gamma \Vdash\!\!\vDash^{j}_i \delta \approx \delta' : \mathcal{E}$.

Lemma 5.26 (Symmetry)**.**

- *If* $\mathcal{D} :: L \mid \Psi \Vdash^{\text{D}}_i \Delta_1 \approx \Delta_2$*, then* $\mathcal{E} :: L \mid \Psi \Vdash^{\text{D}}_i \Delta_2 \approx \Delta_1$*.*
- *If* $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_i \delta \approx \delta' : \mathcal{D}$*, then* $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_i \delta' \approx \delta : \mathcal{E}$*.*

Lemma 5.27 (Transitivity)**.**

- *If* $\mathcal{D}_1 :: L \mid \Psi \Vdash^{\text{D}}_i \Delta_1 \approx \Delta_2$ *and* $\mathcal{D}_2 :: L \mid \Psi \Vdash^{\text{D}}_i \Delta_2 \approx \Delta_3$*, then* $\mathcal{D}_3 :: L \mid \Psi \Vdash^{\text{D}}_i \Delta_1 \approx \Delta_3$*.*
- *If* $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_i \delta_1 \approx \delta_2 : \mathcal{D}_1$ *and* $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_i \delta_2 \approx \delta_3 : \mathcal{D}_2$*, then* $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_i \delta_1 \approx \delta_3 : \mathcal{D}_3$*.*

5.4.4 Semantics for MLTT and Code

At the end of Sec. 5.3.6, I mentioned that terms at layer C need to maintain both semantic information and syntactic information. In Chapter 4 and also Hu and Pientka (2024b), I achieve this by embedding the semantic information in an inductively defined judgment which stores syntactic information. This is exactly how I proceed here as well. First, I define semantics for types, terms and regular substitutions that are stable under regular substitutions. Effectively, these definitions give the semantics for pure MLTT. I always set $j = \text{D}$ because I am only concerned about types from MLTT for now. I define $\boxed{L \mid \Psi; \Gamma \Vvdash^{\text{D}}_{\geq \text{D}} T \approx T' \mathbin{@} l}$ as for all $L' \mid \Phi \Longrightarrow L \mid \Psi$, $k \geq \text{D}$, and $L' \mid \Phi; \Delta \Vvdash^{\text{D}}_k \delta \approx \delta' : \Gamma$, it holds that $L' \mid \Phi; \Delta \Vvdash^{\text{D}}_k T[\delta] \approx T'[\delta'] \mathbin{@} l$. The condition $k \geq \text{D}$ means that T and T' are related at both layers D and M. The judgment is $\boxed{L \mid \Psi; \Gamma \Vvdash^{\text{D}}_{\geq \text{D}} t \approx t' : T \mathbin{@} l}$ defined similarly, but instead I require the conclusion to be $L' \mid \Phi; \Delta \Vvdash^{\text{D}}_k t[\delta] \approx t'[\delta'] : T[\delta] \mathbin{@} l$. The generalizations $\boxed{L \mid \Psi \Vdash^{\text{D}}_{\geq \text{D}} \Delta \approx \Delta'}$ and $\boxed{L \mid \Psi; \Gamma \Vvdash^{\text{D}}_{\geq \text{D}} \delta \approx \delta' : \Delta}$ are defined in a similar manner. These semantic judgments capture the running information of these objects. For convenience, I also define asymmetric variants by requiring both sides to be equal, e.g. $\boxed{L \mid \Psi; \Gamma \Vvdash^{\text{D}}_{\geq \text{D}} T \mathbin{@} l}$ is defined as $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_{\geq \text{D}} T \approx T \mathbin{@} l$. Given that these judgments characterize pure MLTT, they are closed under semantically related regular substitutions. For example,

Lemma 5.28 (Regular Substitutions)**.**

- *If* $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_{\geq \text{D}} T \approx T' \mathbin{@} l$ *and* $L \mid \Psi; \Delta \Vvdash^{\text{D}}_{\geq \text{D}} \delta \approx \delta' : \Gamma$*, then* $L \mid \Psi; \Delta \Vvdash^{\text{D}}_{\geq \text{D}} T[\delta] \approx T'[\delta'] \mathbin{@} l$*.*

$$\frac{\begin{array}{c}\boxed{L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\mathrm{C}} S \mathbin{@} l \qquad L \mid \Psi; \Gamma, x : S \mathbin{@} l \Vvdash^{\mathrm{D}}_{\mathrm{C}} T \mathbin{@} l'} \\ L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l'\end{array}}{L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\mathrm{C}} \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l'}$$

$$\frac{\begin{array}{c}\boxed{\begin{array}{l}L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\mathrm{C}} S \mathbin{@} l \quad L \mid \Psi; \Gamma, x : S \mathbin{@} l \Vvdash^{\mathrm{D}}_{\mathrm{C}} T \mathbin{@} l' \\ L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\mathrm{C}} t : \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l' \quad L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\mathrm{C}} s : S \mathbin{@} l\end{array}} \\ L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} T' \approx T[s/x] \mathbin{@} l' \quad L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} (t : \Pi^{l,l'}(x : S).T)\ s : T' \mathbin{@} l'\end{array}}{L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\mathrm{C}} (t : \Pi^{l,l'}(x : S).T)\ s : T' \mathbin{@} l'}$$

Figure 5.8: Selected rules for semantic judgment for code

- *If* $L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} t \approx t' : T \mathbin{@} l$ *and* $L \mid \Psi; \Delta \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} \delta \approx \delta' : \Gamma$*, then* $L \mid \Psi; \Delta \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} t[\delta] \approx t'[\delta'] : T[\delta] \mathbin{@} l$.

- *If* $L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} \delta \approx \delta' : \Delta$ *and* $L \mid \Psi; \Delta' \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} \delta_1 \approx \delta'_1 : \Gamma$*, then* $L \mid \Phi; \Delta' \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} \delta \circ \delta_1 \approx \delta' \circ \delta'_1 : \Delta$.

Given the semantic judgments for running types, terms, etc., I then encapsulate them with syntactic information about shapes in the semantic judgments for code. In particular, I inductively define $\boxed{L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\mathrm{C}} T \mathbin{@} l}$ for the semantics of code T, $\boxed{L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\mathrm{C}} t : T \mathbin{@} l}$ for the semantics of code t of type T, and $\boxed{L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\mathrm{C}} \delta : \Delta}$ for the semantics of code for regular substitution δ. I give two example rules for the judgments in Fig. 5.8 (see all the rules in Appendix G). The framed premises in the rules come from the typing rules. They recursively record the syntactic information of sub-structures. The other premises are for semantic information. For a type, e.g. a Π type, $L \mid \Psi; \Gamma \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} \Pi^{l,l'}(x : S).T \mathbin{@} l$ denotes that the Π type can be run at both layers D and M. For a term, e.g. a function application, there needs to be two pieces of information: the semantic information of $t\ s$ and the semantic equivalence between T' and $T[s/x]$. Note that $T[s/x]$ is originated from layer C and is brought to layer D via lifting to establish a semantic equivalence with T'. This semantic equivalence characterizes code promotion, where code objects T and s are lifted to layer D for computation. All other rules in the semantic judgments follow this exact pattern to encode both kinds of information. The following semantic lifting lemma extracts the semantic information from the judgments.

Lemma 5.29 (Semantic lifting)**.**

- *If* $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{\text{C}} T \mathbin{@} l$, *then* $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{\geq \text{D}} T \mathbin{@} l$.
- *If* $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{i} t : T \mathbin{@} l$, *then* $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{\geq \text{D}} t : T \mathbin{@} l$.
- *If* $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{i} \delta : \Delta$, *then* $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{\geq \text{D}} \delta : \Delta$.

Since the semantic judgments for MLTT are closed under regular substitutions and the semantic judgments for code are basically typing judgments with extra semantic information, I show that the semantic judgments for code are also closed under regular substitutions, e.g.

Lemma 5.30 (Regular substitutions)**.**

- *If* $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{\text{C}} T \mathbin{@} l$ *and* $L \mid \Psi; \Delta \Vdash^{\text{D}}_{\text{C}} \delta : \Gamma$, *then* $L \mid \Psi; \Delta \Vdash^{\text{D}}_{\text{C}} T[\delta] \mathbin{@} l$.
- *If* $i \in \{\text{V}, \text{C}\}$, $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{i} t : T \mathbin{@} l$ *and* $L \mid \Psi; \Delta \Vdash^{\text{D}}_{i} \delta : \Gamma$, *then* $L \mid \Psi; \Delta \Vdash^{\text{D}}_{i} t[\delta] : T[\delta] \mathbin{@} l$.
- *If* $i \in \{\text{V}, \text{C}\}$, $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{i} \delta : \Delta$ *and* $L \mid \Psi; \Delta' \Vdash^{\text{D}}_{i} \delta' : \Gamma$, *then* $L \mid \Phi; \Delta' \Vdash^{\text{D}}_{i} \delta \circ \delta' : \Delta$.

The closure under regular substitutions is crucial to model code composition semantically. Given a regular substitution δ, though it does not propagate under `box`, i.e. $(\texttt{box}\ e)[\delta] = \texttt{box}\ e$, it might still be applied if it is given as part of a meta-variable. In other words, assuming a meta-substitution σ, $u^{\delta}[\sigma] = \sigma(u)[\delta[\sigma]]$ where $\sigma(u)$ first looks up u in σ, and then the regular substitution $\delta[\sigma]$ is applied to the result of the lookup. The regular substitution lemma ensures that the overall result of code composition still maintains both semantic and syntactic information properly.

Next, I define the symmetrized variants $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{\text{C}} T \approx T' \mathbin{@} l$ as $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{\text{C}} T \mathbin{@} l$ and $T = T'$, and $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{\text{C}} t \approx t' : T \mathbin{@} l$ as $L \mid \Psi; \Gamma \Vdash^{\text{D}}_{\text{C}} t : T \mathbin{@} l$ and $t = t'$. Effectively, I extend the logical relations for types and terms in Sec. 5.4.2 with $i = \text{C}$. This extension allows me to conveniently express the final semantic judgments for the fundamental theorems in Sec. 5.4.7.

$$\frac{}{L \vDash \cdot \approx \cdot} \qquad \frac{\forall\ L' \Longrightarrow L\ .\ L' \vDash \Phi \approx \Phi'}{L \vDash \Phi, u : \mathsf{Ctx} \approx \Phi', u : \mathsf{Ctx}}$$

$$\frac{\begin{array}{l}\mathcal{E} :: \forall\ \theta :: L' \Longrightarrow L\ .\ L' \vDash \Phi \approx \Phi' \qquad i \in \{\textsc{c}, \textsc{d}\} \\ \mathcal{F} :: \forall\ \theta :: L' \Longrightarrow L \text{ and } L' \mid \Psi \Vdash \sigma \approx \sigma' : \mathcal{E}(\theta) \text{ and } k \geq \textsc{d}\ . \\ \qquad L' \mid \Psi \Vvdash^{\textsc{d}}_{k} \Gamma[\sigma] \approx \Gamma'[\sigma'] \hfill (1)\end{array}}{L \vDash \Phi, u : (\Gamma \vdash_i \ @\ l) \approx \Phi', u : (\Gamma' \vdash_i \ @\ l)}$$

$$\frac{\begin{array}{l}\mathcal{E} :: \forall\ \theta :: L' \Longrightarrow L\ .\ L' \vDash \Phi \approx \Phi' \qquad i \in \{\textsc{v}, \textsc{c}\} \\ \mathcal{F} :: \forall\ \theta :: L' \Longrightarrow L \text{ and } L' \mid \Psi \Vdash \sigma \approx \sigma' : \mathcal{E}(\theta) \text{ and } k \geq \textsc{d}\ .\ L' \mid \Psi \Vvdash^{\textsc{d}}_{k} \Gamma[\sigma] \approx \Gamma'[\sigma'] \\ \mathcal{A} :: \forall\ \theta :: L' \Longrightarrow L \text{ and } \mathcal{B} :: L' \mid \Psi \Vdash \sigma \approx \sigma' : \mathcal{E}(\theta) \text{ and } k \geq \textsc{d} \text{ and} \\ \qquad L' \mid \Psi; \Delta \Vvdash^{\textsc{d}}_{k} \delta \approx \delta' : \mathcal{F}(\theta, \mathcal{B}, k)\ .\ L' \mid \Psi; \Delta \Vvdash^{\textsc{d}}_{k} T[\sigma][\delta] \approx T'[\sigma'][\delta'] \ @\ l \qquad (2)\end{array}}{L \vDash \Phi, u : (\Gamma \vdash_i T \ @\ l) \approx \Phi', u : (\Gamma' \vdash_i T' \ @\ l)}$$

Figure 5.9: Logical relations for meta-contexts

5.4.5 Logical Relations for Meta-Contexts and Meta-Substitutions

The semantic judgments for code in Sec. 5.4.4 are used in two places in the semantics: one is the logical relations for meta-contexts and substitutions in this section, and the other is the logical relations for types and terms for layer $\textsc{m}$ in the next section. Continuing the recipe, the logical relations for meta-contexts and substitutions are defined inductive-recursively in the PER style. I first define those for meta-contexts $\boxed{\mathcal{D} :: L \vDash \Psi \approx \Phi}$ inductively in Fig. 5.9. The Kripke structure of the logical relations is in the weakening of universe contexts. All premises $\forall\ L' \Longrightarrow L\ .\ L' \vDash \Phi \approx \Phi'$ builds the Kripke structure into the logical relations. The premise (1) requires the relation between Γ and Γ' to be stable under related meta-substitutions for $k \geq \textsc{d}$. Similarly, the premise (2) requires the relation between T and T' to be stable under both related meta-and regular substitutions.

The logical relations for meta-substitutions $\boxed{L \mid \Psi \Vdash \sigma \approx \sigma' : \mathcal{D}}$ are defined by recursion on those for meta-contexts. I only consider the third case, which is for extension of $(\Gamma \vdash_i \ @\ l)$ and $(\Gamma' \vdash_i \ @\ l)$, where $i \in \{\textsc{c}, \textsc{d}\}$, which needs to satisfy the following conditions:

- $\sigma = \sigma_1, T/u$ and $\sigma' = \sigma'_1, T'/u$, with two related types T and T' to substitute u;
- for all $\theta :: L' \Longrightarrow L$, it holds that $L' \mid \Psi \Vdash \sigma_1 \approx \sigma'_1 : \mathcal{E}(\theta)$, i.e. σ_1 and σ'_1 are recursively related;
- finally, depending on the value of i,
  - if $i = \text{D}$, then $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_{\geq \text{D}} T \approx T' \,@\, l$, so T and T' are related at both layers D and M. In particular, they do not need to be syntactically identical;
  - if $i = \text{C}$, then $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_{\text{C}} T \approx T' \,@\, l$, which stores the syntactic information of T and implies $T = T'$. Due to the semantic lifting lemma, it also implies $L \mid \Psi; \Gamma \Vvdash^{\text{D}}_{\geq \text{D}} T \approx T' \,@\, l$.

In short, the logical relations for meta-substitutions relate two meta-substitutions pointwise, and store the syntactic information of types and terms for contextual kinds at layer C. In Sec. 5.4.7, the fundamental theorems use these logical relations to require types, terms, etc. to be stable under meta-substitutions.

5.4.6 Logical Relations for Layer M

Up until this section, I only use the logical relations for $j = \text{D}$. In this section, I go all the way back and revisit the logical relations for types and terms, but for $i = j = \text{M}$. This section gives semantics to types for meta-programming and is the last step before giving the semantic judgments. The logical relations for types at layer M is defined by extending Fig. 5.6 after setting $i = j = \text{M}$ with Fig. 5.10. As the first step, I first reduce T and T' to some normal types, e.g. contextual types, meta-function types, or universe-polymorphic functions. For contextual types for types to be related, I require Δ and Δ' to be related at both layers D and M (due to $\geq$ D). For contextual types for terms, I in addition require the logical relation between T_1 and T'_1 to be stable under regular substitutions. Since the logical relations for contextual types is not recursive for $i = j = \text{M}$, I can safely restart the universe level at 0. This justifies the syntactic rules as well, where contextual types live on universe level 0. The logical relation of contextual types for terms $L \mid \Psi; \Gamma \Vvdash^{j}_{i} t \approx t' : \texttt{El}(\mathcal{D})$ is defined by the following conditions:

- first, $L \mid \Psi; \Gamma \vdash_{\text{M}} t \rightsquigarrow^* w : \square(\Delta \vdash_{\text{C}} T_1 \,@\, l) \,@\, 0$ and $L \mid \Psi; \Gamma \vdash_{\text{M}} t' \rightsquigarrow^* w' : \square(\Delta' \vdash_{\text{C}} T'_1 \,@\, l) \,@\, 0$, reducing t and t' to WHNFs, and

$$\frac{\begin{array}{l} L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T \rightsquigarrow^{*} \Box(\Delta \vdash_{\mathrm{C}} \ @\, l) @\, 0 \quad L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T' \rightsquigarrow^{*} \Box(\Delta' \vdash_{\mathrm{C}} \ @\, l) @\, 0 \\ L \mid \Psi \Vdash^{\mathrm{D}}_{\geq \mathrm{D}} \Delta \approx \Delta' \end{array}}{L \mid \Psi; \Gamma \Vvdash^{\mathrm{M}}_{\mathrm{M}} T \approx T' @\, 0}$$

$$\frac{\begin{array}{l} L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T \rightsquigarrow^{*} \Box(\Delta \vdash_{\mathrm{C}} T_1 @\, l) @\, 0 \quad L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T' \rightsquigarrow^{*} \Box(\Delta' \vdash_{\mathrm{C}} T_1' @\, l) @\, 0 \\ L \mid \Psi \Vdash^{\mathrm{D}}_{\geq \mathrm{D}} \Delta \approx \Delta' \quad L \mid \Psi; \Delta \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} T_1 \approx T_1' @\, l \end{array}}{L \mid \Psi; \Gamma \Vvdash^{\mathrm{M}}_{\mathrm{M}} T \approx T' @\, 0}$$

$$\frac{\begin{array}{l} L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T \rightsquigarrow^{*} (g : \mathsf{Ctx}) \Rightarrow^{l} T_1 @\, l \quad L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T' \rightsquigarrow^{*} (g : \mathsf{Ctx}) \Rightarrow^{l} T_1' @\, l \\ \mathcal{E} :: \forall\ \psi :: L' \mid \Phi; \Delta'' \Longrightarrow_i L \mid \Psi; \Gamma \text{ and } L' \mid \Phi \Vdash^{\mathrm{D}}_{\geq \mathrm{D}} \Delta \approx \Delta' \ . \\ \qquad L' \mid \Phi; \Delta'' \Vvdash^{\mathrm{M}}_{\mathrm{M}} T_1[\Delta/g] \approx T_1'[\Delta'/g] @\, l \end{array}}{L \mid \Psi; \Gamma \Vvdash^{\mathrm{M}}_{\mathrm{M}} T \approx T' @\, l}$$

$$\frac{\begin{array}{l} L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T \rightsquigarrow^{*} (U : (\Delta \vdash_{\mathrm{D}} \ @\, l)) \Rightarrow^{l'} T_1 @\, l' \\ L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T' \rightsquigarrow^{*} (U : (\Delta' \vdash_{\mathrm{D}} \ @\, l)) \Rightarrow^{l'} T_1' @\, l' \\ L' \mid \Phi \Vdash^{\mathrm{D}}_{\geq \mathrm{D}} \Delta \approx \Delta' \\ \mathcal{E} :: \forall\ \psi :: L' \mid \Phi; \Delta'' \Longrightarrow_i L \mid \Psi; \Gamma \text{ and } L' \mid \Phi; \Delta \Vvdash^{\mathrm{D}}_{\geq \mathrm{D}} T_2 \approx T_2' @\, l \ . \\ \qquad L' \mid \Phi; \Delta'' \Vvdash^{\mathrm{M}}_{\mathrm{M}} T_1[T_2/U] \approx T_1'[T_2'/U] @\, l' \end{array}}{L \mid \Psi; \Gamma \Vvdash^{\mathrm{M}}_{\mathrm{M}} T \approx T' @\, l'}$$

$$\frac{\begin{array}{l} L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T \rightsquigarrow^{*} \overrightarrow{\ell} \Rightarrow^{l} T_1 @\, \omega \quad L \mid \Psi; \Gamma \vdash_{\mathrm{M}} T' \rightsquigarrow^{*} \overrightarrow{\ell} \Rightarrow^{l} T_1' @\, \omega \\ \mathcal{E} :: \forall\ \psi :: L' \mid \Phi; \Delta \Longrightarrow_i L \mid \Psi; \Gamma \text{ and } \overrightarrow{l} \text{ that are well-formed in } L' \text{ and } |\overrightarrow{\ell}| = |\overrightarrow{l}| \ . \\ \qquad L' \mid \Phi; \Delta \Vvdash^{\mathrm{M}}_{\mathrm{M}} T_1[\overrightarrow{l}/\overrightarrow{\ell}] \approx T_1'[\overrightarrow{l}/\overrightarrow{\ell}] @\, l[\overrightarrow{l}/\overrightarrow{\ell}] \end{array}}{L \mid \Psi; \Gamma \Vvdash^{\mathrm{M}}_{\mathrm{M}} T \approx T' @\, \omega} \quad (3)$$

Figure 5.10: Logical relations for types at layer M

- $L \mid \Psi; \Gamma \vdash_{\mathrm{M}} w \simeq w' : \Box(\Delta \vdash_{\mathrm{C}} T_1 @\, l) @\, 0$, i.e. the WHNFs are generically equivalent, and
- finally, $L \mid \Psi; \Gamma \Vvdash w \simeq w' : \Box(\Delta \vdash_{\mathrm{C}} T_1 @\, l)$, which is an inductive relation of two

cases:

$$\frac{L \mid \Psi; \Delta \Vdash\!\!\vDash^{\mathrm{D}}_{\mathrm{C}} t_1 : T_1 \mathbin{@} l}{L \mid \Psi; \Gamma \Vdash\!\!\vDash \texttt{box}\ t_1 \simeq \texttt{box}\ t_1 : \square(\Delta \vdash_{\mathrm{C}} T_1 \mathbin{@} l)}$$

$$\frac{L \mid \Psi; \Gamma \vdash_{\mathrm{M}} \nu \sim \nu' : \square(\Delta \vdash_{\mathrm{C}} T_1 \mathbin{@} l) \mathbin{@} 0}{L \mid \Psi; \Gamma \Vdash\!\!\vDash \nu \simeq \nu' : \square(\Delta \vdash_{\mathrm{C}} T_1 \mathbin{@} l)}$$

The last condition relates w and w' on the contextual types $\square(\Delta \vdash_{\mathrm{C}} T_1 \mathbin{@} l)$ in two cases. Either $w = w' = \texttt{box}\ t_1$ for some t_1, which accompanies its semantic judgment for code, i.e. $L \mid \Psi; \Delta \Vdash\!\!\vDash^{\mathrm{D}}_{\mathrm{C}} t_1 : T_1 \mathbin{@} l$. When recursing on w, in the semantics, the mutual recursion principle for code is in fact interpreted as the mutual induction principle for $L \mid \Psi; \Delta \Vdash\!\!\vDash^{\mathrm{D}}_{\mathrm{C}} t_1 : T_1 \mathbin{@} l$. On the other hand, if one composes t_1 with some other code, the regular substitution lemma (Lemma 5.30) ensures that the result code still maintains proper semantic and syntactic information. Finally, if t_1 is run as a program, then the semantic lifting lemma (Lemma 5.29) is applied to obtained the semantic information for execution, in conjunction with the layering restriction lemma (Lemma 5.14) if t_1 is run at layer M. In conclusion, the semantics justifies all usages of code. If w and w' are neutral, then I cannot say more than that they are generically equivalent.

The next two cases are meta-functions for contexts and for types. They are very similar to Π types. Their premises $\mathcal{E}$ extensionally quantify related inputs so that the output types remain related. In fact, meta-functions are even simpler than Π types because the relations of inputs are obtain from $j = \mathrm{D}$, so they are not even recursive in $i = j = \mathrm{M}$. The relations for their terms are similar to Π types as well. I first require these terms to reduce to WHNFs and then applying these WHNFs to related inputs produces related outputs.

The last case in the logical relations is the universe-polymorphic functions. In the premise (3), I substitute some arbitrary well-formed $\overrightarrow{l}$ for $\overrightarrow{\ell}$ in l. I cannot know in particular which exact universe level the result of the substitution is. The only thing that I am sure about is that $l[\overrightarrow{l}/\overrightarrow{\ell}]$ is some finite, well-formed universe level. Therefore, in order to refer to any finite universe level, universe-polymorphic functions must be modeled on level ω, hence requiring a transfinite recursion on universe levels in the definition of the logical relations.

The logical relations for regular contexts and substitutions for $i = j = \text{M}$ have been defined in Fig. 5.7.

5.4.7 Semantic Judgments and Fundamental Theorems

Finally, I define the semantic judgments for DeLaM. The semantic judgments states the stability of principal objects in the judgments under all substitutions at all higher layers. First, I define the semantic judgment for meta-contexts $\boxed{L \Vdash \Psi}$ as for all $L' \vdash \phi : L$, it holds that $L' \vDash \Psi[\phi] \approx \Psi[\phi]$. Then the semantic judgment for equivalent regular contexts $\boxed{L \mid \Psi \Vdash_i \Gamma \approx \Delta}$ where $i \in \{\text{D}, \text{M}\}$ is defined as a conjunction of $L \Vdash \Psi$ and for all $L' \vdash \phi : L$, $L' \mid \Phi \VDash \sigma \approx \sigma' : \Psi[\phi]$ and $k \geq i$, it holds that $L' \mid \Phi \Vdash\!\!\vDash_k^{\Uparrow(i)} \Gamma[\phi][\sigma] \approx \Delta[\phi][\sigma']$. Effectively, this judgment says that the relation between Γ and Δ is stable under all universe and meta-substitutions at all layers $k \geq i$. Note that in the conclusion, I always set $j = \Uparrow(i)$, which is where regular contexts live when terms live at layer i. Its asymmetric version $\boxed{L \mid \Psi \Vdash_i \Gamma}$ requires both sides to be equal: $L \mid \Psi \Vdash_i \Gamma \approx \Gamma$.

Next, I define the semantic judgment for types $\boxed{L \mid \Psi; \Gamma \Vdash_i T \approx T' \mathbin{@} l}$. It follows the same principle. I first require $L \mid \Psi \Vdash_{\Uparrow(i)} \Gamma$, and then for all $L' \vdash \phi : L$, $L' \mid \Phi \Vdash \sigma \approx \sigma' : \Psi[\phi]$, $k \geq i$ and $L' \mid \Phi; \Delta \Vdash\!\!\vDash_k^{\Uparrow(i)} \delta \approx \delta' : \Gamma[\phi][\sigma]$, it holds that

$$L' \mid \Phi; \Delta \Vdash\!\!\vDash_k^{\Uparrow(i)} T[\phi][\sigma][\delta] \approx T'[\phi][\sigma'][\delta'] \mathbin{@} l[\phi]$$

In other words, the relation between T and T' is stable under all substitutions at all layers above i. Then the semantic judgment for terms $\boxed{L \mid \Psi; \Gamma \Vdash_i t \approx t' : T \mathbin{@} l}$ is defined in the same way, except that I require $L \mid \Psi; \Gamma \Vdash_{\Uparrow(i)} T \mathbin{@} l$, and at the end, the conclusion is changed to

$$L' \mid \Phi; \Delta \Vdash\!\!\vDash_k^{\Uparrow(i)} t[\phi][\sigma][\delta] \approx t'[\phi][\sigma'][\delta'] : T[\phi][\sigma][\delta] \mathbin{@} l[\phi]$$

The last semantic judgment is for regular substitutions $\boxed{L \mid \Psi; \Gamma \Vdash_i \delta \approx \delta' : \Delta}$, which is again defined similarly. These semantic judgments set up the right inductive invariants for the fundamental theorems, so that they can be proved by induction on the syntactic judgments. The fundamental theorems are stated as follows:

Theorem 5.31 (Fundamental)**.**

- *If* $L \vdash \Psi$*, then* $L \Vdash \Psi$*.*
- *If* $L \mid \Psi \vdash_i \Gamma$ *and* $i \in \{\textsc{d}, \textsc{m}\}$*, then* $L \mid \Psi \Vdash_i \Gamma$*.*
- *If* $L \mid \Psi \vdash_i \Gamma \approx \Delta$ *and* $i \in \{\textsc{d}, \textsc{m}\}$*, then* $L \mid \Psi \Vdash_i \Gamma \approx \Delta$*.*
- *If* $L \mid \Psi; \Gamma \vdash_i T \mathbin{@} l$*, then* $L \mid \Psi; \Gamma \Vdash_i T \mathbin{@} l$*.*
- *If* $L \mid \Psi; \Gamma \vdash_i T \approx T' \mathbin{@} l$*, then* $L \mid \Psi; \Gamma \Vdash_i T \approx T' \mathbin{@} l$*.*
- *If* $L \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l$*, then* $L \mid \Psi; \Gamma \Vdash_i t : T \mathbin{@} l$*.*
- *If* $L \mid \Psi; \Gamma \vdash_i t \approx t' : T \mathbin{@} l$*, then* $L \mid \Psi; \Gamma \Vdash_i t \approx t' : T \mathbin{@} l$*.*
- *If* $L \mid \Psi; \Gamma \vdash_i \delta : \Delta$*, then* $L \mid \Psi; \Gamma \Vdash_i \delta : \Delta$*.*
- *If* $L \mid \Psi; \Gamma \vdash_i \delta \approx \delta' : \Delta$*, then* $L \mid \Psi; \Gamma \Vdash_i \delta \approx \delta' : \Delta$*.*

The proof of the fundamental theorems is a more complicated version of what is shown in Sec. 4.9 and follows a very interesting pattern, where I must work backwards following the order of layers. More specifically, to prove the first statement of Theorem 5.31 in an induction, if a typing rule is only available for $i = \textsc{m}$, then I proceed normally as in other type theories. However, if a rule is available at multiple layers, e.g. the congruence rule for Π types, then I have multiple statements to prove.

First, let $i = \textsc{m}$. Then the semantic judgment eventually requires a proof of $L' \mid \Phi; \Delta \Vdash\!\vDash^{\textsc{m}}_{\textsc{m}} T[\phi][\sigma][\delta] \approx T'[\phi][\sigma'][\delta'] \mathbin{@} l[\phi]$, since $k \geq \textsc{m}$ means $k = \textsc{m}$. Next let $i = \textsc{d}$. In this case, $k \in \{\textsc{d}, \textsc{m}\}$, so the proof requires $L' \mid \Phi; \Delta \Vdash\!\vDash^{\textsc{d}}_{k} T[\phi][\sigma][\delta] \approx T'[\phi][\sigma'][\delta'] \mathbin{@} l[\phi]$. This case is very similar to the case for $i = \textsc{m}$, with very minor differences in layers. When $i = \textsc{c}$, then $k \geq \textsc{c}$, so k now can take three different values. As the proof obligation, I need to prove $L' \mid \Phi; \Delta \Vdash\!\vDash^{\textsc{d}}_{k} T[\phi][\sigma][\delta] \approx T'[\phi][\sigma'][\delta'] \mathbin{@} l[\phi]$. Note that here cases for $k \geq \textsc{d}$ actually have been proved when $i = \textsc{d}$, so the only addition is to handle $k = \textsc{c}$. But then even this case is quite trivial, because when $k = \textsc{c}$, the goal becomes the semantic judgment for code of types, which is just a repackaging of the cases for $k \geq \textsc{d}$. Interested readers may refer to the technical report (Hu and Pientka, 2024a) for detailed proofs.

If we consider what information layers contain, this proof pattern makes even more sense. For a term at layer M, the only information that it has is how it runs at layer M. Meanwhile if a term is from layer D, then it can be run at both layers D and M due to lifting. For code from layer C, I must in addition keep track of its syntactic shape, which adds strictly more information on top of its running information. The proof of the fundamental theorems is similar to opening an onion: the proof peels off and assigns semantics to DeLaM from the outside layer by layer as it adds more and more information, until the very end when DeLaM is entirely modeled.

If I set all substitutions to identities and $k = i$, then the fundamental theorems simply imply the logical relations.

Corollary 5.32 (Completeness of logical relations)**.** *If* $i \in \{\textsc{d}, \textsc{m}\}$*,*

- *If* $L \mid \Psi; \Gamma \vdash_i T \approx T' \;@\; l$*, then* $L \mid \Psi; \Gamma \Vdash^i_i T \approx T' \;@\; l$ *and* $L \mid \Psi; \Gamma \vdash_i T \simeq T' \;@\; l$*.*
- *If* $L \mid \Psi; \Gamma \vdash_i t \approx t' : T \;@\; l$*, then* $L \mid \Psi; \Gamma \Vdash^i_i t \approx t' : T \;@\; l$ *and* $L \mid \Psi; \Gamma \vdash_i t \simeq t' : T \;@\; l$*.*

Note that $j = i = \Uparrow(i)$ because $i \in \{\textsc{d}, \textsc{m}\}$. In other words, syntactically equivalent types or terms are also logically related and thus generically equivalent. In Sec. 5.5.3, where I set the generic equivalences to the conversion checking algorithm, this corollary immediately proves the completeness of the conversion checking algorithm.

5.5 Consequences of Fundamental Theorems

With the fundamental theorems, I can now instantiate the generic equivalences to obtain important conclusions like weak normalization, injectivity of type constructors, consistency and the decidability of convertibility. In this section, we focus on deriving these conclusions.

5.5.1 First Instantiation: Syntactic Equivalence

The first instantiation assigns syntactic equivalence for types and terms to the generic equivalences. In this case, the laws are quite trivial to prove. The first important theorem to extract from the fundamental theorems is weak normalization.

Theorem 5.33 (Weak normalization)**.** *If* $i \in \{\textsc{d}, \textsc{m}\}$*, then*

- *if* $L \mid \Psi; \Gamma \vdash_i T \mathbin{@} l$*, then for some WHNF* W*,* $L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* W \mathbin{@} l$*;*

- *if* $L \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l$*, then for some WHNF* w*,* $L \mid \Psi; \Gamma \vdash_i t \rightsquigarrow^* w : T \mathbin{@} l$*.*

It is only a weak normalization theorem, because I fix one specific reduction strategy. The weak normalization theorem says that well-formed types and terms always reduce to some WHNFs. This theorem is proved from Corollary 5.32, where weak normalization is built in the logical relations.

The next important theorem is injectivity of type constructors.

Theorem 5.34 (Injectivity of type constructors)**.**

- *If* $L \mid \Psi; \Gamma \vdash_i \Pi^{l,l'}(x : S).T \approx \Pi^{l,l'}(x : S').T' \mathbin{@} l \sqcup l'$ *and* $i \in \{\text{D}, \text{M}\}$*, then* $L \mid \Psi; \Gamma \vdash_i S \approx S' \mathbin{@} l$ *and* $L \mid \Psi; \Gamma, x : S \mathbin{@} l \vdash_i T \approx T' \mathbin{@} l'$*.*

- *If* $L \mid \Psi; \Gamma \vdash_{\text{M}} \square(\Delta \vdash_{\text{C}} \mathbin{@} l) \approx \square(\Delta' \vdash_{\text{C}} \mathbin{@} l) \mathbin{@} 0$*, then* $L \mid \Psi \vdash_{\text{D}} \Delta \approx \Delta'$*.*

- *If* $L \mid \Psi; \Gamma \vdash_{\text{M}} \square(\Delta \vdash_{\text{C}} T \mathbin{@} l) \approx \square(\Delta' \vdash_{\text{C}} T' \mathbin{@} l) \mathbin{@} 0$*, then* $L \mid \Psi \vdash_{\text{D}} \Delta \approx \Delta'$ *and* $L \mid \Psi; \Delta \vdash_{\text{D}} T \approx T' \mathbin{@} l$*.*

The theorem says that type constructors are injective w.r.t. syntactic equivalence. This theorem is proved from Corollary 5.32. The logical relations for types require matching sub-structures to be related, from which I extract their equivalences.

The last theorem that I obtain from this instantiation is the consistency theorem.

Theorem 5.35 (Consistency)**.** *There is no term* t *that satisfies this typing judgment:*

$$\cdot \mid \cdot; \cdot \vdash_{\text{M}} t : \ell \Longrightarrow^{1+\ell} \Pi^{1+\ell,\ell}(x : \textit{Ty}_\ell).\textit{El}^\ell\ x \mathbin{@} \omega$$

The consistency theorem says that, it is not possible to generically construct a term of any type on any universe level. The proof proceeds as follows. First, this theorem is the same as proving that there is no t' such that $\ell \mid \cdot; x : \texttt{Ty}_\ell \mathbin{@} 1 + \ell \vdash_{\text{M}} t' : \texttt{El}^\ell\ x \mathbin{@} \ell$. Let us assume such t'. Then it has a neutral type $\texttt{El}^\ell\ x$, so t' must reduce to some neutral ν by the logical relations of $\texttt{El}^\ell\ x$. Then an induction on ν shows that ν must be eventually blocked by x. But x as a neutral type cannot be eliminated, nor can it have type $\texttt{El}^\ell\ x$, so a contradiction is established.

$\boxed{L \mid \Psi; \Gamma \vdash_i T \mathrel{\hat{\Longleftrightarrow}} T' \mathbin{@} l}$ and $\boxed{L \mid \Psi; \Gamma \vdash_i V \longleftrightarrow V' \mathbin{@} l}$ and $\boxed{L \mid \Psi; \Gamma \vdash_i W \Longleftrightarrow W' \mathbin{@} l}$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* W \mathbin{@} l \\ L \mid \Psi; \Gamma \vdash_i T' \rightsquigarrow^* W' \mathbin{@} l \\ L \mid \Psi; \Gamma \vdash_i W \Longleftrightarrow W' \mathbin{@} l \end{array}}{L \mid \Psi; \Gamma \vdash_i T \mathrel{\hat{\Longleftrightarrow}} T' \mathbin{@} l} \qquad \frac{\begin{array}{c} L \mid \Psi \vdash_i \Gamma \qquad u : (\Delta \vdash_{i'} \ \mathbin{@} l) \in \Psi \\ i' \leq i \qquad L \mid \Psi; \Gamma \vdash_i \delta \mathrel{\hat{\Longleftrightarrow}} \delta' : \Delta \end{array}}{L \mid \Psi; \Gamma \vdash_i u^{\delta} \longleftrightarrow u^{\delta'} \mathbin{@} l}$$

$$\frac{L \mid \Psi; \Gamma \vdash_i \nu \longleftrightarrow \nu' : \mathtt{Ty}_l \mathbin{@} 1 + l}{L \mid \Psi; \Gamma \vdash_i \mathtt{El}^l\ \nu \longleftrightarrow \mathtt{El}^l\ \nu' \mathbin{@} l} \qquad \frac{L \mid \Psi \vdash_i \Gamma}{L \mid \Psi; \Gamma \vdash_i \mathtt{Nat} \Longleftrightarrow \mathtt{Nat} \mathbin{@} 0}$$

$$\frac{L \mid \Psi \vdash_i \Gamma}{L \mid \Psi; \Gamma \vdash_i \mathtt{Ty}_l \Longleftrightarrow \mathtt{Ty}_l \mathbin{@} 1 + l} \qquad \frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_i S \mathrel{\hat{\Longleftrightarrow}} S' \mathbin{@} l \\ L \mid \Psi; \Gamma, x : S \mathbin{@} l \vdash_i T \mathrel{\hat{\Longleftrightarrow}} T' \mathbin{@} l' \end{array}}{L \mid \Psi; \Gamma \vdash_i \Pi^{l,l'}(x : S).T \Longleftrightarrow \Pi^{l,l'}(x : S').T' \mathbin{@} l \sqcup l'}$$

$$\frac{L \mid \Psi; \Gamma \vdash_i V \longleftrightarrow V' \mathbin{@} l}{L \mid \Psi; \Gamma \vdash_i V \Longleftrightarrow V' \mathbin{@} l} \qquad \frac{L \mid \Psi \vdash_{\text{M}} \Gamma \qquad L, \overrightarrow{\ell} \mid \Psi; \Gamma \vdash_{\text{M}} T \mathrel{\hat{\Longleftrightarrow}} T' \mathbin{@} l}{L \mid \Psi; \Gamma \vdash_{\text{M}} (\overrightarrow{\ell} \Rightarrow^{l} T) \Longleftrightarrow (\overrightarrow{\ell} \Rightarrow^{l'} T') \mathbin{@} \omega}$$

$$\frac{L \mid \Psi \vdash_{\text{M}} \Gamma \qquad L \mid \Psi \vdash_{\text{D}} \Delta \mathrel{\hat{\Longleftrightarrow}} \Delta'}{L \mid \Psi; \Gamma \vdash_{\text{M}} \square(\Delta \vdash_{\text{C}} \ \mathbin{@} l) \Longleftrightarrow \square(\Delta' \vdash_{\text{C}} \ \mathbin{@} l) \mathbin{@} 0}$$

$$\frac{L \mid \Psi \vdash_{\text{M}} \Gamma \qquad L \mid \Psi \vdash_{\text{D}} \Delta \mathrel{\hat{\Longleftrightarrow}} \Delta' \qquad L \mid \Psi; \Delta \vdash_{\text{D}} T \mathrel{\hat{\Longleftrightarrow}} T' \mathbin{@} l}{L \mid \Psi; \Gamma \vdash_{\text{M}} \square(\Delta \vdash_{\text{C}} T \mathbin{@} l) \Longleftrightarrow \square(\Delta' \vdash_{\text{C}} T' \mathbin{@} l) \mathbin{@} 0}$$

$\boxed{L \mid \Psi; \Gamma \vdash_i t \mathrel{\hat{\Longleftrightarrow}} t' : T \mathbin{@} l}$ and $\boxed{L \mid \Psi; \Gamma \vdash_i w \Longleftrightarrow w' : W \mathbin{@} l}$ for terms

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* W \mathbin{@} l \qquad L \mid \Psi; \Gamma \vdash_i t \rightsquigarrow^* w : T \mathbin{@} l \\ L \mid \Psi; \Gamma \vdash_i t' \rightsquigarrow^* w' : T \mathbin{@} l \qquad L \mid \Psi; \Gamma \vdash_i w \Longleftrightarrow w' : W \mathbin{@} l \end{array}}{L \mid \Psi; \Gamma \vdash_i t \mathrel{\hat{\Longleftrightarrow}} t' : T \mathbin{@} l}$$

$$\frac{L \mid \Psi; \Gamma \vdash_i \nu \longleftrightarrow \nu' : W \mathbin{@} l}{L \mid \Psi; \Gamma \vdash_i \nu \Longleftrightarrow \nu' : V \mathbin{@} l}$$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_i w : \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l' \qquad L \mid \Psi; \Gamma \vdash_i w' : \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l' \\ L \mid \Psi; \Gamma, x : S \mathbin{@} l \vdash_i (w : \Pi^{l,l'}(x : S).T)\ x \mathrel{\hat{\Longleftrightarrow}} (w' : \Pi^{l,l'}(x : S).T)\ x : T \mathbin{@} l' \end{array}}{L \mid \Psi; \Gamma \vdash_i w \Longleftrightarrow w' : \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l'}$$

$$\frac{L \mid \Psi \vdash_{\text{M}} \Gamma \qquad L \mid \Psi; \Delta \vdash_{\text{C}} t : T \mathbin{@} l \qquad t = t'}{L \mid \Psi; \Gamma \vdash_{\text{M}} \mathtt{box}\ t \Longleftrightarrow \mathtt{box}\ t' : \square(\Delta \vdash_{\text{C}} T \mathbin{@} l) \mathbin{@} 0}$$

$$\frac{L \mid \Psi; \Gamma \vdash_{\text{M}} \nu \longleftrightarrow \nu' : \square(\Delta \vdash_{\text{C}} T \mathbin{@} l) \mathbin{@} 0}{L \mid \Psi; \Gamma \vdash_{\text{M}} \nu \Longleftrightarrow \nu' : \square(\Delta \vdash_{\text{C}} T \mathbin{@} l) \mathbin{@} 0}$$

Figure 5.11: Selected rules for conversion checking algorithm

$\boxed{L \mid \Psi; \Gamma \vdash_i \nu \longleftrightarrow \nu' : W \mathbin{@} l}$ and $\boxed{L \mid \Psi; \Gamma \vdash_i \nu \overset{\wedge}{\longleftrightarrow} \nu' : T \mathbin{@} l}$ for neutral terms

$$\frac{L \mid \Psi; \Gamma \vdash_i \nu \overset{\wedge}{\longleftrightarrow} \nu' : T \mathbin{@} l \qquad L \mid \Psi; \Gamma \vdash_i T \rightsquigarrow^* W \mathbin{@} l}{L \mid \Psi; \Gamma \vdash_i \nu \longleftrightarrow \nu' : W \mathbin{@} l} \qquad \frac{L \mid \Psi \vdash_i \Gamma \qquad x : T \mathbin{@} l \in \Gamma}{L \mid \Psi; \Gamma \vdash_i x \overset{\wedge}{\longleftrightarrow} x : T \mathbin{@} l}$$

$$\frac{L \mid \Psi \vdash_i \Gamma \qquad u : (\Delta \vdash_{i'} T \mathbin{@} l) \in \Psi \qquad i' \le i \qquad L \mid \Psi; \Gamma \vdash_i \delta \overset{\wedge}{\Longleftrightarrow} \delta' : \Delta}{L \mid \Psi; \Gamma \vdash_i u^{\delta} \overset{\wedge}{\longleftrightarrow} u^{\delta'} : T[\delta] \mathbin{@} l}$$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \vdash_i S \overset{\wedge}{\Longleftrightarrow} S' \mathbin{@} l \qquad L \mid \Psi; \Gamma, x : S \mathbin{@} l \vdash_i T \overset{\wedge}{\Longleftrightarrow} T' \mathbin{@} l' \\ L \mid \Psi; \Gamma \vdash_i \nu \longleftrightarrow \nu' : \Pi^{l,l'}(x : S'').T'' \mathbin{@} l \sqcup l' \qquad L \mid \Psi; \Gamma \vdash_i s \overset{\wedge}{\Longleftrightarrow} s' : S'' \mathbin{@} l \end{array}}{L \mid \Psi; \Gamma \vdash_i (\nu : \Pi^{l,l'}(x : S).T)\ s \overset{\wedge}{\longleftrightarrow} (\nu' : \Pi^{l,l'}(x : S').T')\ s' : T[s/x] \mathbin{@} l'}$$

$$\frac{\begin{array}{c} L \mid \Psi \vdash_{\text{M}} \Gamma \qquad L \mid \Psi \vdash_{\text{D}} \Delta \overset{\wedge}{\Longleftrightarrow} \Delta' \qquad L \mid \Psi; \Gamma, x : \square(\Delta \vdash_{\text{C}} \mathbin{@} l') \mathbin{@} 0 \vdash_{\text{M}} M \overset{\wedge}{\Longleftrightarrow} M' \mathbin{@} l \\ L \mid \Psi, u : (\Delta \vdash_{\text{C}} \mathbin{@} l'); \Gamma \vdash_{\text{M}} t \overset{\wedge}{\Longleftrightarrow} t' : M[\texttt{box}\ u^{\textsf{id}}/x] \mathbin{@} l \\ L \mid \Psi; \Gamma \vdash_{\text{M}} \nu \longleftrightarrow \nu' : \square(\Delta \vdash_{\text{C}} \mathbin{@} l') \mathbin{@} 0 \\ \text{left} = \texttt{letbox}^{l}_{x.M}\ u \leftarrow (\nu : \square(\Delta \vdash_{\text{C}} \mathbin{@} l'))\ \texttt{in}\ t \\ \text{right} = \texttt{letbox}^{l}_{x.M'}\ u \leftarrow (\nu' : \square(\Delta' \vdash_{\text{C}} \mathbin{@} l'))\ \texttt{in}\ t' \end{array}}{L \mid \Psi; \Gamma \vdash_{\text{M}} \text{left} \overset{\wedge}{\longleftrightarrow} \text{right} : M[t/x] \mathbin{@} l}$$

Figure 5.12: Selected rules for conversion checking algorithm for neutral terms

5.5.2 Conversion Checking

Following Abel et al. (2018) and Chapter 4, I define the conversion checking algorithm. Due to layering and meta-programming constructs, there are more operations in the conversion checking algorithm than that by Abel et al., because I also need to compare regular contexts and substitutions. The conversion checking algorithm is split into two modes: checking and inference. In the checking mode, the algorithm returns true or false, while in the inference mode, the algorithm infers a universe level and potentially a type on that level for neutral terms. Selected rules for the algorithm are defined Fig. 5.11 and 5.12. The algorithm is layered at $i \in \{\text{D}, \text{M}\}$, the only two layers where interesting computation occurs. The main entry points are $\boxed{L \mid \Psi; \Gamma \vdash_i T \overset{\wedge}{\Longleftrightarrow} T' \mathbin{@} l}$, which checks the convertibility of T and T' on level l, and $\boxed{L \mid \Psi; \Gamma \vdash_i t \overset{\wedge}{\Longleftrightarrow} t' : T \mathbin{@} l}$, which checks the convertibility of t and t' of type T on level l. Both entry points first reduce the inputs to WHNFs. Then the WHNFs are compared by $\boxed{L \mid \Psi; \Gamma \vdash_i W \Longleftrightarrow W' \mathbin{@} l}$ and $\boxed{L \mid \Psi; \Gamma \vdash_i w \Longleftrightarrow w' : W \mathbin{@} l}$, which actually do the case analyses based on the shapes. At some point, the checking for WHNFs enters the inference mode, in order to compare neutrals. The neutral form checking algorithm for types $\boxed{L \mid \Psi; \Gamma \vdash_i V \longleftrightarrow V' \,@\, l}$ returns the universe level l of V and V' if they are convertible. The neutral form checking algorithm for terms is a bit more complex. The actual worker is $\boxed{L \mid \Psi; \Gamma \vdash_i \nu \mathrel{\hat{\longleftrightarrow}} \nu' : T \,@\, l}$, which returns a type T and its universe level l, if ν and ν' are convertible. However, before returning the type T to the checking mode, $\boxed{L \mid \Psi; \Gamma \vdash_i \nu \longleftrightarrow \nu' : W \,@\, l}$ first reduces it to a WHNF, so the output type of this algorithm is a normal type W. Finally, the checking mode for types and terms is generalized to regular contexts and substitutions pointwise, giving $\boxed{L \mid \Psi \vdash_{\text{D}} \Gamma \mathrel{\hat{\Longleftrightarrow}} \Delta}$ and $\boxed{L \mid \Psi; \Gamma \vdash_i \delta \mathrel{\hat{\Longleftrightarrow}} \delta' : \Delta}$. The checking mode for regular contexts is needed when I compare contextual types. I also need to check the convertibility between regular substitutions when I encounter neutral meta-variables u^δ and $u^{\delta'}$, in which case I need to compare δ and δ'.

The conversion checking algorithm is very close to that by Abel et al. in its spirit. The checking mode for terms is type-directed. I employ a suitable check according to the shape of the input type. For functions, I check in suitably extended contexts, and for other types, I case-analyze terms accordingly. The inference mode is syntax-directed. It fails immediately if two neutral forms fail to have the same syntactic structure.

5.5.3 Second Instantiation: Conversion Checking Algorithm

In the second instantiation, I assign the conversion checking algorithm to the generic equivalences: $L \mid \Psi; \Gamma \vdash_i V \sim V' \,@\, l$ as $L \mid \Psi; \Gamma \vdash_i V \longleftrightarrow V' \,@\, l$, $L \mid \Psi; \Gamma \vdash_i T \simeq T' \,@\, l$ as $L \mid \Psi; \Gamma \vdash_i T \mathrel{\hat{\Longleftrightarrow}} T' \,@\, l$, and $L \mid \Psi; \Gamma \vdash_i t \simeq t' : T \,@\, l$ as $L \mid \Psi; \Gamma \vdash_i t \mathrel{\hat{\Longleftrightarrow}} t' : T \,@\, l$. The most complex case is $L \mid \Psi; \Gamma \vdash_i \nu \sim \nu' : T \,@\, l$, which is assigned a conjunction of $L \mid \Psi; \Gamma \vdash_i \nu \mathrel{\hat{\longleftrightarrow}} \nu' : T' \,@\, l$ and $L \mid \Psi; \Gamma \vdash_i T \approx T' \,@\, l$. Here I always take $i \in \{\text{D}, \text{M}\}$. First, the soundness lemma for the conversion checking algorithm is proved by simple mutual induction:

Lemma 5.36 (Soundness)**.**

- *If* $L \mid \Psi; \Gamma \vdash_i T \mathrel{\hat{\Longleftrightarrow}} T' \,@\, l$, *then* $L \mid \Psi; \Gamma \vdash_i T \approx T' \,@\, l$.
- *If* $L \mid \Psi; \Gamma \vdash_i t \mathrel{\hat{\Longleftrightarrow}} t' : T \,@\, l$, *then* $L \mid \Psi; \Gamma \vdash_i t \approx t' : T \,@\, l$.

The completeness lemma is established by Corollary 5.32.

Lemma 5.37 (Completeness)**.**

- *If* $L \mid \Psi; \Gamma \vdash_i T \approx T' \mathbin{@} l$*, then* $L \mid \Psi; \Gamma \vdash_i T \mathrel{\hat{\Longleftrightarrow}} T' \mathbin{@} l$*.*
- *If* $L \mid \Psi; \Gamma \vdash_i t \approx t' : T \mathbin{@} l$*, then* $L \mid \Psi; \Gamma \vdash_i t \mathrel{\hat{\Longleftrightarrow}} t' : T \mathbin{@} l$*.*

Therefore, conversion checking and syntactic equivalence are logically equivalent. To establish the decidability of convertibility, I need the following lemma, which establishes the decidability of conversion checking between reflexively convertible types or terms.

Lemma 5.38 (Decidability of conversion checking)**.**

- *if* $L \mid \Phi; \Delta \vdash_i T \mathrel{\hat{\Longleftrightarrow}} T \mathbin{@} l$*,* $L \mid \Psi; \Gamma \vdash_i T' \mathrel{\hat{\Longleftrightarrow}} T' \mathbin{@} l$*,* $L \vdash \Phi \approx \Psi$ *and* $L \mid \Phi \vdash_i \Delta \approx \Gamma$*, then whether* $L \mid \Phi; \Delta \vdash_i T \mathrel{\hat{\Longleftrightarrow}} T' \mathbin{@} l$ *is decidable.*
- *if* $L \mid \Phi; \Delta \vdash_i t \mathrel{\hat{\Longleftrightarrow}} t : T \mathbin{@} l$*,* $L \mid \Psi; \Gamma \vdash_i t' \mathrel{\hat{\Longleftrightarrow}} t' : T \mathbin{@} l$*,* $L \vdash \Phi \approx \Psi$ *and* $L \mid \Phi \vdash_i \Delta \approx \Gamma$*, then whether* $L \mid \Phi; \Delta \vdash_i t \mathrel{\hat{\Longleftrightarrow}} t' : T \mathbin{@} l$ *is decidable.*

By using the completeness lemma, we obtain the desired decidability proof.

Theorem 5.39 (Decidability of convertibility)**.**

- *If* $L \mid \Psi; \Gamma \vdash_i T \mathbin{@} l$ *and* $L \mid \Psi; \Gamma \vdash_i T' \mathbin{@} l$*, then whether* $L \mid \Psi; \Gamma \vdash_i T \approx T' \mathbin{@} l$ *is decidable.*
- *If* $L \mid \Psi; \Gamma \vdash_i t : T \mathbin{@} l$ *and* $L \mid \Psi; \Gamma \vdash_i t' : T \mathbin{@} l$*, then whether* $L \mid \Psi; \Gamma \vdash_i t \approx t' : T \mathbin{@} l$ *is decidable.*

5.6 Summary

In this chapter, I develop DeLaM, a dependent type theory based on layering and the matryoshka principle. DeLaM supports not only code running, but also recursion on code objects, and therefore I have found a candidate to resolve my original research problem: supporting meta-programming in dependent type theory. In particular, in the examples, I have shown that DeLaM can be used to develop domain-specific solvers like `ac-check` that focuses on the syntax of terms, as well as a "try-all" heuristic like `crush` that analyzes the syntax of types, in a completely type-safe manner. In summary, DeLaM demonstrates a promising path to support dependently typed meta-programming in proof assistants.

Despite that DeLaM has nice properties like normalization and the decidability of convertibility, it still requires work to understand how DeLaM can be applied in practice. More studies are required to understand what other features are needed for practical use, and what is a suitable programming paradigm to write meta-programs in systems like DeLaM. These questions advocate more research in this subject and I would hope to see that one day, dependently typed meta-programming is materialized in proof assistants.

Part III

Discussions and Conclusions

6

Related Work And Discussions

6.1 Modal Type Theories

There is a long history of combining modalities and programming languages. Many researchers have given different kinds of formulations. Prawitz (1965) first proposes a formulation in natural deduction for the modal logic $S4$. However, as pointed out by Bierman and de Paiva (2000); Pfenning and Davies (2001), Prawitz's formulation is not closed under substitutions, so the system is unsound. Indeed, how to support coherent substitutions is the main challenge of designing modal type theories. One fix proposed by Bierman and de Paiva (2000) is to "bolt" substitutions onto the introduction rule:

$$\frac{\forall\ i \in [0, n).\Gamma \vdash s_i : \Box S_i \qquad x_0 : \Box S_0, \cdots, x_{n-1} : \Box S_{n-1} \vdash t : \Box T}{\Gamma \vdash \texttt{box}\ t\ \texttt{with}\ s_0/x_0, \cdots, s_{n-1}/x_{n-1} : \Box T} \qquad \frac{\Gamma \vdash t : \Box T}{\Gamma \vdash \texttt{unbox}\ t : T}$$

Here the introduction rule for $\Box$ maintains a substitution which replaces all modal assumptions that t depends on (i.e. s_i for x_i for all i). The second premise ensures that t only depends on modal assumptions of type $\Box S_i$. This formulation captures the intuition in modal logic $S4$ that “valid facts only depend on valid assumptions”. The elimination rule simply extracts T out of $\Box$. The β rule applies the pre-stored substitutions:

$$\texttt{unbox}\ (\texttt{box}\ t\ \texttt{with}\ s_0/x_0, \cdots, s_{n-1}/x_{n-1}) \rightsquigarrow t[s_0/x_0, \cdots, s_{n-1}/x_{n-1}]$$

Though this formulation is closed under substitutions, it does not seem very practical. For one, having to specify a substitution when introducing $\Box$ seems too restrictive. It would be more helpful if this substitution can be specified during elimination. For comparison, function arguments are provided during function application; functions would not be very useful if arguments must be specified as part of function abstractions.

Therefore, in this thesis, I investigate the Kripke and dual-context styles proposed by Davies and Pfenning (2001); Pfenning and Davies (2001); Pfenning and Wong (1995). The novelty of Davies, Pfenning and Wong’s work is to realize that the modal logic $S4$ corresponds to (compositional) meta-programming by reading the modality $\Box$ as code. Their work not only gives a logical account for meta-programming directly, but also sets a stepping stone for intensional analysis accomplished in this thesis. This series of work by Pfenning and others also includes two different formulations of modal logics, leading to two different styles of meta-programming, the Kripke style and the dual-context style, described in Chapter 1. Davies and Pfenning (2001) give a translation between the syntax of the Kripke-style and dual-context-style $\lambda^{\Box}$, showing that two styles have identical expressive power. This translation, however, is static and does not take equivalence into account. In fact, this translation does not preserve equivalence because $\Box$ in the Kripke style is extensional due to the η rule, while the dual-context style is not. Therefore, in the presence of dependent types where types may include arbitrary computation, it is not straightforward to compare the expressive power of Mint and dependently typed variants of the dual-context-style systems, e.g. crisp type theory by Licata et al. (2018).

In Part I of this thesis, I focus on the Kripke style. In the Kripke style, typing judgments are relative to context stacks. Each context in a stack models a Kripke world in a Kripke universes and the $\Box$ modality models travels among these worlds. The relation among worlds is governed by the range of modal offsets, which control the number of worlds to travel backwards (see Sec. 2.1 and 2.2). Martini and Masini (1996) present a variant of the Kripke style, but their system annotates all terms with a level for these Kripke worlds, so it is too verbose to use. Kripke-style systems are also investigated under the name of the Fitch style. The Fitch style is first motivated by Borghuis (1994), where Borghuis motivates his modal Pure Type Systems (PTS) by the Fitch-style natural deduction. The Fitch-style natural deduction uses sequences of *sub-ordinate proofs* to organize sub-proofs linearly, instead of trees in the Gentzen-style natural deduction. A modal conclusion in the Fitch style only has a restricted access (called reiteration) to modal assumptions. This restricted reiteration structure inspires Borghuis to also employ context stacks (or generalized contexts in Borghuis' terminologies) for typing modal PTS. Interestingly, despite different motivations, the Kripke and Fitch styles[18] eventually arrive at virtually identical systems. Unlike the Kripke-style $\lambda^{\Box}$, where the elimination form of $\Box$ ($\texttt{unbox}_n$) integrates both modal and local weakening, Borghuis' elimination rule has explicit rules for local weakening and modal weakening. As a consequence, weakening is not a property of the overall system in Borghuis' work but is built in the definition. Furthermore, Borghuis studies strong normalization via a translation of the modal PTS to a PTS, whereas I give a direct strong normalization proof for both $\lambda^{\Box}$ and MINT.

Clouston (2018) develops a few other modal λ-calculi in Fitch style, including systems K and idempotent $S4$, and their categorical semantics. In idempotent $S4$, $\Box\Box A$ and $\Box A$ are equivalent. In the formulation, it is the same as using the `unbox` eliminator by dropping modal offsets entirely. Clouston switches to another variant, which uses a special lock symbol 🔒 to separate worlds in a single context. Clearly, this method is ultimately the same as context stacks, though is still arguably inconvenient due to extra side conditions like "delete all of the locks that occur in the context" or "no lock occurs in context Γ". These checks and operations on contexts with locks correspond to modal structural properties in the Kripke style. For example, "deleting all of the locks" corresponds to the property of modal fusion (see Sec. 2.2). The condition that "no locks occur in context Γ" is naturally captured simply by the stack structure of the

[18]In fact, Borghuis did not refer to his formulation as "the Fitch style". This name was first introduced by Clouston (2018) at a much later time. See (Bellin et al., 2001, Sec. 6.1) for a contemporary remark.

context stack in Kripke-style systems. Hence, the context-stack formulation is cleaner and easier to use.

Compared to the dual-context style, the Kripke or Fitch style is challenging because the latter's substitution calculus is not very obvious. Indeed, a significant portion of Part I is devoted to develop K-substitutions. Due to this challenge, direct normalization proofs for even simply typed $S4$ and its sub-systems are only given very recently (Valliappan et al., 2022; Hu and Pientka, 2022a). Valliappan et al. (2022) focus on the lock-based formulation and give a formulation for non-idempotent $S4$, where $\square\square A$ and $\square A$ are not equivalent and the eliminator $\texttt{unbox}_n$ requires modal offsets. Unlike the Kripke style, where all four sub-systems of $S4$ only differ by the range of modal offsets and truncoids successfully capture the Kripke structure of $\square$ in both syntax and semantics, Valliappan et al. (2022) have to give different formulations for the four sub-systems, leading to four similar but distinct normalization proofs. In (Hu and Pientka, 2022a), I give a categorical strong normalization proof for all four sub-systems of $S4$ and describe contextual types in the Kripke style. A contextual type in the Kripke style is relative to a context stack. Murase et al. (2023) present a Fitch-style $S4$ with contextual types only relative to contexts and context polymorphism.

Gratzer et al. (2019) study a dependently typed idempotent $S4$. Their normalization proof follows Abel (2013) as Part I. Their approach is different from mine in an extra parameter of poset to model the Kripke structure, so their subsequent proofs must be aware of this poset. In my presentation, truncoids abstract away the exact Kripke structure, so one normalization proof can be instantiated to apply for all four sub-systems of $S4$. A dependently typed variant of the system K and its categorical characterization is given by Birkedal et al. (2020).

In Part II, I switch my focus to the layered style, a variant of the dual-context style, to support intensional analysis. The dual-context style is much easier to understand and extend due to its quite obvious substitution calculus: a pair of global and local substitutions, where two kinds of substitutions respect distributivity (see Lemma 4.8). Kavvos (2020) looks into the dual-context style and gives formulations for sub-systems of $S4$, including K, T and $K4$. These sub-systems in the dual-context style have different introduction rules for $\square$ and variable rules from $S4$, so the dual-context style has a disadvantage of less modularity in the typing rules compared to the Kripke style.

Kavvos further gives their strong normalization proofs using the classic Tait's computability (Tait, 1967) and their categorical semantics.

Shulman (2018) gives a dependently typed dual-context-style system, spatial type theory. In addition to the $\square$ modality, he introduces other modalities to relate topology and homotopy. Licata et al. (2018) restrict spatial type theory to only the $\square$ modality and introduce crisp type theory. This type theory is motivated to obtain an internal models of universes in homotopy type theory (Program, 2013).

Recently, the dual-context style is generalized to capture multiple modalities by Gratzer et al. (2020); Gratzer (2022). The result is a dependently typed parametric type theory, multimodal type theory (MTT)[19]. MTT is parameterized by a 2-category[20]. This 2-category not only models the modalities, but also describes how the modalities interact. Thus, it could be possible to use MTT to model layers D and M of DeLaM. Layers V and C, contextual types and the recursors on code, however, heavily rely on a fixed interpretation (i.e. viewing terms at both layers as static pieces of code), it is not clear how full DeLaM can be fit into MTT's framework.

Another candidate that might subsume layered modal type theory and DeLaM is adjoint logic (Jang et al., 2024b). Adjoint logic is a logical system which simultaneously contains multiple logical systems. A logical system might access truths from other logical systems using pairs of adjoint modalities. Jang et al. (2024b) show that adjoint logic can be used to model many programming styles, including memory management and meta-programming. As of now, this adjoint style of logic has not been fully scaled to dependent type theory, so how it concretely models the layered-style systems remains future work.

In the layered systems, I use layers to account for the number of nested $\square$'s, which shares some similarities with graded and quantitative systems (Atkey, 2018; Abel and Bernardy, 2020; Moon et al., 2021). The latter systems use grades to keep track of uses of variables. It would be interesting to have a universal framework to contain all these

[19]MTT also uses the 🔒 symbol to manage contexts, which I find somewhat misleading, as it is unrelated to the same symbol used in the Fitch style.

[20]For readers who are not familiar with the concept, a 1-category is a category in the classical sense. A 2-category can be viewed as a 2-level, higher algebraic structure, where the equivalence on the first level possesses a non-trivial 1-categorical structure. In a regular 1-category, this higher categorical structure is simply not considered.

different uses of modalities, though it requires further investigations.

The layered systems distinguish computational behaviors at different layers and uses the static code lemma to support pattern matching or recursors on code. This approach is similar to GuTT (Gratzer and Birkedal, 2022), a guarded type theory supporting Löb induction. GuTT has two layers. The first layer excludes dynamics of Löb induction (but not for other terms) and enjoys normalization. The lost dynamics is recovered at the second layer, at the cost of normalization. GuTT and the layered systems are similar in that they both take advantage of differences between layers and one layer is the extension of the other.

6.2 Normalization for Type Theories

The normalization of natural deduction systems can be dated back to Gentzen (1935)'s PhD thesis, where he proves the consistency of propositional and first order logic. Gentzen's consistency proof constructs an intermediate system called the sequent calculus and uses the process of cut elimination, which effectively corresponds to a substitution lemma and proves that the cut rule in the sequent calculus is redundant. Consistency is concluded by showing the equivalence between natural deduction and the cut-free sequent calculus. Reading his proof computationally, the proof in fact constitutes a strong normalization algorithm which computes the $\beta\eta$ normal form of a term.

As opposed to Gentzen's purely syntactic approach, in the context of type theory, people tend to use semantic approaches. One of the most classical approaches is Tait's computability, or logical relations, or reducibility candidates (Tait, 1967; Girard, 1989). The original problem which Tait solves is the strong normalization of STLC with β reduction. A proof following Tait's steps typically proceeds as follows: first, establish the Church-Rosser property (Barendregt, 1985), which states that syntactically equivalent terms eventually reduce to the same term (the common reduct). Then prove progress and preservation (Wright and Felleisen, 1994) to show that this common reduct has the same type as the original terms. Strong normalization is then proved by characterizing strongly normalizing terms of each type constructor via logical relations. This process is standard in much work prior to 2000 (Luo, 1990; Coquand and Gallier, 1990; Girard, 1972; Coquand, 1985, etc.). However, this process becomes very verbose for larger systems and sometimes even too tedious, so many researchers have looked into other ways to establish normalization.

In Part I, I presented normalization by evaluation (NbE). NbE is originally proposed by Martin-Löf (1975); Berger and Schwichtenberg (1991). Instead of term reductions, NbE employs some mathematical domain, in which computation is performed, and then extracts normal forms from this domain. This method is more convenient as there is no longer need to directly establish syntactic properties like Church-Rosser and subject reduction. Instead, these properties are (implicitly) inherited from the chosen mathematical domain, so the proof is much more light-weight and its size is much smaller for complex systems than Tait's original approach. Other than NbE based on untyped domain models, which is the method used in Part I and is pioneered by Abel (2013), another frequent approach is based on category theory. Altenkirch et al. (1995) first give an NbE proof for STLC based on a presheaf model. In this setting, a presheaf model is in fact a Kripke model, where the base category of weakenings of contexts corresponds to the Kripke relation formed by weakenings. The categorical lingo helps to organize thoughts and foresee necessary intermediate lemmas. NbE based on presheaf models is a frequent method in my published work (Hu and Pientka, 2022a, 2024b). Altenkirch and Kaposi (2016a) further scale the presheaf model to dependent types. Their model is effectively an instance of categories with families (CwFs) (Dybjer, 1995), a categorical formulation for dependent types. Cubric et al. (1998) describe a different categorical NbE proof, based on a category theory enriched by PERs. This method is not very easy to extend because it requires the development of a whole new variant of category theory with PERs.

In Part II, I switched to another variant of Tait's computability method, based on (Abel et al., 2018), which only proves weak normalization. The advantage, however, is to avoid proving Church-Rosser, which entails many technical setups. The remedy for weak normalization is a type-directed conversion checking algorithm. Though this method induces a significant larger proof size compared to NbE proofs, many researchers (Pientka et al., 2019; Pujet and Tabareau, 2022, 2023; Adjedj et al., 2024, etc.) including myself still find this method useful because of its clear, mechanized reference in Agda. I will discuss this method in more details in the next section about mechanization.

Another improvement of the classical Tait's method is Sterling (2022)'s synthetic Tait's computability method. This method connects type theory with topos theory and resolves difficult problems in type theory, like normalization, using properties in topos theory. This new method notably proves normalization of cubical type theory (Cohen et al., 2015) and normalization of MTT (Gratzer, 2022). Nevertheless, I personally find that this method at the current stage requires a very high bar for a mathematical background for general computer scientists. I am hoping that future simplifications will make this method more accessible and hence more frequently used in the research community.

6.3 Mechanization of Normalization for Type Theories

From the 1990's the question of how to mechanize the normalization proof for dependent type theory has been fundamental to gain trust in the type-theoretic foundation which proof assistants such as Coq, Agda and Lean are built on. One of the earliest works is by Barras and Werner (1997). They formalize strong normalization for the calculus of construction (CoC) in Coq using logical relations. More recently, Abel et al. (2018) mechanize a normalization proof for Martin-Löf logical framework in Agda and is what Part II is based on. Pujet and Tabareau (2022) extend this work by mechanizing observational equality (Altenkirch et al., 2007) and a two-level cumulative universe hierarchy. In addition to the level 0 and level 1 universes, they have an extra level ∞ universe, which subsumes both level 0 and 1 universes. Then the proof can simply lift all types to the ∞ level and avoid explicit discussions of universe levels entirely. This treatment resembles the typical paper proof. This work is further superseded by Pujet and Tabareau (2023), which mechanizes impredicative observational equality. Cumulativity of universes, however, is removed from the universe hierarchy. Adjedj et al. (2024) redesign and port (Abel et al., 2018) to Coq. This work also takes advantage of Coq's excellent extraction mechanism to extract a certified executable for type-checker.

In contrast, mechanizations of NbE algorithms in dependent type theory are less common. Danielsson (2006) presents the first mechanization of NbE for Martin-Löf logical framework using induction-recursion in AgdaLight. As pointed out by Chapman

(2008), Danielsson (2006)'s formulation contains non-strictly positive predicates, which compromise the trust in this work.

Chapman (2008) formalizes Martin-Löf logical framework in the style of CwFs in Agda and presents a sound normalizer. However, the normalizer is not shown complete whereas normalization algorithms in Part I are shown complete and sound.

Altenkirch and Kaposi (2016a,b, 2017) mechanize an NbE algorithm for Martin-Löf logical framework in Agda and prove completeness and soundness using a presheaf formulation akin to CwFs. Their development explores an advanced combination of intrinsic syntactic representations and involved features like induction-induction (Forsberg and Setzer, 2010) and quotient inductive types. In comparison, this thesis only relies on two standard extensions: induction-recursion and functional extensionality. The simplicity leads to a mechanization of a full hierarchy of universes in Agda in Chapter 3 and a (slow) certified executable after extraction. This mechanization is the only one for untyped domain models in Agda to my knowledge.

Most mechanizations of NbE are done in Agda, as it supports induction-recursion, which strengthens the logical power of the meta-language to define the semantics for universes. Nevertheless, attempts are also made in Coq. Wieczorek and Biernacki (2018) mechanize an NbE algorithm à la Abel (2013) for Martin-Löf logical framework in Coq. Since Coq does not support induction-recursion, they universally quantify over the impredicative universe `Prop` in their models. Their algorithm can also be extracted to and run in Haskell or OCaml. One benefit of using Coq is that `Prop` is automatically removed during extraction. Hence, their extraction code is cleaner than the one generated from Agda. Recently, Jang et al. (2024a) are working on the same NbE method and developing a fully certified type-checker for MLTT in Coq. This work not only re-defines the PER model using impredicativity in Coq to accommodate an infinite cumulative universe hierarchy, but also is designed for future extensions, e.g. a mechanization of normalization for Cocon (Pientka et al., 2019).

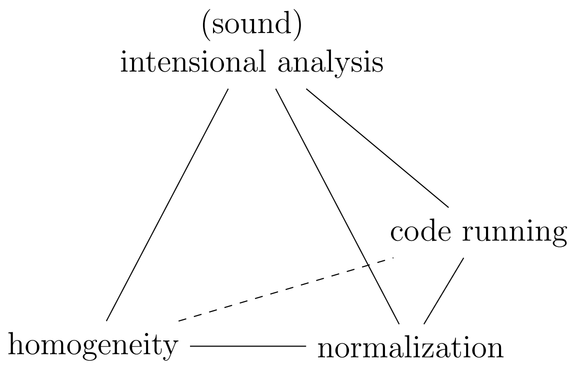


Figure 6.1: Impossible tetrahedron for meta-programming foundations

6.4 Modalities, Meta-programming and Intensional Analysis

Early ideas of meta-programming using quasi-quoting style can be traced back to Lisp/Scheme (Abelson and Sussman, 1996). In Lisp's untyped setting, all programs are represented as lists, so intensional analysis is reduced to inspections of lists and is relatively simple. Supporting type-safe meta-programming leads to all sorts of complications. MetaML (Taha and Sheard, 2000) is an early example for type-safe meta-programming. MetaML employs a quasi-quoting style similar to Lisp. However, MetaML does not support any form of intensional analysis. In fact, in order to enable compile-time optimization of generated code, MetML deliberately avoids intensional analysis. Though quasi-quoting has a long history in meta-programming and is modeled by the Kripke style in typed settings, as described at the beginning of Chapter 4, it does not seem compatible with intensional analysis. On the other hand, though the dual-context style forces programmers to write meta-programs in a comonadic style, it has a better setup for intensional analysis, so I developed the layered style based on the dual-context style in Part II. The dual-context style is also the approach taken in Beluga (Pientka, 2008; Pientka and Dunfield, 2008) and Moebius (Jang et al., 2022).

Boespflug and Pientka (2011) extend the dual-context style to the multi-context style. Though the multi-context style and the Kripke style both use multiple contexts for typing, the number of contexts in the former is fixed (hence context arrays), while in the latter, contexts are often pushed and popped during typing (hence context stacks). Moebius (Jang et al., 2022) combines the multi-context style and contextual types, and supports pattern matching on code in System F. Moebius has subject reduction. However, to adapt Moebius to a type theory, normalization must be proved, but it is not obvious how to support coverage. Whether layering provides a solution requires a future investigation.

Cocon (Pientka et al., 2019) is a 2-layered meta-programming type theory and is similar to DELAM in many ways. At the lower layer in Cocon is a logical framework (LF) (Harper et al., 1993) to define object languages. A term in an object language in LF can be analyzed and recursed on in MLTT at the higher layer. Though code running is not a built-in facility in Cocon, it might be defined as a recursive function in MLTT if the object language can be soundly embedded into MLTT. Semantically, Cocon is similar to DELAM in the 2-layered structure. Both systems need one model for each layer. However, due to the flexibility of LF, the model for the lower layer in Cocon does not and cannot have any specific relation to the one for MLTT at the higher layer, while in DELAM, models at different layers are related by the layering restriction lemma. It is this lemma which enables code running universally in DELAM. A categorical semantics for Cocon is given by Pientka and Schöpp (2020); Hu et al. (2022). Kovács (2022); Allais (2024) define 2-level type theory (2LTT), which is similar to Cocon, but it focuses more on using dependent type theory to compose code of practical languages and does not support intensional analysis.

In Fig. 6.1, I summarize different features of meta-programming systems and put them into an impossible tetrahedron. The idea is that in this tetrahedron, it is possible to find a type theory supporting each surface (hence trivially each edge), but as of now, no system achieves all four vertices of the tetrahedron. I set one of the vertices to be "homogeneity", in the hope that future investigations could break this tetrahedron and find a more uniform way to design a type theory that combines code running and intensional analysis than layering. On the surface of intensional analysis, normalization and code running, are clearly the layered systems in Part II. On the surface of homogeneity, normalization and intensional analysis is Cocon, which does not guarantee code running for all code. On the surface of homogeneity, normalization and code running is (contextual) $\lambda^{\Box}$ and its dependently typed variant, where only code composition and code running are possible. On the surface of homogeneity, intensional analysis and code running lies Moebius, for example, which does not have normalization because it needs a way to ensure coverage for pattern matching. Breaking this tetrahedron effectively implies a new method to support meta-programming in type theories.

There are other dependently typed meta-programming systems using modalities to quantify code. Kawata and Igarashi (2019) study λ^{MD}, a logical framework with stages. λ^{MD} is similar to MINT in that it also employs quasi-quoting and does not support intensional analysis. One difference with MINT is that λ^{MD} uses stage variables to keep track of stages, while MINT use context stacks and `unbox` levels for the same purpose. Brady and Hammond (2006) improve Pasalic et al. (2002)'s Meta-D by extending Martin-Löf type theory with stages, similar to MINT. Fundamentally, the type theory by Brady and Hammond (2006) is the dependently typed system T with cross-stage persistence. Extending the T variant of MINT with cross-stage persistence would constitute Brady and Hammond (2006)' system. Though the authors claim that their type theory is strongly normalizing, they do not provide any proof, whereas MINT's normalization proof naturally adapts to all subsystems of $S4$, including T.

Ωmega (Viera and Pardo, 2006) is a sound, simply typed meta-programming system with pattern matching on code. Ωmega implements the quasi-quoting style. The open context of a code is annotated in the type, similar to contextual types. However, the type of the code itself is not remembered, so their type system is not as complex due to reduced type information.

6.5 Future Work

In this section, I discuss some future work for DELAM.

6.5.1 Russell-Style Universes in DeLaM

DELAM employs a Tarski-style universe hierarchy, where types and terms belong to different grammars. This separation is introduced purely due to technical considerations. The Tarski style is closer to the semantics and hence advantageous in formulating the recursion principles for code in DELAM both in syntax and in semantics, as in the semantic judgments for code. However, the Tarski style is not often used in proof assistants, where the Russell style is the common practice. Therefore, an important future work to make DeLaM more practical is to derive its Russell-style variant. Syntactically, this implies that the recursion principle for code in this variant is no longer mutual, because there is only one grammar for both types and terms. In retrospect, I think that the Russell style should be fairly straightforward to work out. Here, I list a few important adjustments. Note that the universes are still non-cumulative; cumulativity induces subtyping, which is an orthogonal complication that I would like to avoid thinking about at this moment. First, there is only one kind of contextual types in the syntax: $\square(\Gamma \vdash_{\text{C}} T \,@\, l)$, and the code of types simply has type $\square(\Gamma \vdash_{\text{C}} \mathtt{Ty}_l \,@\, 1 + l)$. Correspondingly, there is only one elimination principle for code:

$$\mathsf{elim}^l\ (\ell, g, u_T, x_t.M)\ \overrightarrow{b}\ (t : \square(\Gamma \vdash_{\text{C}} T \,@\, l'))$$

Since the recursion principle is no longer mutual, it only requires one motive M on universe level l for the return type of the recursion. In this case, this whole recursion has type $M[l'/\ell, \Gamma/g, T/u_T, t/x_t]$. The recursor still requires a list of branches $\overrightarrow{b}$, but there are fewer of branches than the Tarski style, as there is no distinction between types and terms anymore. On the surface, the recursion still occurs for sub-structures, so termination remains quite natural. To model the semantics for code, the steps taken in Sec. 5.4.4 should still be correct in principle. A semantic judgment for MLTT is needed to capture the running information at layers D and M, and this judgment is embedded in another inductively defined judgment for code of MLTT to recursively remember the syntactic shapes of the terms. Whether the adjustments outlined here will work, however, is left for the future.

6.5.2 NbE for DeLaM

An NbE algorithm based on an untyped domain model has many advantages. It not only gives a strong normalization algorithm, hence also a trivial conversion checking algorithm, but also is very easy to mechanize and implement. The proofs are also much less complex than the current method based on reductions and an explicit conversion checking algorithm. Therefore, developing an NbE algorithm with an untyped domain model is a valuable option.

To develop an NbE algorithm, it is convenient to first convert DeLaM into a version with explicit substitutions. The purpose of introducing explicit substitutions is to delay the action of substitutions to evaluation, so that proofs about the PER model are simplified. This part should be achievable by following Abel and Pientka (2010).

The main difficulty of giving an NbE lies in the definition of the domain model. More specifically, the domain model must capture the syntactic aspects of code, so the part of the domain model for code is necessarily isomorphic to the syntax of DeLaM. What should this part be? I can think of two possible options, each with its own advantages and disadvantages.

The first option is that this part of the domain model is just the syntax of DeLaM, i.e. the syntax of DeLaM is a subset of the domain model. This option is quite convenient, especially when giving the semantics for the recursors on code, as the recursors directly act on code itself. This option is in fact taken in the categorical normalization proof in the published work for layered modal simple type theory (Hu and Pientka, 2024b). In the settings of an untyped domain model, however, this option is not entirely beneficial. With dependent types, the well-formedness of code is relative to a global context, so this option would already require the PER model to respect global weakenings among global contexts, adding an extra Kripke structure to the PER model. This addition is not very desirable because the advantage of an untyped domain model is to only focus on the *runtime* of the evaluation process, so the PER model itself does not need to respect any weakening originally. Bringing in a Kripke structure to the PER model seems to have defeated the purpose.

Another possible option is to introduce a representation of syntax in the untyped domain model, where global variables, similar to local variables, are represented by *de Bruijn levels*. This option recovers the PER model from the problem in the previous option by avoiding the addition of a Kripke structure. However, the downside is that the relation between the actual syntax and the domain representation of syntax is no longer identity. This representation must simulate global substitutions, which have been defined in the syntax, and relate its action on the recursors on code with syntax. To handle lifting, the evaluation process must also evaluate this domain representation, in addition to syntax, to other domain values.

Some comparisons between both options seem to suggest that the second option is slightly better. Whether this intuition is the case requires more careful investigations.

6.5.3 Mechanization of DeLaM

Due to time limitation, I leave the mechanization of DeLaM for future work. Although the mechanization in principle should just follow this thesis and the technical report (Hu and Pientka, 2024a) closely, there are a number of difficulties which might have been over-simplified for the purpose of paper presentation.

One immediate problem is universe levels and their equivalence. Throughout this thesis, I intentionally conflate the equivalence of universe levels with their equality, and omit the fact that the decision of equivalence requires an algorithm. Though this algorithm is relatively simple, in an actual mechanization, related details must be explicitly spelled out and therefore introduce more noise than the paper presentation.

A more challenging problem caused by universe polymorphism is universe variables and their well-foundedness. To model the universe hierarchy, the logical relations must be defined by recursion on universe levels. In this thesis, this is expressed by a transfinite recursion due to the ω level, which is above all countable levels. In mechanization, how this transfinite structure should be handled remains unclear at this point.

Another complication is the parametricity of the logical relations. In the thesis, the logical relations are defined parametrically. In particular, the logical relations for types and terms overlap in cases for MLTT. This parametricity is for conciseness, but also introduces conceptual connections between layers D and M and reveals the connection established by the layering restriction lemma. Proofs for overlapping cases are also argued altogether, so the size of the paper proof might be halved due to the parametricity. However, this parametricity might not be easily mechanized. The major difficulty is that the logical relations for layer M require too many complex intermediate definitions after defining the logical relations for layer D. In particular, to model static code, the semantic judgment for code is inductively defined, which itself refers to the logical relations for terms and substitutions at layer D. Though this difficulty does not ultimately impede the mechanization, it is likely to blow up the scale of the mechanization by requiring duplications of all necessary lemmas at both layers D and M. Given the size of the proofs of normalization and decidable convertibility is already quite large on paper, a mechanized proof could be very difficult to manage, and as more progress is made, the type-checking time of the project could become less and less bearable.

Finally, a complete development of an NbE algorithm described in Sec. 6.5.2 is likely to reduce the workload of the mechanization. Therefore, it is more viable to first work on NbE on paper first, and then transcribe the paper proof in a proof assistant.

6.5.4 Other Extensions for DeLaM

The DeLaM presented in Chapter 5 is minimal in that more facilities are needed in order to write more potentially useful meta-programs / tactics. In fact, a few extensions are quite straightforward by looking at the semantics. For example, it should be possible to obtain code for local contexts and recurse on the structure of local contexts. To enable these features, DeLaM can be extended with $\square\mathsf{Ctx}$, which indicates the static code of a local context. Since the semantics of $\square\mathsf{Ctx}$ live at layer $\textsc{d}$ as modeled by the logical relations, this type can live on the universe level 0 at layer $\textsc{m}$.

Similarly, it should be possible to meta-program and recurse on the code of a local substitution, following Pientka (2008). For a local context Δ, a contextual type for a local substitution is $\square(\Gamma \vdash_{\textsc{c}} \Delta)$. Again, since the semantics of local substitutions already has been given in the logical relations at layer $\textsc{d}$, $\square(\Gamma \vdash_{\textsc{c}} \Delta)$ can also live in the universe level 0 at layer $\textsc{m}$.

In principle, many types and operations on objects from MLTT can be added to layer $\textsc{m}$ in DeLaM to provide more practical facilities for meta-programming.

6.6 An Outline of Implementing DeLaM

Implementing DeLaM in a proof assistant is an obvious and very interesting future work. It will allows us to understand what features should be included in the type-theoretic foundation to make type-safe meta-programming practical. Nevertheless, due to time limitations, I do not intend to tackle this problem in this thesis, so I will leave an implementation as a future work. However, I would still like to put down my visions and thoughts on this implementation. As we all know, there is a large gap between how a type theory is defined on paper and how it is actually implemented. DeLaM is not a very conventional type theory, in that it can be implemented in many different styles, depending on the angles which the implementors take. In this section, I only discuss my angle and leave other angles to the imagination of the readers.

The implementation that I envision is based on the observation that there are essentially two (related) dependently typed systems in DeLaM, induced by layers C and M. Both systems should be extendable. The idea is then to separate the implementation into two modes, mode C (default) and mode M. Mode M is what one would probably expect: it implements layer M of DeLaM, and allows users to write meta-programs. Meta-programs can pattern matching on any MLTT definitions that are in the current scope. Mode M should also be able to use all definitions from its imports, both from mode M and from mode C. Hence, the examples in Sec. 5.1 can be written in this mode. Necessary normalization occurs in order to do type-checking.

Mode C, on the other hand, only extends MLTT. In this mode, the exact syntax of MLTT definitions is preserved, so these definitions could participate in meta-programs in mode M. Mode C is special, in that users could still import modules from mode M and refer to definitions and meta-programs from mode M in mode C. Before being type-checked, definitions in mode C are first passed to a code generation phase. If a definition refers to any meta-program, then necessary β reduction is performed by the code generator to reduce away all meta-programs, so that the eventual term fits in MLTT. It is crucial that the code generator does not do more reduction than necessary, because if users use meta-programs to define a function in MLTT, usually they would want to keep the exact generated syntax. This exact syntax can be subsequently pattern matched on in mode M. To give a concrete example, `ac-check`, `search` and `crush` in Sec. 5.1 are put in module `A` in mode M, because they are meta-programs. Meanwhile, `lem` and `lem2` can be put in a different module `B` in mode C, which imports module `A`. Before type-checking `lem` and `lem2`, the code generator first reduces the meta-programs. The code generator first attempts to reduce the calls to `ac-check` and `crush` to normal forms, and continues to reduce away `letbox`. Eventually, meta-programs in both lemmas reduce to some valid MLTT syntax, so their definitions type-check in MLTT at layer C. In this way, mode C strictly correspond to MLTT. However, this code generation phase might fail even if a term has a pure MLTT type. In this case, this term includes meta-programs that cannot be fully reduced away. The following is one such program:

```
foo : Nat → Nat
foo x = letbox u ← (if eq? x zero then box zero else box zero) in u
```

In this program, I artificially create a branch based on the value of the input `x`. Both branches return `box zero` so this comparison is essentially bogus, but it does prevent the `letbox` from being fully reduced. Therefore, `foo` cannot be type-checked in mode C.

This way of implementing DeLaM has multiple advantages. First, it ensures the extendability of both modes, so users may choose to program in MLTT or meta-program in DeLaM. The explicit separation between modes is helpful for users to keep track of the mode that they are working in. This organization also gives freedom of trusted bases to the users. They can directly choose DeLaM as the trusted base. In this case, they simply do everything in mode M. For users who would like to keep a minimal trusted base, they can choose to work in mode C most of time, and only use the meta-programming facilities in mode M for convenience. In this case, the type-checker makes sure that all meta-programs are eventually reduced away, so only pure MLTT terms are admitted. Many existing proof assistants could be extended based on this organization. The actual time to extend an existing proof assistant, e.g. Agda, is difficult to establish with today's knowledge, so I leave it as future work.

7

Conclusions

In this thesis, I investigate various modal type theories with applications in meta-programming. In all type theories in this thesis, I focus on the necessity modality $\square$, which describes types of code under Curry-Howard correspondence. This reading provides a convenient logical foundation, which serves as a good starting point to design type theories. In Part I, I first look into the Kripke-style modal type theories. The Kripke-style systems faithfully model the familiar quasi-quoting style for meta-programming. In the Kripke style, type theories maintain a stack of contexts, and we can use the constructor `box` and the eliminator `unbox` to push and pop this stack. The Kripke style also has an additional advantage of modeling all four sub-systems of $S4$ at once, by simply tweaking the range of modal offsets. In Part I, I first give a modular strong normalization proof for $\lambda^{\square}$, a simply typed Kripke-style modal type theory. Instantiating this proof proves strong normalization of all four sub-systems of $S4$. I then scale $\lambda^{\square}$ to dependent types, introducing MINT, a modal intuitionistic type theory. The strong normalization proof scales naturally to MINT.

Though MINT readily serves as a good program logic for meta-programming systems like MetaML where users can compose and execute code, it is not very clear how Mint can support intensional analysis, in particular a general recursion principle on the structure of code. To tackle this problem, in Part II, I investigate the layered systems, which are a modified style from the dual-context style. In a layered modal type theory, sub-languages form a hierarchy, where a sub-language from an inner layer is subsumed by one from an outer layer. This characteristic is called the matryoshka principle, which is the fundamental philosophy behind the layered style. Following the footsteps in Part I, I first design an (almost) simply typed layered modal type theory and prove its weak normalization and decidability of convertibility. Finally, I scale this design to dependent types, introducing DeLaM, a dependent layered modal type theory. DeLaM not only supports quotation, composition and execution of MLTT code, but also supports a general recursion principle on the structure of code. Hence, DeLaM achieves the research objective for this part. As future work, I have also outlined a few places in DeLaM to make potential improvements, as well as a number of key points when implementing it.

I believe that the research results in this thesis contribute to a bigger picture of combining type theory and meta-programming and can stimulate more research towards this direction.

Bibliography

Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. 1991. Explicit Substitutions. *J. Funct. Program.* 1, 4 (1991), 375–416. `https://doi.org/10.1017/S0956796800000186`

Andreas Abel. 2013. *Normalization by Evaluation: Dependent Types and Impredicativity.* Habilitation Thesis. Ludwig-Maximilians-Universität München, Munich, Germany. `https://www.cse.chalmers.se/~abela/habil.pdf`

Andreas Abel and Jean-Philippe Bernardy. 2020. A Unified View of Modalities in Type Systems. *Proc. ACM Program. Lang.* 4, ICFP (2020), 90:1–90:28. `https://doi.org/10.1145/3408972`

Andreas Abel, Joakim Öhman, and Andrea Vezzosi. 2018. Decidability of Conversion for Type Theory in Type Theory. *Proc. ACM Program. Lang.* 2, POPL (2018), 23:1–23:29. `https://doi.org/10.1145/3158111`

Andreas Abel and Brigitte Pientka. 2010. Explicit Substitutions for Contextual Type Theory. In *Proceedings 5th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2010, Edinburgh, UK, July 14, 2010 (EPTCS, Vol. 34)*, Karl Crary and Marino Miculan (Eds.). 5–20. `https://doi.org/10.4204/EPTCS.34.3`

Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. 2017. Normalization by Evaluation for Sized Dependent Types. *Proc. ACM Program. Lang.* 1, ICFP (2017), 33:1–33:30. `https://doi.org/10.1145/3110277`

Harold Abelson and Gerald J. Sussman. 1996. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press.

Arthur Adjedj, Meven Lennon-Bertrand, Kenji Maillard, Pierre-Marie Pédrot, and Loïc Pujet. 2024. Martin-Löf à la Coq. In *Proceedings of the 13th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2024, London, UK, January 15-16, 2024*, Amin Timany, Dmitriy Traytel, Brigitte Pientka, and Sandrine Blazy (Eds.). ACM, 230–245. `https://doi.org/10.1145/3636501.3636951`

Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. 2003. A Functional Correspondence between Evaluators and Abstract Machines. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2003, Uppsala, Sweden, August 27-29, 2003*. ACM, 8–19. `https://doi.org/10.1145/888251.888254`

Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. 2015. Univalent Categories and the Rezk Completion. *Math. Struct. Comput. Sci.* 25, 5 (2015), 1010–1039. `https://doi.org/10.1017/S0960129514000486`

Guillaume Allais. 2024. Scoped and Typed Staging by Evaluation. In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2024, London, UK, January 16, 2024*, Gabriele Keller and Meng Wang (Eds.). ACM, 83–93. `https://doi.org/10.1145/3635800.3636964`

Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical Reconstruction of a Reduction Free Normalization Proof. In *Proceedings of the 6th International Conference on Category Theory and Computer Science, CTCS 1995, Cambridge, UK, August 7-11, 1995 (Lecture Notes in Computer Science, Vol. 953)*, David H. Pitt, David E. Rydeheard, and Peter T. Johnstone (Eds.). Springer, 182–199. `https://doi.org/10.1007/3-540-60164-3_27`

Thorsten Altenkirch and Ambrus Kaposi. 2016a. Normalisation by Evaluation for Dependent Types. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, Porto, Portugal, June 22-26, 2016 (LIPIcs,* *Vol. 52)*, Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:16. `https://doi.org/10.4230/LIPICS.FSCD.2016.6`

Thorsten Altenkirch and Ambrus Kaposi. 2016b. Type Theory in Type Theory Using Quotient Inductive Types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, Florida, USA, January 20-22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 18–29. `https://doi.org/10.1145/2837614.2837638`

Thorsten Altenkirch and Ambrus Kaposi. 2017. Normalisation by Evaluation for Type Theory, in Type Theory. *Log. Methods Comput. Sci.* 13, 4 (2017). `https://doi.org/10.23638/LMCS-13(4:1)2017`

Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 2007. Observational Equality, Now!. In *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, Aaron Stump and Hongwei Xi (Eds.). ACM, 57–68. `https://doi.org/10.1145/1292597.1292608`

Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *Proceedings of the 9th International Conference on Interactive Theorem Proving, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018 (Lecture Notes in Computer Science, Vol. 10895)*, Jeremy Avigad and Assia Mahboubi (Eds.). Springer, 20–39. `https://doi.org/10.1007/978-3-319-94821-8_2`

Kenneth Appel and Wolfgang Haken. 1977. The Solution of the Four-Color-Map Problem. *Scientific American* 237, 4 (1977), 108–121. `https://doi.org/10.1038/scientificamerican1077-108`

Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*

2018, Oxford, UK, July 09-12, 2018, Anuj Dawar and Erich Grädel (Eds.). ACM, 56–65. `https://doi.org/10.1145/3209108.3209189`

Hendrik Pieter Barendregt. 1985. *The Lambda Calculus - Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics, Vol. 103. North-Holland.

Bruno Barras and Benjamin Werner. 1997. Coq in Coq. `https://www.lix.polytechnique.fr/Labo/Bruno.Barras/publi/coqincoq.pdf` Unpublished manuscript.

Gianluigi Bellin, Valeria C. V. de Paiva, and Eike Ritter. 2001. Extended Curry-Howard Correspondence for A Basic Constructive Modal Logic. In *Proceedings of Methods for Modalities.*

Ulrich Berger and Helmut Schwichtenberg. 1991. An Inverse of the Evaluation Functional for Typed Lambda-calculus. In *Proceedings of the 6th Annual Symposium on Logic in Computer Science, LICS 1991, Amsterdam, the Netherlands, July 15-18, 1991.* IEEE Computer Society, 203–211. `https://doi.org/10.1109/LICS.1991.151645`

Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. 2022. Type Theory with Explicit Universe Polymorphism. In *28th International Conference on Types for Proofs and Programs, TYPES 2022, LS2N, University of Nantes, France, June 20-25, 2022 (LIPIcs, Vol. 269)*, Delia Kesner and Pierre-Marie Pédrot (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:16. `https://doi.org/10.4230/LIPICS.TYPES.2022.13`

Gavin M. Bierman and Valeria de Paiva. 2000. On An Intuitionistic Modal Logic. *Stud Logica* 65, 3 (2000), 383–416. `https://doi.org/10.1023/A:1005291931660`

Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2020. Modal Dependent Type Theory and Dependent Right Adjoints. *Math. Struct. Comput. Sci.* 30, 2 (2020), 118–138. `https://doi.org/10.1017/S0960129519000197`

Mathieu Boespflug and Brigitte Pientka. 2011. Multi-level Contextual Type Theory. In *Proceedings of the 6th International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2011, Nijmegen, the Netherlands, August 26, 2011 (EPTCS, Vol. 71)*, Herman Geuvers and Gopalan Nadathur (Eds.). 29–43. `https://doi.org/10.4204/EPTCS.71.3`

V. A. J. Borghuis. 1994. *Coming to Terms with Modal Logic : on the Interpretation of Modalities in Typed Lambda-calculus.* PhD Thesis. Technische Universiteit Eindhoven, Eindhoven, the Netherlands. `https://doi.org/10.6100/IR427575`

Edwin C. Brady and Kevin Hammond. 2006. A Verified Staged Interpreter is A Verified Compiler. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006*, Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen (Eds.). ACM, 111–120. `https://doi.org/10.1145/1173706.1173724`

William Burr. 2020. False Warnings of Soviet Missile Attacks Put U.S. Forces on Alert in 1979-1980. `https://nsarchive.gwu.edu/briefing-book/nuclear-vault/2020-03-16/false-warnings-soviet-missile-attacks-during-1979-80-led-alert-actions-us-strategic-forces`

Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *J. Autom. Reason.* 61, 1-4 (2018), 367–422. `https://doi.org/10.1007/S10817-018-9457-5`

Andrew Cave and Brigitte Pientka. 2012. Programming with Binders and Indexed Datatypes. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 413–424. `https://doi.org/10.1145/2103656.2103705`

James Chapman. 2008. Type Theory Should Eat Itself. In *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice, LFMTP@LICS 2008, Pittsburgh, Pennsylvania, USA, June 23, 2008 (Electronic*

Notes in Theoretical Computer Science, Vol. 228), Andreas Abel and Christian Urban (Eds.). Elsevier, 21–36. `https://doi.org/10.1016/J.ENTCS.2008.12.114`

David R. Christiansen and Edwin C. Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 284–297. `https://doi.org/10.1145/2951913.2951932`

William D. Clinger and Jonathan Rees. 1991. Macros That Work. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages, POPL 1996, Orlando, Florida, USA, January 21-23, 1991*, David S. Wise (Ed.). ACM Press, 155–162. `https://doi.org/10.1145/99583.99607`

Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Proceedings of the 21st International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018 (Lecture Notes in Computer Science, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, 258–275. `https://doi.org/10.1007/978-3-319-89366-2_14`

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2015. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *21st International Conference on Types for Proofs and Programs, TYPES 2015, Tallinn, Estonia, May 18-21, 2015 (LIPIcs, Vol. 69)*, Tarmo Uustalu (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:34. `https://doi.org/10.4230/LIPICS.TYPES.2015.5`

Committee on Government Reform and Oversight. 1998. The Year 2000 Problem: Fourth Report by the Committee on Government Reform and Oversight, Together with Additional Views. `https://www.congress.gov/105/crpt/hrpt827/CRPT-105hrpt827.pdf`

Thierry Coquand. 1985. *Une théorie des constructions.* PhD Thesis. Université Paris VII., Paris, France.

Thierry Coquand and Jean Gallier. 1990. A Proof of Strong Normalization for the Theory of Constructions Using a Kripke-like Interpretation. *Technical Reports (CIS)* (July 1990). https://repository.upenn.edu/cis_reports/568 University of Pennsylvania, Philadelphia, Pennsylvania, USA.

Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. 2004. C-CoRN, the Constructive Coq Repository at Nijmegen. In *Proceedings of the 3rd International Conference on Mathematical Knowledge Management, MKM 2004, Bialowieza, Poland, September 19-21, 2004 (Lecture Notes in Computer Science, Vol. 3119)*, Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec (Eds.). Springer, 88–103. https://doi.org/10.1007/978-3-540-27818-4_7

Djordje Cubric, Peter Dybjer, and Philip J. Scott. 1998. Normalization and the Yoneda Embedding. *Math. Struct. Comput. Sci.* 8, 2 (1998), 153–192. http://journals.cambridge.org/action/displayAbstract?aid=44745

Ryan Culpepper, Matthias Felleisen, Matthew Flatt, and Shriram Krishnamurthi. 2019. From Macros to DSLs: The Evolution of Racket. In *3rd Summit on Advances in Programming Languages, SNAPL 2019, Providence, Rhode Island, USA, May 16-17, 2019 (LIPIcs, Vol. 136)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 5:1–5:19. https://doi.org/10.4230/LIPICS.SNAPL.2019.5

Nils Anders Danielsson. 2006. A Formalisation of a Dependently Typed Language as An Inductive-Recursive Family. In *International Workshop on Types for Proofs and Programs, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 4502)*, Thorsten Altenkirch and Conor McBride (Eds.). Springer, 93–109. https://doi.org/10.1007/978-3-540-74464-1_7

Rowan Davies and Frank Pfenning. 1996. A Modal Analysis of Staged Computation. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL 1996, St. Petersburg Beach, Florida, USA, January 21-24, 1996, Hans-Juergen Boehm and Guy L. Steele Jr. (Eds.). ACM Press, 258–270. https://doi.org/10.1145/237721.237788

Rowan Davies and Frank Pfenning. 2001. A Modal Analysis of Staged Computation. *J. ACM* 48, 3 (2001), 555–604. https://doi.org/10.1145/382780.382785

N.G de Bruijn. 1972. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagat. Math.* 75, 5 (1972), 381–392. https://doi.org/10.1016/1385-7258(72)90034-0

Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Proceedings of the 28th International Conference on Automated Deduction, CADE 2021, Virtual Event, July 12-15, 2021 (Lecture Notes in Computer Science, Vol. 12699)*, André Platzer and Geoff Sutcliffe (Eds.). Springer, 625–635. https://doi.org/10.1007/978-3-030-79876-5_37

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Proceedings of the 25th International Conference on Automated Deduction, CADE 2015, Berlin, Germany, August 1-7, 2015 (Lecture Notes in Computer Science, Vol. 9195)*, Amy P. Felty and Aart Middeldorp (Eds.). Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

David Delahaye. 2000. A Tactic Language for the System Coq. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning, LPAR 2000, Reunion Island, France, November 11-12, 2000 (Lecture Notes in Computer Science, Vol. 1955)*, Michel Parigot and Andrei Voronkov (Eds.). Springer, 85–95. https://doi.org/10.1007/3-540-44404-1_7

Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael D. Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. 2014. The Matter of Heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference, IMC 2014, Vancouver, British Columbia,* *Canada, November 5-7, 2014*, Carey Williamson, Aditya Akella, and Nina Taft (Eds.). ACM, 475–488. https://doi.org/10.1145/2663716.2663755

Peter Dybjer. 1995. Internal Type Theory. In *International Workshop on Types for Proofs and Programs, TYPES 1995, Torino, Italy, June 5-8, 1995, Selected Papers (Lecture Notes in Computer Science, Vol. 1158)*, Stefano Berardi and Mario Coppo (Eds.). Springer, 120–134. https://doi.org/10.1007/3-540-61780-9_66

Peter Dybjer. 2000. A General Formulation of Simultaneous Inductive-Recursive Definitions in Type Theory. *J. Symb. Log.* 65, 2 (2000), 525–549. https://doi.org/10.2307/2586554

Peter Dybjer and Anton Setzer. 2001. Indexed Induction-Recursion. In *Proceedings of the International Seminar on Proof Theory in Computer Science, PTCS 2001, Dagstuhl Castle, Germany, October 7-12, 2001 (Lecture Notes in Computer Science, Vol. 2183)*, Reinhard Kahle, Peter Schroeder-Heister, and Robert F. Stärk (Eds.). Springer, 93–113. https://doi.org/10.1007/3-540-45504-3_7

Peter Dybjer and Anton Setzer. 2003. Induction-recursion and Initial Algebras. *Ann. Pure Appl. Log.* 124, 1-3 (2003), 1–47. https://doi.org/10.1016/S0168-0072(02)00096-9

Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A Metaprogramming Framework for Formal Verification. *Proc. ACM Program. Lang.* 1, ICFP (2017), 34:1–34:29. https://doi.org/10.1145/3110278

Fredrik Nordvall Forsberg and Anton Setzer. 2010. Inductive-Inductive Definitions. In *Proceedings of the 24th International Workshop on Computer Science Logic, CSL 2010, Brno, Czech Republic, August 23-27, 2010 (Lecture Notes in Computer Science, Vol. 6247)*, Anuj Dawar and Helmut Veith (Eds.). Springer, 454–468. https://doi.org/10.1007/978-3-642-15205-4_35

Murdoch James Gabbay and Aleksandar Nanevski. 2013. Denotation of Contextual Modal Type Theory (CMTT): Syntax and Meta-programming. *J. Appl. Log.* 11, 1 (2013), 1–29. https://doi.org/10.1016/j.jal.2012.07.002

Gerhard Gentzen. 1935. Untersuchungen über das logische Schließen. I. *Mathematische Zeitschrift* 39, 1 (Dec. 1935), 176–210. https://doi.org/10.1007/BF01201353

Jean-Yves Girard. 1972. *Interpétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD Thesis. Université Paris VII., Paris, France.

Jean-Yves Girard. 1989. *Proofs and Types.* Number 7 in Cambridge tracts in theoretical computer science. Cambridge University Press, Cambridge [England] ; New York. https://dl.acm.org/doi/10.5555/64805

Georges Gonthier. 2023. *A Computer-checked Proof of the Four Color Theorem.* Technical Report. Inria. https://inria.hal.science/hal-04034866

Georges Gonthier et al. 2008. Formal Proof–the Four-color Theorem. *Notices of the AMS* 55, 11 (2008), 1382–1393. https://www.ams.org/notices/200811/tx081101382p.pdf

Jean Goubault-Larrecq. 1996. *On Computational Interpretations of the Modal Logic S4: II. The $\lambda ev Q$-calculus.* Technical Report. Univeristy of Karlsruhe, Karlsruhe, Germany.

W. T. Gowers, Ben Green, Freddie Manners, and Terence Tao. 2023. On A Conjecture of Marton. *CoRR* abs/2311.05762 (2023). arXiv:2311.05762 https://arxiv.org/abs/2311.05762

Daniel Gratzer. 2022. Normalization for Multimodal Type Theory. In *In Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2022, Haifa, Israel, August 2-5, 2022*, Christel Baier and Dana Fisman (Eds.). ACM, 2:1–2:13. https://doi.org/10.1145/3531130.3532398

Daniel Gratzer and Lars Birkedal. 2022. A Stratified Approach to Löb Induction. In *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel (LIPIcs, Vol. 228)*, Amy P. Felty (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 23:1–23:22. https://doi.org/10.4230/LIPIcs.FSCD.2022.23

Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. 2020. Multimodal Dependent Type Theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2020, Saarbrücken, Germany, July 8-11, 2020*, Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller (Eds.). ACM, 492–506. https://doi.org/10.1145/3373718.3394736

Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. 2019. Implementing A Modal Dependent Type Theory. *Proc. ACM Program. Lang.* 3, ICFP (2019), 107:1–107:29. https://doi.org/10.1145/3341711

Jason Gross, Adam Chlipala, and David I. Spivak. 2014. Experience Implementing A Performant Category-Theory Library in Coq. In *Proceedings of the 5th International Conference on Interactive Theorem Proving, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014 (Lecture Notes in Computer Science, Vol. 8558)*, Gerwin Klein and Ruben Gamboa (Eds.). Springer, 275–291. https://doi.org/10.1007/978-3-319-08970-6_18

Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao, and David Costanzo. 2011. CertiKOS: A Certified Kernel for Secure Cloud Computing. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems, APSys 2011, Shanghai, China, July 11-12, 2011*, Haibo Chen, Zheng Zhang, Sue Moon, and Yuanyuan Zhou (Eds.). ACM, 3. https://doi.org/10.1145/2103799.2103803

Robert Harper, Furio Honsell, and Gordon D. Plotkin. 1993. A Framework for Defining Logics. *J. ACM* 40, 1 (1993), 143–184. https://doi.org/10.1145/138027.138060

Jason Z. S. Hu and Jacques Carette. 2021. Formalizing Category Theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2021, Virtual Event, Denmark, January 17-19, 2021*, Catalin Hritcu and Andrei Popescu (Eds.). ACM, 327–342. https://doi.org/10.1145/3437992.3439922

Jason Z. S. Hu, Junyoung Jang, and Brigitte Pientka. 2023. Normalization by Evaluation for Modal Dependent Type Theory. *J. Funct. Program.* 33 (2023). https://doi.org/10.1017/S0956796823000060

Jason Z. S. Hu and Brigitte Pientka. 2022a. A Categorical Normalization Proof for the Modal Lambda-Calculus. In *Proceedings of the 38th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2022, Cornell University, Ithaca, New York, USA, with a satellite event at IRIF, Denis Diderot University, Paris, France, and online, July 11-13, 2022 (EPTICS, Vol. 1)*, Justin Hsu and Christine Tasson (Eds.). EpiSciences. `https://doi.org/10.46298/ENTICS.10360`

Jason Z. S. Hu and Brigitte Pientka. 2022b. An Investigation of Kripke-style Modal Type Theories. *CoRR* abs/2206.07823 (2022). arXiv:2206.07823 `https://doi.org/10.48550/arXiv.2206.07823`

Jason Z. S. Hu and Brigitte Pientka. 2023. Layered Modal Type Theories. *CoRR* abs/2305.06548 (2023). arXiv:2305.06548 `https://doi.org/10.48550/arXiv.2305.06548`

Jason Z. S. Hu and Brigitte Pientka. 2024a. DeLaM: A Dependent Layered Modal Type Theory for Meta-programming. *CoRR* abs/2404.17065 (2024). arXiv:2404.17065 `https://doi.org/10.48550/arXiv.2404.17065`

Jason Z. S. Hu and Brigitte Pientka. 2024b. Layered Modal Type Theory: Where Meta-programming Meets Intensional Analysis. In *Proceedings of the 33rd European Symposium on Programming on Programming Languages and Systems, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Part I (Lecture Notes in Computer Science, Vol. 14576)*, Stephanie Weirich (Ed.). Springer, 52–82. `https://doi.org/10.1007/978-3-031-57262-3_3`

Jason Z. S. Hu and Brigitte Pientka. 2025. A Dependent Type Theory for Meta-programming with Intensional Analysis. *Proc. ACM Program. Lang.* 9, POPL (2025). `https://doi.org/10.1145/3704851`

Jason Z. S. Hu, Brigitte Pientka, and Ulrich Schöpp. 2022. A Category Theoretic View of Contextual Types: From Simple Types to Dependent Types. *ACM Trans. Comput. Log.* 23, 4 (2022), 25:1–25:36. `https://doi.org/10.1145/3545115`

Gérard P. Huet and Amokrane Saïbi. 2000. Constructive Category Theory. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, 239–276.

Junyoung Jang, Antoine Gaulin, Jason Z. S. Hu, and Brigitte Pientka. 2024a. McLTT: A Bottom-up Approach to Implementing A Proof Assistant. `https://github.com/Beluga-lang/McLTT`

Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. 2022. Mœbius: Metaprogramming using Contextual Types: The Stage Where System F Can Pattern Match on Itself. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. `https://doi.org/10.1145/3498700`

Junyoung Jang, Sophia Roshal, Frank Pfenning, and Brigitte Pientka. 2024b. Adjoint Natural Deduction. In *9th International Conference on Formal Structures for Computation and Deduction, FSCD 2024, July 10-13, 2024, Tallinn, Estonia (LIPIcs, Vol. 299)*, Jakob Rehof (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:23. `https://doi.org/10.4230/LIPICS.FSCD.2024.15`

Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. 2018. Mtac2: Typed Tactics for Backward Reasoning in Coq. *Proc. ACM Program. Lang.* 2, ICFP (2018), 78:1–78:31. `https://doi.org/10.1145/3236773`

G. A. Kavvos. 2020. Dual-Context Calculi for Modal Logic. *Log. Methods Comput. Sci.* 16, 3 (2020). `https://doi.org/10.23638/LMCS-16(3:10)2020`

G. A. Kavvos. 2021. Intensionality, Intensional Recursion and the Gödel-Löb Axiom. *IfCoLoG Journal of Logics and their Applications* 8, 8 (2021), 2287–2312. `https://collegepublications.co.uk/ifcolog/?00050`

Akira Kawata and Atsushi Igarashi. 2019. A Dependently Typed Multi-stage Calculus. In *Proceedings of the 17th Asian Symposium on Programming Languages and Systems, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019 (Lecture Notes in Computer Science, Vol. 11893)*, Anthony Widjaja Lin (Ed.). Springer, 53–72. `https://doi.org/10.1007/978-3-030-34175-6_4`

Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml - System Description. In *Proceedings of the 12th International Symposium on Functional and Logic Programming, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014 (Lecture Notes in Computer Science, Vol. 8475)*, Michael Codish and Eijiro Sumii (Eds.). Springer, 86–102. `https://doi.org/10.1007/978-3-319-07151-0_6`

Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. 1986. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986*, William L. Scherlis, John H. Williams, and Richard P. Gabriel (Eds.). ACM, 151–161. `https://doi.org/10.1145/319838.319859`

András Kovács. 2022. Staged Compilation with Two-level Type Theory. *Proc. ACM Program. Lang.* 6, ICFP (2022), 540–569. `https://doi.org/10.1145/3547641`

Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 2018. MoSeL: a general, extensible modal framework for interactive proofs in separation logic. *Proc. ACM Program. Lang.* 2, ICFP (2018), 77:1–77:30. `https://doi.org/10.1145/3236772`

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 205–217. `https://doi.org/10.1145/3009837.3009855`

Saul A. Kripke. 1963. Semantical Analysis of Modal Logic I Normal Modal Propositional Calculi. *Mathematical Logic Quarterly* 9, 5-6 (1963), 67–96. `https://doi.org/10.1002/malq.19630090502`

Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. 2016. CompCert - A Formally Verified Optimizing Compiler. In *8th European Congress on Embedded Real Time Software and Systems, ERTS 2016, Toulouse, France, January, 2016.* `https://inria.hal.science/hal-01238879`

Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. 2018. Internal Universes in Models of Homotopy Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK (LIPIcs, Vol. 108)*, Hélène Kirchner (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22:1–22:17. https://doi.org/10.4230/LIPICS.FSCD.2018.22

Zhaohui Luo. 1990. *An Extended Calculus of Constructions*. PhD Thesis. University of Edinburgh, Edinburgh, Scotland. https://era.ed.ac.uk/handle/1842/12487

Assia Mahboubi and Enrico Tassi. 2022. *Mathematical Components*. Zenodo. https://doi.org/10.5281/zenodo.7118596

Geoffrey Mainland. 2012. Explicitly Heterogeneous Metaprogramming with MetaHaskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP 2012, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 311–322. https://doi.org/10.1145/2364527.2364572

Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Studies in proof theory, Vol. 1. Bibliopolis.

Per Martin-Löf. 1975. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium 1973*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, 73–118. https://doi.org/10.1016/S0049-237X(08)71945-1

Simone Martini and Andrea Masini. 1996. A Computational Interpretation of Modal Proofs. In *Proof Theory of Modal Logic*, Heinrich Wansing (Ed.). Springer Netherlands, Dordrecht, 213–241. https://doi.org/10.1007/978-94-017-2798-3_12

Benjamin Moon, Harley Eades III, and Dominic Orchard. 2021. Graded Modal Dependent Type Theory. In *Proceedings of the 30th European Symposium on Programming on Programming Languages and Systems, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021,* *Luxembourg City, Luxembourg, March 27 - April 1, 2021 (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer, 462–490. `https://doi.org/10.1007/978-3-030-72019-3_17`

Yuito Murase, Yuichi Nishiwaki, and Atsushi Igarashi. 2023. Contextual Modal Type Theory with Polymorphic Contexts. In *Proceedings of the 32nd European Symposium on Programming on Programming Languages and Systems, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023 (Lecture Notes in Computer Science, Vol. 13990)*, Thomas Wies (Ed.). Springer, 281–308. `https://doi.org/10.1007/978-3-031-30044-8_11`

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. 2008. Contextual Modal Type Theory. *ACM Trans. Comput. Log.* 9, 3 (2008), 23:1–23:49. `https://doi.org/10.1145/1352582.1352591`

Ulf Norell. 2007. *Towards A Practical Programming Language Based on Dependent Type Theory.* PhD Thesis. Chalmers University of Technology, Gothenburg, Sweden.

Erik Palmgren. 1998. On Universes in Type Theory. In *Twenty Five Years of Constructive Type Theory.* Oxford University Press. `https://doi.org/10.1093/oso/9780198501275.003.0012`

Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Squid: Type-safe, Hygienic, and Reusable Quasiquotes. In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala, SCALA@SPLASH 2017, Vancouver, British Columbia, Canada, October 22-23, 2017*, Heather Miller, Philipp Haller, and Ondrej Lhoták (Eds.). ACM, 56–66. `https://doi.org/10.1145/3136000.3136005`

Emir Pasalic, Walid Taha, and Tim Sheard. 2002. Tagless Staged Interpreters for Typed Languages. In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming, ICFP 2002, Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, Mitchell Wand and Simon L. Peyton Jones (Eds.). ACM, 218–229. `https://doi.org/10.1145/581478.581499`

Daniel Peebles, James Deikun, Ulf Norell, Dan Doel, Darius Jahandarie, and James Cook. 2018. Categories: Categories Parametrized by Morphism Equality in Agda. `https://github.com/copumpkin/categories`

Frank Pfenning and Rowan Davies. 2001. A Judgmental Reconstruction of Modal Logic. *Math. Struct. Comput. Sci.* 11, 4 (2001), 511–540. `https://doi.org/10.1017/S0960129501003322`

Frank Pfenning and Christine Paulin-Mohring. 1989. Inductively Defined Types in the Calculus of Constructions. In *Proceedings of the 5th International Conference on Mathematical Foundations of Programming Semantics, MFPS 1989, Tulane University, New Orleans, Louisiana, USA, March 29-April 1, 1989 (Lecture Notes in Computer Science, Vol. 442)*, Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt (Eds.). Springer, 209–228. `https://doi.org/10.1007/BFB0040259`

Frank Pfenning and Hao-Chi Wong. 1995. On A Modal Lambda Calculus for S4. In *11th Annual Conference on Mathematical Foundations of Programming Semantics, MFPS 1995, Tulane University, New Orleans, Louisiana, USA, March 29-April 1, 1995 (Electronic Notes in Theoretical Computer Science, Vol. 1)*, Stephen D. Brookes, Michael G. Main, Austin Melton, and Michael W. Mislove (Eds.). Elsevier, 515–534. `https://doi.org/10.1016/S1571-0661(04)00028-3`

Brigitte Pientka. 2008. A Type-theoretic Foundation for Programming with Higher-order Abstract Syntax and First-class Substitutions. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, George C. Necula and Philip Wadler (Eds.). ACM, 371–382. `https://doi.org/10.1145/1328438.1328483`

Brigitte Pientka and Andreas Abel. 2015. Well-Founded Recursion over Contextual Objects. In *13th International Conference on Typed Lambda Calculi and Applications, TLCA 2015, Warsaw, Poland, July 1-3, 2015 (LIPIcs, Vol. 38)*, Thorsten Altenkirch (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 273–287. `https://doi.org/10.4230/LIPICS.TLCA.2015.273`

Brigitte Pientka and Jana Dunfield. 2008. Programming with Proofs and Explicit Contexts. In *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, Valencia, Spain, July 15-17, 2008*, Sergio Antoy and Elvira Albert (Eds.). ACM, 163–173. `https://doi.org/10.1145/1389449.1389469`

Brigitte Pientka and Ulrich Schöpp. 2020. Semantical Analysis of Contextual Types. In *Proceedings of the 23rd International Conference on Foundations of Software Science and Computation Structures, FoSSaCS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020 (Lecture Notes in Computer Science, Vol. 12077)*, Jean Goubault-Larrecq and Barbara König (Eds.). Springer, 502–521. `https://doi.org/10.1007/978-3-030-45231-5_26`

Brigitte Pientka, David Thibodeau, Andreas Abel, Francisco Ferreira, and Rébecca Zucchini. 2019. A Type Theory for Defining Logics and Proofs. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, British Columbia, Canada, June 24-27, 2019*. IEEE, 1–13. `https://doi.org/10.1109/LICS.2019.8785683`

Dag Prawitz. 1965. *Natural Deduction: A Proof-theoretical Study*. Stockholm.

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study. `https://homotopytypetheory.org/book/`

Loïc Pujet and Nicolas Tabareau. 2022. Observational Equality: Now for Good. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–27. `https://doi.org/10.1145/3498693`

Loïc Pujet and Nicolas Tabareau. 2023. Impredicative Observational Equality. *Proc. ACM Program. Lang.* 7, POPL (2023), 2171–2196. `https://doi.org/10.1145/3571739`

Pierre-Marie Pédrot. 2019. Ltac2: Tactical Warfare. In *The 5th International Workshop on Coq for Programming Languages, CoqPL 2019, Lisbon, Portugal*. 13–19.

John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.* 11, 4 (1998), 363–397. https://doi.org/10.1023/A:1010027404223

Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. 2001. Primitive Recursion for Higher-order Abstract Syntax. *Theor. Comput. Sci.* 266, 1-2 (2001), 1–57. https://doi.org/10.1016/S0304-3975(00)00418-7

Tim Sheard and Simon L. Peyton Jones. 2002. Template Meta-programming for Haskell. *SIGPLAN Notices* 37, 12 (2002), 60–75. https://doi.org/10.1145/636517.636528

Michael Shulman. 2018. Brouwer's Fixed-point Theorem in Real-cohesive Homotopy Type Theory. *Math. Struct. Comput. Sci.* 28, 6 (2018), 856–941. https://doi.org/10.1017/S0960129517000147

Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2020. The MetaCoq Project. *J. Autom. Reason.* 64, 5 (2020), 947–999. https://doi.org/10.1007/s10817-019-09540-0

Jonathan Sterling. 2022. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory.* PhD Thesis. Carnegie Mellon University, Pittsburgh, Pennsylvania, USA. https://doi.org/10.1184/r1/19632681.v1

Walid Taha. 2000. A Sound Reduction Semantics for Untyped CBN Multi-stage Computation. Or, the Theory of MetaML is Non-trivial (Extended Abstract). In *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 2000, Boston, Massachusetts, USA, January 22-23, 2000*, Julia L. Lawall (Ed.). ACM, 34–43. https://doi.org/10.1145/328690.328697

Walid Taha and Tim Sheard. 1997. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM 1997, Amsterdam, the Netherlands,* *June 12-13, 1997*, John P. Gallagher, Charles Consel, and A. Michael Berman (Eds.). ACM, 203–217. https://doi.org/10.1145/258993.259019

Walid Taha and Tim Sheard. 2000. MetaML and Multi-stage Programming with Explicit Annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658

Terence Tao. 2023. A Maclaurin Type Inequality. *CoRR* abs/2310.05328 (2023). arXiv:2310.05328 https://arxiv.org/abs/2310.05328

The Agda Team. 2024. Agda 2.6.4.3. https://wiki.portal.chalmers.se/agda/pmwiki.php

The Coq Development Team. 2023. *The Coq Proof Assistant.* https://doi.org/10.5281/zenodo.8161141

The Mathlib Community. 2020. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, Louisiana, USA, January 20-21, 2020*, Jasmin Blanchette and Catalin Hritcu (Eds.). ACM, 367–381. https://doi.org/10.1145/3372885.3373824

Amin Timany and Bart Jacobs. 2016. Category Theory in Coq 8.5. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, Porto, Portugal, June 22-26, 2016 (LIPIcs, Vol. 52)*, Delia Kesner and Brigitte Pientka (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:18. https://doi.org/10.4230/LIPIcs.FSCD.2016.30

Nachiappan Valliappan, Fabian Ruch, and Carlos Tomé Cortiñas. 2022. Normalization for Fitch-style Modal Calculi. *Proc. ACM Program. Lang.* 6, ICFP (2022), 772–798. https://doi.org/10.1145/3547649

Paul van der Walt and Wouter Swierstra. 2012. Engineering Proof by Reflection in Agda. In *24th International Symposium on Implementation and Application of Functional Languages, IFL 2012, Oxford, UK, August 30-September 1, 2012, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 8241)*, Ralf Hinze (Ed.). Springer, 157–173. `https://doi.org/10.1007/978-3-642-41582-1_10`

Marcos Viera and Alberto Pardo. 2006. A Multi-stage Language with Intensional Analysis. In *Proceedings of 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006*, Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen (Eds.). ACM, 11–20. `https://doi.org/10.1145/1173706.1173709`

Washington Post. 2007. Sept. 26, 1983: The Man Who Saved the World by Doing ... Nothing. `https://www.wired.com/2007/09/dayintech-0926-2/`

Makarius Wenzel et al. 2024. The Isabelle/Isar Reference Manual. `https://isabelle.in.tum.de/doc/isar-ref.pdf`

Alfred North Whitehead and Bertrand Russell. 1925–1927. *Principia Mathematica.* Cambridge University Press, Cambridge [England].

Pawel Wieczorek and Dariusz Biernacki. 2018. A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, California, USA, January 8-9, 2018*, June Andronick and Amy P. Felty (Eds.). ACM, 266–279. `https://doi.org/10.1145/3167091`

John Wiegley. 2019. Category-theory: Category Theory in Coq. `https://github.com/jwiegley/category-theory`

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94. `https://doi.org/10.1006/INCO.1994.1093`

Noam Zeilberger. 2008. Focusing and Higher-order Abstract Syntax. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Lan-*

guages, POPL 2008, San Francisco, California, USA, January 7-12, 2008, George C. Necula and Philip Wadler (Eds.). ACM, 359–369. `https://doi.org/10.1145/1328438.1328482`

Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2013. Mtac: A Monad for Typed Tactic Programming in Coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming, ICFP 2013, Boston, Massachusetts, USA, September 25-27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 87–100. `https://doi.org/10.1145/2500365.2500579`

Appendices

A

Missing Typing Rules for Chapter 2

The additional rules primarily characterize how explicit K-substitutions interact with terms. First we have identity and composition.

$$\frac{\overrightarrow{\Gamma} \vdash t : T}{\overrightarrow{\Gamma} \vdash t[\overrightarrow{I}\,] \approx t : T} \qquad \frac{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'' \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}' \qquad \overrightarrow{\Gamma}'' \vdash t : T}{\overrightarrow{\Gamma} \vdash t[\overrightarrow{\sigma} \circ \overrightarrow{\delta}\,] \approx t[\overrightarrow{\sigma}][\overrightarrow{\delta}\,] : T}$$

Then there are a few cases for variables. When hitting a term extension, then the term is extracted if the topmost variable x is being substituted. In the following rules, I assume x is the topmost variable. It is fine since I am working with de Bruijn indices, so x has de Bruijn index 0. In the second rule, the de Bruijn index for y on the left hand side is one larger than that on the right:

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'; \Gamma \qquad \overrightarrow{\Gamma} \vdash t : A}{\overrightarrow{\Gamma} \vdash x[\overrightarrow{\sigma}, t] \approx t : A} \qquad \frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'; \Gamma \qquad \overrightarrow{\Gamma} \vdash t : B \qquad y : A \in \Gamma}{\overrightarrow{\Gamma} \vdash y[\overrightarrow{\sigma}, t] \approx y[\overrightarrow{\sigma}] : A}$$

In the following rule, the de Bruijn index for y is actually increased by one on the right hand side:

$$\frac{y : A \in \Gamma}{\vec{\Gamma}; (\Gamma, x : B) \vdash y[\mathsf{wk}] \approx y : A}$$

The following rules characterize how K-substitutions are applied to other terms:

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta} \qquad \vec{\Delta}; \cdot \vdash t : A}{\vec{\Gamma} \vdash \texttt{box}\ t[\vec{\sigma}] \approx \texttt{box}\ (t[\vec{\sigma}; \Uparrow^1]) : \Box A}$$

$$\frac{\vec{\Delta} \vdash t : \Box A \qquad \vec{\Gamma}; \vec{\Gamma}' \vdash \vec{\sigma} : \vec{\Delta}; \vec{\Delta}' \qquad |\vec{\Delta}'| = n \qquad |\vec{\Gamma}'| = \mathcal{O}(\vec{\sigma}, n)}{\vec{\Gamma}; \vec{\Gamma}' \vdash \texttt{unbox}_n\ t[\vec{\sigma}] \approx \texttt{unbox}_{\mathcal{O}(\vec{\sigma}, n)}\ (t[\vec{\sigma} \mid n]) : A}$$

$$\frac{\vec{\Gamma}; \Gamma \vdash \vec{\sigma} : \vec{\Delta}; \Delta \qquad \vec{\Delta}; \Delta, x : A \vdash t : B}{\vec{\Gamma}; \Gamma \vdash \lambda x.t[\vec{\sigma}] \approx \lambda x.(t[(\vec{\sigma} \circ \mathsf{wk}_x), x/x]) : A \longrightarrow B}$$

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta} \qquad \vec{\Delta} \vdash s : A \longrightarrow B \qquad \vec{\Delta} \vdash t : A}{\vec{\Gamma} \vdash s\ t[\vec{\sigma}] \approx (s[\vec{\sigma}])\ (t[\vec{\sigma}]) : B}$$

K-substitutions also have non-trivial interactions among themselves. First, there are identity laws and associativity of composition:

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vdash \vec{\sigma} \circ \vec{I} \approx \vec{\sigma} : \vec{\Delta}} \qquad \frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vdash \vec{I} \circ \vec{\sigma} \approx \vec{\sigma} : \vec{\Delta}}$$

$$\frac{\vec{\Gamma}'' \vdash \vec{\sigma}'' : \vec{\Gamma}''' \qquad \vec{\Gamma}' \vdash \vec{\sigma}' : \vec{\Gamma}'' \qquad \vec{\Gamma} \vdash \vec{\sigma} : \vec{\Gamma}'}{\vec{\Gamma} \vdash (\vec{\sigma}'' \circ \vec{\sigma}') \circ \vec{\sigma} \approx \vec{\sigma}'' \circ (\vec{\sigma}' \circ \vec{\sigma}) : \vec{\Gamma}'''}$$

The following rules characterize the propagation of composition under term and modal extensions:

$$\frac{\vec{\Gamma}' \vdash \vec{\sigma} : \vec{\Gamma}''; \Gamma \qquad \vec{\Gamma}' \vdash t : A \qquad \vec{\Gamma} \vdash \vec{\delta} : \vec{\Gamma}'}{\vec{\Gamma} \vdash \vec{\sigma}, t \circ \vec{\delta} \approx (\vec{\sigma} \circ \vec{\delta}), t[\vec{\delta}] : \vec{\Gamma}''; (\Gamma, x : A)}$$

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Gamma}' \qquad \vec{\Gamma}'' \vdash \vec{\delta} : \vec{\Gamma}; \vec{\Delta} \qquad |\vec{\Delta}| = n}{\vec{\Gamma}'' \vdash (\vec{\sigma}; \Uparrow^n) \circ \vec{\delta} \approx (\vec{\sigma} \circ \vec{\delta} \mid n); \Uparrow^{\mathcal{O}(\vec{\delta}, n)} : \vec{\Gamma}'; \cdot}$$

The following rules describe weakenings and modal extensions:

$$\frac{\vec{\Gamma}' \vdash \vec{\sigma} : \vec{\Gamma}; \Gamma \qquad \vec{\Gamma}' \vdash t : A}{\vec{\Gamma}' \vdash \mathsf{wk}_x \circ (\vec{\sigma}, t) \approx \vec{\sigma} : \vec{\Gamma}; \Gamma} \qquad \frac{\vec{\Gamma}' \vdash \vec{\sigma} : \vec{\Gamma}; (\Gamma, x : A)}{\vec{\Gamma}' \vdash \vec{\sigma} \approx (\mathsf{wk} \circ \vec{\sigma}), (x[\vec{\sigma}]) : \vec{\Gamma}; (\Gamma, x : A)}$$

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}; \cdot \qquad |\vec{\Delta}| > 0 \qquad \mathcal{O}(\vec{\sigma}, 1) = n}{\vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\sigma} \mid 1; \Uparrow^n : \vec{\Delta}; \cdot}$$

B

Full Set of Rules for Mint

$\boxed{\vdash \overrightarrow{\Gamma}}$ Context stack $\overrightarrow{\Gamma}$ is well formed.

$$\frac{}{\vdash \epsilon; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma}}{\vdash \overrightarrow{\Gamma}; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma}; \Gamma \qquad \overrightarrow{\Gamma}; \Gamma \vdash T : \texttt{Ty}_i}{\vdash \overrightarrow{\Gamma}; \Gamma, x : T}$$

$\boxed{\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}}$ $\overrightarrow{\Gamma}$ and $\overrightarrow{\Delta}$ are equivalent context stacks.

$$\frac{}{\vdash \epsilon; \cdot \approx \epsilon; \cdot} \qquad \frac{\vdash \overrightarrow{\Gamma} \approx \overrightarrow{\Delta}}{\vdash \overrightarrow{\Gamma}; \cdot \approx \overrightarrow{\Delta}; \cdot}$$

$$\frac{\begin{array}{c}\vdash \overrightarrow{\Gamma}; \Gamma \approx \overrightarrow{\Delta}; \Delta \\ \overrightarrow{\Gamma}; \Gamma \vdash T \approx T' : \texttt{Ty}_i \qquad \overrightarrow{\Delta}; \Delta \vdash T \approx T' : \texttt{Ty}_i \qquad \overrightarrow{\Gamma}; \Gamma \vdash T : \texttt{Ty}_i \qquad \overrightarrow{\Delta}; \Delta \vdash T' : \texttt{Ty}_i\end{array}}{\vdash \overrightarrow{\Gamma}; \Gamma x : T \approx \overrightarrow{\Delta}; \Delta, x : T'}$$

$\boxed{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}$: $\vec{\sigma}$ is a K-substitution susbtituting terms in $\vec{\Delta}$ into ones in $\vec{\Gamma}$.

$$\frac{\vdash \vec{\Gamma}}{\vec{\Gamma} \vdash \vec{I} : \vec{\Gamma}} \qquad \frac{\vdash \vec{\Gamma}; \Gamma, x : T}{\vec{\Gamma}; \Gamma, x : T \vdash \mathsf{wk} : \vec{\Gamma}; \Gamma} \qquad \frac{\begin{array}{c}\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Gamma}'; \Gamma \\ \vec{\Gamma}'; \Gamma \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash t : T[\vec{\sigma}]\end{array}}{\vec{\Gamma} \vdash \vec{\sigma}, t : \vec{\Gamma}'; \Gamma, x : T}$$

$$\frac{\begin{array}{c}\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta} \\ \vdash \vec{\Gamma}; \vec{\Gamma}' \qquad |\vec{\Gamma}'| = n\end{array}}{\vec{\Gamma}; \vec{\Gamma}' \vdash \vec{\sigma}; \Uparrow^n : \vec{\Delta}; \cdot} \qquad \frac{\begin{array}{c}\vec{\Gamma}' \vdash \vec{\sigma} : \vec{\Gamma}'' \\ \vec{\Gamma} \vdash \vec{\delta} : \vec{\Gamma}'\end{array}}{\vec{\Gamma} \vdash \vec{\sigma} \circ \vec{\delta} : \vec{\Gamma}''} \qquad \frac{\begin{array}{c}\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta} \\ \vdash \vec{\Delta} \approx \vec{\Delta}'\end{array}}{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}'}$$

$\boxed{\vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\delta} : \vec{\Delta}}$ $\vec{\sigma}$ and $\vec{\delta}$ are equivalent in K-substituting terms in $\vec{\Delta}$ into ones in $\vec{\Gamma}$.

The congruence rules:

$$\frac{\vdash \vec{\Gamma}}{\vec{\Gamma} \vdash \vec{I} \approx \vec{I} : \vec{\Gamma}} \qquad \frac{\vdash \vec{\Gamma}; \Gamma, x : T}{\vec{\Gamma}; \Gamma, x : T \vdash \mathsf{wk} \approx \mathsf{wk} : \vec{\Gamma}; \Gamma}$$

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Gamma}'; \Gamma \qquad \vec{\Gamma}'; \Gamma \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash t \approx t' : T[\vec{\sigma}]}{\vec{\Gamma} \vdash \vec{\sigma}, t \approx \vec{\sigma}, t' : \vec{\Gamma}'; \Gamma, x : T}$$

$$\frac{\begin{array}{c}\vec{\Gamma} \vdash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Delta} \\ \vdash \vec{\Gamma}; \vec{\Gamma}' \qquad |\vec{\Gamma}'| = n\end{array}}{\vec{\Gamma}; \vec{\Gamma}' \vdash \vec{\sigma}; \Uparrow^n \approx \vec{\sigma}'; \Uparrow^n : \vec{\Delta}; \cdot} \qquad \frac{\begin{array}{c}\vec{\Gamma}' \vdash \vec{\sigma} \approx \vec{\sigma}' : \vec{\Gamma}'' \\ \vec{\Gamma} \vdash \vec{\delta} \approx \vec{\delta}' : \vec{\Gamma}'\end{array}}{\vec{\Gamma} \vdash \vec{\sigma} \circ \vec{\delta} \approx \vec{\sigma}' \circ \vec{\delta}' : \vec{\Gamma}''}$$

The categorical rules:

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vdash \vec{\sigma} \circ \vec{I} \approx \vec{\sigma} : \vec{\Delta}} \qquad \frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vdash \vec{I} \circ \vec{\sigma} \approx \vec{\sigma} : \vec{\Delta}}$$

$$\frac{\vec{\Gamma}'' \vdash \vec{\sigma}'' : \vec{\Gamma}''' \qquad \vec{\Gamma}' \vdash \vec{\sigma}' : \vec{\Gamma}'' \qquad \vec{\Gamma} \vdash \vec{\sigma} : \vec{\Gamma}'}{\vec{\Gamma} \vdash (\vec{\sigma}'' \circ \vec{\sigma}') \circ \vec{\sigma} \approx \vec{\sigma}'' \circ (\vec{\sigma}' \circ \vec{\sigma}) : \vec{\Gamma}'''}$$

Other rules:

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma}' \approx \overrightarrow{\sigma} : \overrightarrow{\Delta}} \qquad \frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta} \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma}' \approx \overrightarrow{\sigma}'' : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}'' : \overrightarrow{\Delta}}$$

$$\frac{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}''; \Gamma \qquad \overrightarrow{\Gamma}'; \Gamma \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma}' \vdash t : T[\overrightarrow{\sigma}] \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}'}{\overrightarrow{\Gamma} \vdash (\overrightarrow{\sigma}, t) \circ \overrightarrow{\delta} \approx (\overrightarrow{\sigma} \circ \overrightarrow{\delta}), t[\overrightarrow{\delta}] : \overrightarrow{\Gamma}''; \Gamma, x : T}$$

$$\frac{\begin{array}{c}\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta} \\ \overrightarrow{\Delta} \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t : T[\overrightarrow{\sigma}]\end{array}}{\overrightarrow{\Gamma} \vdash \mathsf{wk} \circ (\overrightarrow{\sigma}, t) \approx \overrightarrow{\sigma} : \overrightarrow{\Delta}} \qquad \frac{\begin{array}{c}\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}' \qquad \overrightarrow{\Gamma}'' \vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}; \overrightarrow{\Delta} \\ |\overrightarrow{\Delta}| = n \qquad \vdash \overrightarrow{\Gamma}; \overrightarrow{\Delta}\end{array}}{\overrightarrow{\Gamma}'' \vdash (\overrightarrow{\sigma}; \Uparrow^n) \circ \overrightarrow{\delta} \approx (\overrightarrow{\sigma} \circ \overrightarrow{\delta} \mid n); \Uparrow^{\mathcal{O}(\overrightarrow{\delta}, n)} : \overrightarrow{\Gamma}'; \cdot}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}; \cdot \qquad |\overrightarrow{\Delta}| > 0}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma} \mid 1; \Uparrow^{\mathcal{O}(\overrightarrow{\sigma}, 1)} : \overrightarrow{\Delta}; \cdot} \qquad \frac{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}; \Gamma, x : T}{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} \approx (\mathsf{wk} \circ \overrightarrow{\sigma}), x[\overrightarrow{\sigma}] : \overrightarrow{\Gamma}; (\Gamma, x : T)}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta} \qquad \vdash \overrightarrow{\Delta} \approx \overrightarrow{\Delta}'}{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}'}$$

To define the variable rule for the typing judgment, the lookup judgment $x : T \in \overrightarrow{\Gamma}$ is defined as:

$$\frac{}{x : T[\mathsf{wk}] \in \overrightarrow{\Gamma}; \Gamma, x : T} \qquad \frac{x : T \in \overrightarrow{\Gamma}; \Gamma}{x : T[\mathsf{wk}] \in \overrightarrow{\Gamma}; \Gamma, y : S}$$

In the first rule, the de Bruijn index of x is 0. In the second rule, the de Bruijn index of x is increased by 1 in the conclusion.

$\boxed{\overrightarrow{\Gamma} \vdash t : T}$ Term t has type T in context stack $\overrightarrow{\Gamma}$.

$$\frac{\vdash \overrightarrow{\Gamma};\Gamma \qquad x : T \in \overrightarrow{\Gamma};\Gamma}{\overrightarrow{\Gamma};\Gamma \vdash x : T} \qquad \frac{\overrightarrow{\Gamma} \vdash t : T \qquad \overrightarrow{\Gamma} \vdash T \approx T' : \texttt{Ty}_i}{\overrightarrow{\Gamma} \vdash t : T'}$$

$$\frac{\overrightarrow{\Delta} \vdash t : T \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash t[\overrightarrow{\sigma}] : T[\overrightarrow{\sigma}]} \qquad \frac{\overrightarrow{\Gamma};\Gamma \vdash S : \texttt{Ty}_i \qquad \overrightarrow{\Gamma};\Gamma, x : S \vdash T : \texttt{Ty}_i}{\overrightarrow{\Gamma};\Gamma \vdash \Pi(x : S).T : \texttt{Ty}_i}$$

$$\frac{\begin{array}{c}\overrightarrow{\Gamma};\Gamma \vdash S : \texttt{Ty}_i \\ \overrightarrow{\Gamma};\Gamma, x : S \vdash t : T\end{array}}{\overrightarrow{\Gamma};\Gamma \vdash \lambda x.t : \Pi(x : S).T} \qquad \frac{\begin{array}{c}\overrightarrow{\Gamma};\Gamma \vdash S : \texttt{Ty}_i \qquad \overrightarrow{\Gamma};\Gamma, x : S \vdash T : \texttt{Ty}_i \\ \overrightarrow{\Gamma};\Gamma \vdash t : \Pi(x : S).T \qquad \overrightarrow{\Gamma};\Gamma \vdash s : S\end{array}}{\overrightarrow{\Gamma};\Gamma \vdash t\ s : T[\overrightarrow{I}, s]}$$

$$\frac{\vdash \overrightarrow{\Gamma}}{\overrightarrow{\Gamma} \vdash \texttt{Nat} : \texttt{Ty}_i} \qquad \frac{\vdash \overrightarrow{\Gamma}}{\overrightarrow{\Gamma} \vdash \mathsf{zero} : \texttt{Nat}} \qquad \frac{\overrightarrow{\Gamma} \vdash t : \texttt{Nat}}{\overrightarrow{\Gamma} \vdash \mathsf{succ}\ t : \texttt{Nat}}$$

$$\frac{\begin{array}{c}\overrightarrow{\Gamma};\Gamma, x : \texttt{Nat} \vdash M : \texttt{Ty}_i \qquad \overrightarrow{\Gamma};\Gamma \vdash s : M[\overrightarrow{I}, \mathsf{zero}] \\ \overrightarrow{\Gamma};\Gamma, x : \texttt{Nat}, y : M \vdash s' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathsf{succ}\ v_1] \qquad \overrightarrow{\Gamma};\Gamma \vdash t : \texttt{Nat}\end{array}}{\overrightarrow{\Gamma};\Gamma \vdash \mathsf{elim}_{\texttt{Nat}}\ (x.M)\ s\ (x, y.s')\ t : M[\overrightarrow{I}, t]} \qquad \frac{\vdash \overrightarrow{\Gamma}}{\overrightarrow{\Gamma} \vdash \texttt{Ty}_i : \texttt{Ty}_{1+i}}$$

$$\frac{\overrightarrow{\Gamma} \vdash T : \texttt{Ty}_i}{\overrightarrow{\Gamma} \vdash T : \texttt{Ty}_{1+i}} \qquad \frac{\overrightarrow{\Gamma}; \cdot \vdash T : \texttt{Ty}_i}{\overrightarrow{\Gamma} \vdash \square T : \texttt{Ty}_i} \qquad \frac{\overrightarrow{\Gamma}; \cdot \vdash t : T}{\overrightarrow{\Gamma} \vdash \texttt{box}\ t : \square T}$$

$$\frac{\overrightarrow{\Gamma}; \cdot \vdash T : \texttt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t : \square T \qquad \vdash \overrightarrow{\Gamma}; \overrightarrow{\Delta} \qquad |\overrightarrow{\Delta}| = n}{\overrightarrow{\Gamma}; \overrightarrow{\Delta} \vdash \texttt{unbox}_n\ t : T[\overrightarrow{I}; \Uparrow^n]}$$

$\boxed{\overrightarrow{\Gamma} \vdash t \approx s : T}$ Terms t and s of type T are equivalent in context stack $\overrightarrow{\Gamma}$.
The congruence rules:

$$\frac{\vdash \overrightarrow{\Gamma};\Gamma \qquad x : T \in \overrightarrow{\Gamma};\Gamma}{\overrightarrow{\Gamma};\Gamma \vdash x \approx x : T} \qquad \frac{\begin{array}{c}\overrightarrow{\Gamma};\Gamma \vdash S : \texttt{Ty}_i \\ \overrightarrow{\Gamma};\Gamma \vdash S \approx S' : \texttt{Ty}_i \qquad \overrightarrow{\Gamma};\Gamma, x : S \vdash T \approx T' : \texttt{Ty}_i\end{array}}{\overrightarrow{\Gamma};\Gamma \vdash \Pi(x : S).T \approx \Pi(x : S').T' : \texttt{Ty}_i}$$

$$\frac{\overrightarrow{\Delta} \vdash t \approx t' : T \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} \approx \overrightarrow{\sigma}' : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash t[\overrightarrow{\sigma}] \approx t'[\overrightarrow{\sigma}'] : T[\overrightarrow{\sigma}]} \qquad \frac{\overrightarrow{\Gamma};\Gamma \vdash S : \texttt{Ty}_i \qquad \overrightarrow{\Gamma};\Gamma, x : S \vdash t \approx t' : T}{\overrightarrow{\Gamma};\Gamma \vdash \lambda x.t \approx \lambda x.t' : \Pi(x : S).T}$$

$$\frac{\begin{array}{c}\vec{\Gamma};\Gamma \vdash S : \mathtt{Ty}_i \\ \vec{\Gamma};\Gamma, x : S \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma};\Gamma \vdash t \approx t' : \Pi(x : S).T \qquad \vec{\Gamma};\Gamma \vdash s \approx s' : S\end{array}}{\vec{\Gamma};\Gamma \vdash t\ s \approx t'\ s' : T[\vec{I}, s]}$$

$$\frac{\vdash \vec{\Gamma}}{\vec{\Gamma} \vdash \mathsf{zero} \approx \mathsf{zero} : \mathtt{Nat}} \qquad \frac{\vec{\Gamma} \vdash t \approx t' : \mathtt{Nat}}{\vec{\Gamma} \vdash \mathsf{succ}\ t \approx \mathsf{succ}\ t' : \mathtt{Nat}}$$

$$\frac{\begin{array}{c}\vec{\Gamma};\Gamma, x : \mathtt{Nat} \vdash M \approx M' : \mathtt{Ty}_i \qquad \vec{\Gamma};\Gamma \vdash s_1 \approx s_1' : M[\vec{I}, \mathsf{zero}] \\ \vec{\Gamma};\Gamma, x : \mathtt{Nat}, y : M \vdash s_2 \approx s_2' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathsf{succ}\ v_1] \qquad \vec{\Gamma};\Gamma \vdash t \approx t' : \mathtt{Nat}\end{array}}{\vec{\Gamma};\Gamma \vdash \mathsf{elim}_{\mathtt{Nat}}\ (x.M)\ s_1\ (x, y.s_2)\ t \approx \mathsf{elim}_{\mathtt{Nat}}\ (x.M')\ s_1'\ (x, y.s_2')\ t' : M[\vec{I}, t]}$$

$$\frac{\vec{\Gamma};\cdot \vdash T \approx T' : \mathtt{Ty}_i}{\vec{\Gamma} \vdash \square T \approx \square T' : \mathtt{Ty}_i} \qquad \frac{\vec{\Gamma};\cdot \vdash t \approx t' : T}{\vec{\Gamma} \vdash \mathtt{box}\ t \approx \mathtt{box}\ t' : \square T}$$

$$\frac{\vec{\Gamma};\cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash t \approx t' : \square T \qquad \vdash \vec{\Gamma};\vec{\Delta} \qquad |\vec{\Delta}| = n}{\vec{\Gamma};\vec{\Delta} \vdash \mathtt{unbox}_n\ t \approx \mathtt{unbox}_n\ t' : T[\vec{I};\Uparrow^n]}$$

The β and η rules:

$$\frac{\begin{array}{c}\vec{\Gamma};\Gamma \vdash S : \mathtt{Ty}_i \qquad \vec{\Gamma};\Gamma, x : S \vdash T : \mathtt{Ty}_i \\ \vec{\Gamma};\Gamma, x : S \vdash t : T \qquad \vec{\Gamma};\Gamma \vdash s : S\end{array}}{\vec{\Gamma};\Gamma \vdash (\lambda x.t)\ s \approx t[\vec{I}, s] : T[\vec{I}, s]} \qquad \frac{\begin{array}{c}\vec{\Gamma};\Gamma \vdash S : \mathtt{Ty}_i \qquad \vec{\Gamma};\Gamma, x : S \vdash T : \mathtt{Ty}_i \\ \vec{\Gamma};\Gamma \vdash t : \Pi(x : S).T\end{array}}{\vec{\Gamma};\Gamma \vdash t \approx \lambda x.(t[\mathsf{wk}]\ x) : \Pi(x : S).T}$$

$$\frac{\begin{array}{c}\vec{\Gamma};\Gamma, x : \mathtt{Nat} \vdash M : \mathtt{Ty}_i \\ \vec{\Gamma};\Gamma \vdash s : M[\vec{I}, \mathsf{zero}] \qquad \vec{\Gamma};\Gamma, x : \mathtt{Nat}, y : M \vdash s' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathsf{succ}\ v_1]\end{array}}{\vec{\Gamma};\Gamma \vdash \mathsf{elim}_{\mathtt{Nat}}\ (x.M)\ s\ (x, y.s')\ \mathsf{zero} \approx s : M[\vec{I}, \mathsf{zero}]}$$

$$\frac{\begin{array}{c}\vec{\Gamma};\Gamma, x : \mathtt{Nat} \vdash M : \mathtt{Ty}_i \qquad \vec{\Gamma};\Gamma \vdash s : M[\vec{I}, \mathsf{zero}] \\ \vec{\Gamma};\Gamma, x : \mathtt{Nat}, y : M \vdash s' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathsf{succ}\ v_1] \\ \vec{\Gamma};\Gamma \vdash t : \mathtt{Nat} \qquad t' = \mathsf{elim}_{\mathtt{Nat}}\ (x.M)\ s\ (x, y.s')\ t\end{array}}{\vec{\Gamma};\Gamma \vdash \mathsf{elim}_{\mathtt{Nat}}\ (x.M)\ s\ (x, y.s')\ (\mathsf{succ}\ t) \approx s'[\vec{I}, t, t'] : M[\vec{I}, \mathsf{succ}\ t]}$$

$$\frac{\begin{array}{c}\vec{\Gamma};\cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma};\cdot \vdash t : T \\ \vdash \vec{\Gamma};\vec{\Delta} \qquad |\vec{\Delta}| = n\end{array}}{\vec{\Gamma};\vec{\Delta} \vdash \mathtt{unbox}_n\ (\mathtt{box}\ t) \approx t[\vec{I};\Uparrow^n] : T[\vec{I};\Uparrow^n]} \qquad \frac{\vec{\Gamma};\cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash t : \square T}{\vec{\Gamma} \vdash t \approx \mathtt{box}\ (\mathtt{unbox}_1\ t) : \square T}$$

General K-substitution rules:

$$\frac{\overrightarrow{\Gamma} \vdash t : T}{\overrightarrow{\Gamma} \vdash t[\overrightarrow{I}] \approx t : T} \qquad \frac{\overrightarrow{\Gamma}' \vdash \overrightarrow{\sigma} : \overrightarrow{\Gamma}'' \qquad \overrightarrow{\Gamma} \vdash \overrightarrow{\delta} : \overrightarrow{\Gamma}' \qquad \overrightarrow{\Gamma}'' \vdash t : T}{\overrightarrow{\Gamma} \vdash t[\overrightarrow{\sigma} \circ \overrightarrow{\delta}] \approx t[\overrightarrow{\sigma}][\overrightarrow{\delta}] : T[\overrightarrow{\sigma} \circ \overrightarrow{\delta}]}$$

The variable rules, where

- in the first rule, the de Bruijn index of x on the right hand side is increased by 1;
- in the second rule, the de Bruijn index of x is 0;
- in the third rule, the de Bruijn index of x on the left hand side is increased by 1;

$$\frac{\vdash \overrightarrow{\Gamma}; \Gamma, y : T \qquad x : T' \in \overrightarrow{\Gamma}; \Gamma}{\overrightarrow{\Gamma}; \Gamma \vdash x[\mathsf{wk}] \approx x : T'[\mathsf{wk}]} \qquad \frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}; \Delta \qquad \overrightarrow{\Delta}; \Delta \vdash T : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t : T[\overrightarrow{\sigma}]}{\overrightarrow{\Gamma} \vdash x[\overrightarrow{\sigma}, t] \approx t : T[\overrightarrow{\sigma}]}$$

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}; \Delta \qquad \overrightarrow{\Delta}; \Delta \vdash T' : \mathtt{Ty}_i \qquad \overrightarrow{\Gamma} \vdash t : T'[\overrightarrow{\sigma}] \qquad x : T \in \overrightarrow{\Delta}; \Delta}{\overrightarrow{\Gamma} \vdash x[\overrightarrow{\sigma}, t] \approx x[\overrightarrow{\sigma}] : T[\overrightarrow{\sigma}]}$$

The Π rules:
To describe how K-substitutions are propagated under different constructs, the following construct q weakens a K-substitution:

$$q_T(\overrightarrow{\sigma}) := (\overrightarrow{\sigma} \circ \mathsf{wk}), x$$

where the de Bruijn index of x is 0 and the subscript T is needed for the following typing rule:

$$\frac{\overrightarrow{\Gamma}; \Gamma \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}; \Delta \qquad \overrightarrow{\Delta}; \Delta \vdash T : \mathtt{Ty}_i}{\overrightarrow{\Gamma}; \Gamma, x : T[\overrightarrow{\sigma}] \vdash q_T(\overrightarrow{\sigma}) : \overrightarrow{\Delta}; \Delta, x : T}$$

This subscript is often omitted when it can be inferred from the surrounding textual context.

$$\frac{\vec{\Gamma}; \Gamma \vdash \vec{\sigma} : \vec{\Delta}; \Delta \qquad \vec{\Delta}; \Delta \vdash S : \mathtt{Ty}_i \qquad \vec{\Delta}; \Delta, x : S \vdash T : \mathtt{Ty}_i}{\vec{\Gamma}; \Gamma \vdash (\Pi(x : S).T)[\vec{\sigma}] \approx \Pi(x : S[\vec{\sigma}]).(T[q(\vec{\sigma})]) : \mathtt{Ty}_i}$$

$$\frac{\vec{\Gamma}; \Gamma \vdash \vec{\sigma} : \vec{\Delta}; \Delta \qquad \vec{\Delta}; \Delta, x : S \vdash t : T}{\vec{\Gamma}; \Gamma \vdash (\lambda x.t)[\vec{\sigma}] \approx \lambda x.(t[q(\vec{\sigma})]) : (\Pi(x : S).T)[\vec{\sigma}]}$$

$$\frac{\begin{array}{c}\vec{\Delta}; \Delta \vdash S : \mathtt{Ty}_i \\ \vec{\Delta}; \Delta, x : S \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}; \Delta \qquad \vec{\Delta}; \Delta \vdash s : \Pi(x : S).T \qquad \vec{\Delta}; \Delta \vdash t : S\end{array}}{\vec{\Gamma} \vdash s\ t[\vec{\sigma}] \approx (s[\vec{\sigma}])\ (t[\vec{\sigma}]) : T[\vec{\sigma}, t[\vec{\sigma}]]}$$

The `Nat` rules:

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vdash \mathtt{Nat}[\vec{\sigma}] \approx \mathtt{Nat} : \mathtt{Ty}_i} \qquad \frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vdash \mathsf{zero}[\vec{\sigma}] \approx \mathsf{zero} : \mathtt{Nat}}$$

$$\frac{\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta} \qquad \vec{\Delta} \vdash t : \mathtt{Nat}}{\vec{\Gamma} \vdash \mathsf{succ}\ t[\vec{\sigma}] \approx \mathsf{succ}\ (t[\vec{\sigma}]) : \mathtt{Nat}}$$

$$\frac{\begin{array}{c}\vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}; \Delta \qquad \vec{\Delta}; \Delta, x : \mathtt{Nat} \vdash M : \mathtt{Ty}_i \\ \vec{\Delta}; \Delta \vdash s : M[\vec{I}, \mathsf{zero}] \qquad \vec{\Delta}; \Delta, x : \mathtt{Nat}, y : M \vdash s' : M[(\mathsf{wk} \circ \mathsf{wk}), \mathsf{succ}\ v_1] \\ \vec{\Delta}; \Delta \vdash t : \mathtt{Nat} \qquad t' = \mathsf{elim}_{\mathtt{Nat}}\ (x.M[q(\vec{\sigma})])\ (s[\vec{\sigma}])\ (x, y.s'[q(q(\vec{\sigma}))])\ (t[\vec{\sigma}])\end{array}}{\vec{\Gamma} \vdash (\mathsf{elim}_{\mathtt{Nat}}\ (x.M)\ s\ (x, y.s')\ t)[\vec{\sigma}] \approx t' : M[\vec{\sigma}, t[\vec{\sigma}]]}$$

The $\square$ rules:

$$\frac{\vec{\Delta}; \cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vdash \square T[\vec{\sigma}] \approx \square(T[\vec{\sigma}; \Uparrow^1]) : \mathtt{Ty}_i} \qquad \frac{\vec{\Delta}; \cdot \vdash t : T \qquad \vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}}{\vec{\Gamma} \vdash \mathtt{box}\ t[\vec{\sigma}] \approx \mathtt{box}\ (t[\vec{\sigma}; \Uparrow^1]) : \square T[\vec{\sigma}]}$$

$$\frac{\vec{\Delta}; \cdot \vdash T : \mathtt{Ty}_i \qquad \vec{\Delta} \vdash t : \square T \qquad |\vec{\Delta}'| = n \qquad \vec{\Gamma} \vdash \vec{\sigma} : \vec{\Delta}; \vec{\Delta}'}{\vec{\Gamma} \vdash \mathtt{unbox}_n\ t[\vec{\sigma}] \approx \mathtt{unbox}_{\mathcal{O}(\vec{\sigma}, n)}\ (t[\vec{\sigma} \mid n]) : T[\vec{\sigma} \mid n; \Uparrow^{\mathcal{O}(\vec{\sigma}, n)}]}$$

The $\mathtt{Ty}$ rule:

$$\frac{\overrightarrow{\Gamma} \vdash \overrightarrow{\sigma} : \overrightarrow{\Delta}}{\overrightarrow{\Gamma} \vdash \mathtt{Ty}_i[\overrightarrow{\sigma}] \approx \mathtt{Ty}_i : \mathtt{Ty}_{1+i}}$$

Other rules:

$$\frac{\overrightarrow{\Gamma} \vdash t \approx t' : T}{\overrightarrow{\Gamma} \vdash t' \approx t : T} \qquad \frac{\overrightarrow{\Gamma} \vdash t \approx t' : T \qquad \overrightarrow{\Gamma} \vdash t' \approx t'' : T}{\overrightarrow{\Gamma} \vdash t \approx t'' : T}$$

$$\frac{\overrightarrow{\Gamma} \vdash t \approx t' : T \qquad \overrightarrow{\Gamma} \vdash T \approx T' : \mathtt{Ty}_i}{\overrightarrow{\Gamma} \vdash t \approx t' : T'} \qquad \frac{\overrightarrow{\Gamma} \vdash T \approx T' : \mathtt{Ty}_i}{\overrightarrow{\Gamma} \vdash T \approx T' : \mathtt{Ty}_{1+i}}$$

C
Substitutions for Layered Modal Type Theory

In this appendix, I define the operations on regular and meta-substitutions for Chapter 4. First, I define the application of regular substitutions on terms and their composition:

$$
\begin{aligned}
x[\delta] &:= \delta(x) \quad \text{(lookup of } x \text{ in } \delta) \\
u^{\delta'}[\delta] &:= u^{\delta' \circ \delta} \\
\mathsf{zero}[\delta] &:= \mathsf{zero} \\
\mathsf{succ}\ t[\delta] &:= \mathsf{succ}\ (t[\delta]) \\
\lambda x.t[\delta] &:= \lambda x.(t[\delta, x/x]) \\
t\ s[\delta] &:= (t[\delta])\ (s[\delta]) \\
\mathtt{box}\ t[\delta] &:= \mathtt{box}\ t \\
\mathtt{letbox}\ u \leftarrow s\ \mathtt{in}\ t[\delta] &:= \mathtt{letbox}\ u \leftarrow s[\delta]\ \mathtt{in}\ (t[\delta]) \\
\mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b}[\delta] &:= \mathsf{match}\ t[\delta]\ \mathsf{with}\ (\overrightarrow{b}[\delta]) \\
\Lambda g.t[\delta] &:= \Lambda g.(t[\delta])
\end{aligned}
$$

$$\begin{aligned} t \mathbin{\$} \Gamma[\delta] &:= (t[\delta]) \mathbin{\$} \Gamma \end{aligned}$$

$$\begin{aligned}
\mathsf{var}_x \Rightarrow t[\delta] &:= \mathsf{var}_x \Rightarrow (t[\delta]) \\
\mathsf{genvar}_{g,T} \Rightarrow t[\delta] &:= \mathsf{genvar}_{g,T} \Rightarrow (t[\delta]) \\
\mathsf{zero} \Rightarrow t[\delta] &:= \mathsf{zero} \Rightarrow (t[\delta]) \\
\mathsf{succ}\ ?u \Rightarrow t[\delta] &:= \mathsf{succ}\ ?u \Rightarrow (t[\delta]) \\
\lambda x.?u \Rightarrow t[\delta] &:= \lambda x.?u \Rightarrow (t[\delta]) \\
?u\ ?u' \Rightarrow t[\delta] &:= ?u\ ?u' \Rightarrow (t[\delta])
\end{aligned}$$

$$\begin{aligned}
\mathsf{wk} \circ \delta &:= \mathsf{wk} \\
\cdot \circ \delta &:= \cdot \\
(\delta', t/x) \circ \delta &:= (\delta' \circ \delta), t[\delta]/x
\end{aligned}$$

Regular substitutions do not propagate under `box`.

The following generalizes the weakening regular substitution wk to some arbitrary codomain regular context:

$$\begin{aligned}
\mathsf{wk}_{\cdot} &:= \cdot \\
\mathsf{wk}_g &:= \mathsf{wk} \\
\mathsf{wk}_{\Gamma, x:T} &:= \mathsf{wk}_\Gamma, x/x
\end{aligned}$$

This generalization satisfies:

Lemma C.1 (Typing of regular weakening). *If $\Psi \vdash_i \Delta, \Gamma$, then $\Psi; \Delta, \Gamma \vdash_i \mathsf{wk}_\Delta : \Delta$.*

The special case is id_Δ, which also fixes the domain regular context, so that the weakening substitution is in fact an identity substitution. I often omit the subscript as it does not cause any ambiguity.

The following defines the well-formedness of meta-substitutions:

$$\frac{\vdash \Psi}{\Psi \vdash \cdot : \cdot} \qquad \frac{\Psi \vdash \sigma : \Phi \qquad \Psi \vdash_0 \Gamma \qquad \Psi \vdash_0 T \qquad \Psi; \Gamma[\sigma] \vdash_0 t : T[\sigma]}{\Psi \vdash \sigma, t/u : \Phi, u : (\Gamma \vdash T)}$$

$$\frac{\Psi \vdash \sigma : \Phi \qquad \Psi \vdash_0 \Gamma}{\Psi \vdash \sigma, \Gamma/g : \Phi, g : \mathsf{Ctx}}$$

Meta-substitutions may be applied to types and regular contexts due to context variables. In particular, a substitution of a context variable replaces it with a concrete regular context.

$$\begin{aligned}
\texttt{Nat}[\sigma] &:= \texttt{Nat} \\
S \longrightarrow T[\sigma] &:= (S[\sigma]) \longrightarrow (T[\sigma]) \\
\Box(\Gamma \vdash T)[\sigma] &:= \Box(\Gamma[\sigma] \vdash T[\sigma]) \\
(g : \mathsf{Ctx}) \Rightarrow T[\sigma] &:= (g : \mathsf{Ctx}) \Rightarrow (T[\sigma, g/g])
\end{aligned}$$

$$\begin{aligned}
\cdot[\sigma] &:= \cdot \\
g[\sigma] &:= \sigma(g) \qquad \text{(lookup } g \text{ in } \sigma\text{)} \\
\Gamma, x : T[\sigma] &:= (\Gamma[\sigma]), x : (T[\sigma])
\end{aligned}$$

As a side note, α-renaming needs to occur when replacing regular contexts. When doing $\Delta[\Gamma/g]$ and variables in Γ clash with those in Δ, those in Γ are renamed properly and keep the names in Δ unchanged.

Similarly, I define the applications of meta-substitutions to terms and regular substitutions. Note that applying a meta-substitution may cause an application of regular substitutions. Unlike regular substitutions, meta-substitutions do propagate under `box`.

$$\begin{aligned}
x[\sigma] &:= x \\
u^{\delta}[\sigma] &:= \sigma(u)[\delta[\sigma]] && \text{(lookup of } u \text{ in } \sigma\text{)} \\
\mathsf{zero}[\sigma] &:= \mathsf{zero} \\
\mathsf{succ}\ t[\sigma] &:= \mathsf{succ}\ (t[\sigma]) \\
\lambda x.t[\sigma] &:= \lambda x.(t[\sigma]) \\
t\ s[\sigma] &:= (t[\sigma])\ (s[\sigma]) \\
\texttt{box}\ t[\sigma] &:= \texttt{box}\ (t[\sigma]) \\
\texttt{letbox}\ u \leftarrow s\ \texttt{in}\ t[\sigma] &:= \texttt{letbox}\ u \leftarrow s[\sigma]\ \texttt{in}\ (t[q(\sigma)]) \\
\mathsf{match}\ t\ \mathsf{with}\ \overrightarrow{b}[\sigma] &:= \mathsf{match}\ t[\sigma]\ \mathsf{with}\ (\overrightarrow{b}[\sigma]) \\
\Lambda g.t[\sigma] &:= \Lambda g.(t[\sigma, g/g]) \\
t\ \$\ \Gamma[\sigma] &:= (t[\sigma])\ \$\ (\Gamma[\sigma])
\end{aligned}$$

$$\begin{aligned}
\mathsf{var}_x \Rightarrow t[\sigma] &:= \mathsf{var}_x \Rightarrow (t[\sigma]) \\
\mathsf{zero} \Rightarrow t[\sigma] &:= \mathsf{zero} \Rightarrow (t[\sigma]) \\
\mathsf{succ}\ ?u \Rightarrow t[\sigma] &:= \mathsf{succ}\ ?u \Rightarrow (t[q(\sigma)]) \\
\lambda x.?u \Rightarrow t[\sigma] &:= \lambda x.?u \Rightarrow (t[q(\sigma)])
\end{aligned}$$

$$\begin{aligned}
?u\ ?u' \Rightarrow t[\sigma] &:= ?u\ ?u' \Rightarrow (t[q(q(\sigma))]) \\
\mathsf{wk}[\sigma] &:= \mathsf{wk}_{\sigma(g)} \qquad (g \text{ is the context variable in the codomain context}) \\
\cdot[\sigma] &:= \cdot \\
(\delta, t/x)[\sigma] &:= (\delta[\sigma]), t[\sigma]/x
\end{aligned}$$

where $q(\sigma) := \sigma, u^{\mathsf{id}}/u$.

Finally, I should handle the meta-substitution action for branches $\overrightarrow{b}$ with care. The idea here is to see if there exists a special branch for generated variables. If so, then by the typing rules, this branch must be expanded into multiple branches for regular variables.

$$\begin{aligned}
\overrightarrow{b}[\sigma] &:= \{b[\sigma] \text{ for all } b \in \overrightarrow{b}\} && (\text{if } \mathsf{genvar}_{g,T} \Rightarrow t \text{ is not in } \overrightarrow{b}) \\
\overrightarrow{b}[\sigma] &:= \{b[\sigma] \text{ for all } b \in \overrightarrow{b} \text{ and is not } \mathsf{genvar}_{g,T} \Rightarrow t, && (\text{otherwise, then } \sigma(g) = \Gamma) \\
&\qquad \mathsf{var}_x \Rightarrow (t[\sigma]) \text{ for all } x : T \in \Gamma, \\
&\qquad \mathsf{genvar}_{g',T} \Rightarrow (t[\sigma]) \text{ if } \Gamma = g', \Gamma'\}
\end{aligned}$$

In the first case, if there does not exist $\mathsf{genvar}_{g,T} \Rightarrow t$ at all, then it means that the current pattern matching is handling code from a concrete regular context with no context variable. It then suffices to just propagate σ to all branches within.

In the second case, $\mathsf{genvar}_{g,T} \Rightarrow t$ does exist, so σ is first acted on all other branches. Then the branch for generated variables should be instantiated to multiple branches for regular variables. If g is bound to Γ in σ, then for each $x : T \in \Gamma$, a branch $\mathsf{var}_x \Rightarrow (t[\sigma])$ is generated. At last, if Γ itself contains another context variable, then a new branch for generated variables should be generated in place of the original one. On the other hand, if Γ contains no context variable, then this new branch is entirely unnecessary and thus removed from the output.

D

Adding Recursor for Natural Numbers

Adding a recursor for natural numbers to layered modal type theory is a standard practice. The recursor has the following syntax:

$$\mathsf{rec}_T\ s\ (x, y.s')\ t$$

In this term, T is the motive and t is the scrutinee. There are two cases for this recursor. The base case where t computes to zero is handled by s. The step case is handled by s', where x is the predecessor and y is the recursive call.

In addition, there are the following rules:

$$\frac{\Psi; \Gamma \vdash_i s : T \qquad \Psi; \Gamma, x : \mathtt{Nat}, y : T \vdash_i s' : T \qquad \Psi; \Gamma \vdash_i t : \mathtt{Nat}}{\Psi; \Gamma \vdash_i \mathsf{rec}_T\ s\ (x, y.s')\ t : T}$$

$$\frac{\Psi; \Gamma \vdash_1 s : T \qquad \Psi; \Gamma, x : \mathtt{Nat}, y : T \vdash_i s' : T \qquad \Psi; \Gamma \vdash_1 t : \mathtt{Nat}}{\Psi; \Gamma \vdash_1 \mathsf{rec}_T\ s\ (x, y.s')\ \mathsf{zero} \approx s : T}$$

$$\frac{\Psi; \Gamma \vdash_1 s : T \qquad \Psi; \Gamma, x : \mathtt{Nat}, y : T \vdash_i s' : T \qquad \Psi; \Gamma \vdash_1 t : \mathtt{Nat}}{\Psi; \Gamma \vdash_1 \mathsf{rec}_T\ s\ (x, y.s')\ \mathsf{succ}\ t \approx s'[t/x, \mathsf{rec}_T\ s\ (x, y.s')\ t/y] : T}$$

and the following branch:

$$\frac{\Psi, u : (\Delta \vdash T), u' : (\Delta, x : \mathtt{Nat}, y : T \vdash T), u'' : (\Delta \vdash \mathtt{Nat}); \Gamma \vdash_1 t : T'}{\Psi; \Gamma \vdash_1 \mathsf{rec}_T\ ?u\ (x, y.?u')\ ?u'' \Rightarrow t : \Delta \vdash T \Rightarrow T'}$$

The following rule extends the β rule of pattern matching on code with the case of the recursor:

$$\frac{\begin{array}{c}\Psi; \Gamma \vdash_0 s : T \qquad \Psi; \Gamma, x : \mathtt{Nat}, y : T \vdash_0 s' : T \qquad \Psi; \Gamma \vdash_1 t' : \mathtt{Nat} \\ \Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T' \qquad \overrightarrow{b}(\mathsf{rec}_T\ s\ (x, y.s')\ t') = \mathsf{rec}_T\ ?u\ (x, y.?u')\ ?u'' \Rightarrow t\end{array}}{\Psi; \Gamma \vdash_1 \mathsf{match\ box}\ (\mathsf{rec}_T\ s\ (x, y.s')\ t')\ \mathsf{with}\ \overrightarrow{b} \approx t[s/u, s'/u', t'/u''] : T'}$$

Since the recursor is a elimination principle, it could have any return type, so $\Psi; \Gamma \vdash_1 \overrightarrow{b} : \Delta \vdash T \Rightarrow T'$ must be extended with a case for the recursor for each coverage rule.

The one-step weak-head reduction is extended with the β rules above. The neutral forms are extended with one extra for stuck recursors: $\mathsf{rec}_T\ s\ (x, y.s')\ v$.

E

Conversion Checking for Neutral Pattern Matching

This appendix complements the missing rules in Sec. 4.10.

$$\frac{\Psi \vdash_1 T' \qquad \Psi; \Gamma \vdash_1 v \longleftrightarrow v' : \Box(\Delta \vdash T) \qquad \Psi; \Gamma \vdash_1 \overrightarrow{b} \mathrel{\hat{\Longleftrightarrow}} \overrightarrow{b}' : \Delta \vdash T \Rightarrow T'}{\Psi; \Gamma \vdash_1 \mathsf{match}\ v\ \mathsf{with}\ \overrightarrow{b} \longleftrightarrow \mathsf{match}\ v'\ \mathsf{with}\ \overrightarrow{b}' : T'}$$

$$\frac{\Psi \vdash_0 g, \Delta \qquad \Psi; \Gamma \vdash_1 t \Longleftrightarrow t' : T'}{\Psi; \Gamma \vdash_1 \mathsf{genvar}_{g,T} \Rightarrow t \mathrel{\hat{\Longleftrightarrow}} \mathsf{genvar}_{g,T} \Rightarrow t' : g, \Delta \vdash T \Rightarrow T'}$$

$$\frac{\Psi \vdash_0 \Delta \qquad x : T \in \Delta \qquad \Psi; \Gamma \vdash_1 t \Longleftrightarrow t' : T'}{\Psi; \Gamma \vdash_1 \mathsf{var}_x \Rightarrow t \mathrel{\hat{\Longleftrightarrow}} \mathsf{var}_x \Rightarrow t' : \Delta \vdash T \Rightarrow T'}$$

$$\frac{\Psi \vdash_0 \Delta \qquad \Psi; \Gamma \vdash_1 t \mathrel{\hat{\Longleftrightarrow}} t' : T'}{\Psi; \Gamma \vdash_1 \mathsf{zero} \Rightarrow t \mathrel{\hat{\Longleftrightarrow}} \mathsf{zero} \Rightarrow t' : \Delta \vdash \mathtt{Nat} \Rightarrow T'}$$

$$\frac{\Psi, u : (\Delta \vdash \mathtt{Nat}); \Gamma \vdash_1 t \stackrel{\hat{}}{\Longleftrightarrow} t' : T'}{\Psi; \Gamma \vdash_1 \mathsf{succ}\ ?u \Rightarrow t \stackrel{\hat{}}{\Longleftrightarrow} \mathsf{succ}\ ?u \Rightarrow t' : \Delta \vdash \mathtt{Nat} \Rightarrow T'}$$

$$\frac{\Psi, u : (\Delta, x : S \vdash T); \Gamma \vdash_1 t \stackrel{\hat{}}{\Longleftrightarrow} t' : T'}{\Psi; \Gamma \vdash_1 \lambda x.?u \Rightarrow t \stackrel{\hat{}}{\Longleftrightarrow} \lambda x.?u \Rightarrow t' : \Delta \vdash S \longrightarrow T \Rightarrow T'}$$

$$\frac{\forall\ \Psi \vdash_0 S\ .\ \Psi, u : (\Delta \vdash S \longrightarrow T), u' : (\Delta \vdash S); \Gamma \vdash_1 t \stackrel{\hat{}}{\Longleftrightarrow} t' : T'}{\Psi; \Gamma \vdash_1 ?u\ ?u' \Rightarrow t \stackrel{\hat{}}{\Longleftrightarrow} ?u\ ?u' \Rightarrow t' : \Delta \vdash T \Rightarrow T'}$$

F

Well-formedness and Reductions of Branches

In this section, I list all the well-formedness conditions for branches. The discussion in this appendix complements Sec. 5.3.6. Recall that the well-formedness of the motives are:

$$L, \ell \mid \Psi, u_\Gamma : \mathsf{Ctx}; \Gamma, x_T : \square(u_\Gamma \vdash_{\mathrm{C}} \ @\ \ell) \ @\ 0 \vdash_{\mathrm{M}} M_{\mathsf{Typ}} \ @\ l_1$$

$$L, \ell \mid \Psi, u_\Gamma : \mathsf{Ctx}, u_T : (u_\Gamma \vdash_{\mathrm{D}} \ @\ \ell); \Gamma, x_t : \square(u_\Gamma \vdash_{\mathrm{C}} u_T^{\mathsf{id}} \ @\ \ell) \ @\ 0 \vdash_{\mathrm{M}} M_{\mathsf{Trm}} \ @\ l_2$$

F.1 Branches for Types

F.1.1 Type of Universes

The typing rule is

$$\frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma}{L \mid \Psi; \Gamma \vdash_i \mathtt{Ty}_l \ @\ 1 + l}$$

The branch in the recursion principle $t_{\mathtt{Ty}}$ is

$$L, \ell \mid \Psi, u_\Gamma : \mathsf{Ctx}; \Gamma \vdash_{\mathrm{M}} t_{\mathtt{Ty}} : M_{\mathsf{Typ}}[1 + \ell/\ell, u_\Gamma/u_\Gamma, \mathtt{box}\ \mathtt{Ty}_\ell/x_T] \ @\ l_1$$

The reduction rule is

$$\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ \mathtt{Ty}_l : \square(\Gamma \vdash_{\mathrm{C}}\ @\ l)) \rightsquigarrow t_{\mathtt{Nat}}[l/\ell, \Gamma/u_\Gamma]$$

The rest of the appendix will follow this pattern to give all well-formedness conditions and reduction rules.

F.1.2 Type of Natural Numbers

The typing rule is

$$\frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma}{L \mid \Psi; \Gamma \vdash_i \mathtt{Nat} \ @\ 0}$$

The branch in the recursion principle $t_{\mathtt{Nat}}$ is

$$L \mid \Psi, u_\Gamma : \mathsf{Ctx}; \Gamma \vdash_{\mathrm{M}} t_{\mathtt{Nat}} : M_{\mathsf{Typ}}[0/\ell, u_\Gamma/u_\Gamma, \mathtt{box}\ \mathtt{Nat}/x_T] \ @\ l_1$$

The reduction rule is

$$\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ \mathtt{Nat} : \square(\Gamma \vdash_{\mathrm{C}}\ @\ 0)) \rightsquigarrow t_{\mathtt{Nat}}[\Gamma/u_\Gamma]$$

F.1.3 Π Types

The typing rule is

$$\frac{L \mid \Psi; \Gamma \vdash_i S \mathbin{@} l \qquad L \mid \Psi; \Gamma, x : S \mathbin{@} l \vdash_i T \mathbin{@} l'}{L \mid \Psi; \Gamma \vdash_i \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l'}$$

The rule for the branch t_Π is

$$\begin{aligned}
&L, \ell, \ell' \\
&\mid \Psi, u_\Gamma : \mathsf{Ctx}, u_S : (u_\Gamma \vdash_{\mathrm{C}} \mathbin{@} \ell), u_T : (u_\Gamma, x : u_S^{\mathsf{id}} \mathbin{@} \ell \vdash_{\mathrm{C}} \mathbin{@} \ell') \\
&\qquad\qquad \text{(meta-assumptions to model the typing rule)} \\
&; \Gamma, x_S : M_{\mathsf{Typ}}[\ell/\ell, u_\Gamma/u_\Gamma, \mathtt{box}\ u_S^{\mathsf{id}}/x_T] \mathbin{@} l_1 \\
&, x_T : M_{\mathsf{Typ}}[\ell'/\ell, (u_\Gamma, x : u_S^{\mathsf{id}} \mathbin{@} \ell)/u_\Gamma, \mathtt{box}\ u_T^{\mathsf{id}}/x_T] \mathbin{@} l_1 \\
&\qquad\qquad \text{(regular assumptions to for the recursive calls)} \\
&\vdash_{\mathrm{M}} t_\Pi : M_{\mathsf{Typ}}[\ell \sqcup \ell'/\ell, u_\Gamma/u_\Gamma, \mathtt{box}\ (\Pi^{\ell,\ell'}(x : u_S^{\mathsf{id}}).u_T^{\mathsf{id}})/x_T] \mathbin{@} l_1
\end{aligned}$$

The reduction rule is

$$\begin{aligned}
&\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ (\Pi^{l,l'}(x : S).T) : \square(\Gamma \vdash_{\mathrm{C}} \mathbin{@} l \sqcup l')) \\
\rightsquigarrow\ & t_\Pi[l/\ell, l'/\ell, \Gamma/u_\Gamma, S/u_S, T/u_T, \\
&\quad \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ S : \square(\Gamma \vdash_{\mathrm{C}} \mathbin{@} l))/x_S, \\
&\quad \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ T : \square(\Gamma, x : S \mathbin{@} l \vdash_{\mathrm{C}} \mathbin{@} l'))/x_T]
\end{aligned}$$

F.1.4 Decoder `El`

The typing rule is

$$\frac{L \mid \Psi; \Gamma \vdash_i t : \mathtt{Ty}_l \mathbin{@} 1 + l}{L \mid \Psi; \Gamma \vdash_i \mathtt{El}^l\ t \mathbin{@} l}$$

The rule for the branch $t_{\mathtt{El}}$ is

$$\begin{aligned}
&L, \ell \\
&\mid \Psi, u_\Gamma : \mathsf{Ctx}, u_t : (u_\Gamma \vdash_{\mathrm{C}} \mathtt{Ty}_\ell \mathbin{@} 1 + \ell) \\
&; \Gamma, x_t : M_{\mathsf{Trm}}[1 + \ell/\ell, u_\Gamma/u_\Gamma, \mathtt{Ty}_\ell/u_T, \mathtt{box}\ u_t^{\mathsf{id}}/x_t] \mathbin{@} l_2 \\
&\vdash_{\mathrm{M}} t_{\mathtt{El}} : M_{\mathsf{Typ}}[\ell/\ell, u_\Gamma/u_\Gamma, \mathtt{box}\ (\mathtt{El}^\ell\ u_t^{\mathsf{id}})/x_T] \mathbin{@} l_1
\end{aligned}$$

The reduction rule is

$$\begin{aligned}&\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\texttt{box}\ (\texttt{El}^l\ t) : \square(\Gamma \vdash_{\text{C}} @\ l))\\ \rightsquigarrow\ & t_{\texttt{El}}[l/\ell, \Gamma/u_\Gamma, t/u_t,\\ &\quad \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\texttt{box}\ t : \square(\Gamma \vdash_{\text{C}} \texttt{Ty}_l\ @\ 1+l))/x_t]\end{aligned}$$

Note that to substitute x_t, the recursor for code of terms is invoked, so both recursors for code of types and terms are indeed mutual.

F.2 Branches for Terms

F.2.1 Variables

The typing rule is

$$\frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma \qquad x : T\ @\ l \in \Gamma}{L \mid \Psi; \Gamma \vdash_i x : T\ @\ l}$$

The rule for the branch t_x is

$$\begin{aligned}&L, \ell\\ &\mid \Psi, u_\Gamma : \mathsf{Ctx}, u_T : (u_\Gamma \vdash_{\text{D}}\ @\ \ell), u_x : (u_\Gamma \vdash_{\text{V}} u_T^{\mathsf{id}}\ @\ \ell)\\ &;\ \Gamma\\ \vdash_{\text{M}}\ & t_x : M_{\mathsf{Trm}}[\ell/\ell, u_\Gamma/u_\Gamma, u_T^{\mathsf{id}}/u_T, \texttt{box}\ u_x^{\mathsf{id}}/x_t]\ @\ l_2\end{aligned}$$

In this case, there are two new kinds of assumptions in the meta-context. Firstly, u_T denotes the type of the variable. It is at layer D, because it is obtained externally, and it is not code. Since it is not at layer C, it cannot be recursed on, so the regular context Γ has no additional assumption. Secondly, the assumption u_x is at layer V. This is the only place where the layer V is actually used. The layer V corresponds to a sub-language merely with variables and is lower than C. Therefore, by lifting, code at layer V can be lifted to C. I use layer V here for the fact that u_x represents code for a variable. If hypothetically I use layer C instead, then u_x can literally be substituted for arbitrary code, which is not an intended behavior.

The reduction rule is

$$\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\texttt{box}\ x : \square(\Gamma \vdash_{\text{C}} W\ @\ l)) \rightsquigarrow t_x[l/\ell, \Gamma/u_\Gamma, x/u_x]$$

Note here u_x is replaced by x, which is a variable. Also the type is W, i.e. a WHNF, because I choose to first reduce the type of the crutinee to a WHNF first.

F.2.2 Encoding of Universes

In this case, let us consider the encoding of universes as a term. The typing rule is

$$\frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma}{L \mid \Psi; \Gamma \vdash_i \mathtt{Ty}_l : \mathtt{Ty}_{1+l} \mathbin{@} 2+l}$$

The branch in the recursion principle $t'_{\mathtt{Ty}}$ is

$$L, \ell \mid \Psi, u_\Gamma : \mathsf{Ctx}; \Gamma \vdash_{\mathrm{M}} t'_{\mathtt{Ty}} : M_{\mathsf{Trm}}[2+\ell/\ell, u_\Gamma/u_\Gamma, \mathtt{Ty}_{1+\ell}/u_T, \mathtt{box}\ \mathtt{Ty}_\ell/x_t] \mathbin{@} l_2$$

I use $2+\ell$ to replace ℓ from M_{Trm}, because this is the universe level for $\mathtt{Ty}_{1+\ell}$, which is the type of $\mathtt{Ty}_\ell$. Therefore, the universe level goes up by 2.

The reduction rule is

$$\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ \mathtt{Ty}_l : \square(\Gamma \vdash_{\mathrm{C}} \mathtt{Ty}_{1+l} \mathbin{@} 2+l)) \rightsquigarrow t'_{\mathtt{Ty}}[l/\ell, \Gamma/u_\Gamma]$$

F.2.3 Encoding of Natural Numbers

The typing rule is

$$\frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma}{L \mid \Psi; \Gamma \vdash_i \mathtt{Nat} : \mathtt{Ty}_0 \mathbin{@} 1}$$

The branch in the recursion principle $t'_{\mathtt{Nat}}$ is

$$L \mid \Psi, u_\Gamma : \mathsf{Ctx}; \Gamma \vdash_{\mathrm{M}} t'_{\mathtt{Nat}} : M_{\mathsf{Trm}}[1/\ell, u_\Gamma/u_\Gamma, \mathtt{Ty}_0/u_T, \mathtt{box}\ \mathtt{Nat}/x_t] \mathbin{@} l_2$$

The reduction rule is

$$\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ \mathtt{Nat} : \square(\Gamma \vdash_{\mathrm{C}} \mathtt{Ty}_0 \mathbin{@} 1)) \rightsquigarrow t'_{\mathtt{Nat}}[\Gamma/u_\Gamma]$$

F.2.4 Encoding of Π Types

The typing rule is

$$\frac{L \mid \Psi; \Gamma \vdash_i s : \mathtt{Ty}_l \mathbin{@} 1 + l \qquad L \mid \Psi; \Gamma, x : \mathtt{El}^l\ s \mathbin{@} l \vdash_i t : \mathtt{Ty}_{l'} \mathbin{@} 1 + l'}{L \mid \Psi; \Gamma \vdash_i \Pi^{l,l'}(x : s).t : \mathtt{Ty}_{l \sqcup l'} \mathbin{@} 1 + (l \sqcup l')}$$

The rule for the branch t'_Π is

$$\begin{array}{l} \quad L, \ell, \ell' \\ \quad \mid \Psi, u_\Gamma : \mathsf{Ctx}, u_s : (u_\Gamma \vdash_{\mathrm{C}} \mathtt{Ty}_\ell \mathbin{@} 1 + \ell), u_t : (u_\Gamma, x : \mathtt{El}^\ell\ u_s^{\mathsf{id}} \mathbin{@} \ell \vdash_{\mathrm{C}} \mathtt{Ty}_{\ell'} \mathbin{@} 1 + \ell') \\ \quad ; \Gamma, x_s : M_{\mathsf{Trm}}[1 + \ell/\ell, u_\Gamma/u_\Gamma, \mathtt{Ty}_\ell/u_T, \mathtt{box}\ u_s^{\mathsf{id}}/x_T] \mathbin{@} l_2 \\ \quad , x_t : M_{\mathsf{Trm}}[1 + \ell'/\ell, (u_\Gamma, x : \mathtt{El}^\ell\ u_s^{\mathsf{id}} \mathbin{@} \ell)/u_\Gamma, \mathtt{Ty}_{1+\ell'}/u_T, \mathtt{box}\ u_t^{\mathsf{id}}/x_t] \mathbin{@} l_2 \\ \vdash_{\mathrm{M}} t'_\Pi : M_{\mathsf{Trm}}[1 + (\ell \sqcup \ell')/\ell, u_\Gamma/u_\Gamma, \mathtt{Ty}_{\ell \sqcup \ell'}/u_T, \mathtt{box}\ (\Pi^{\ell,\ell'}(x : u_s^{\mathsf{id}}).u_t^{\mathsf{id}})/x_t] \mathbin{@} l_2 \end{array}$$

The reduction rule is

$$\begin{array}{l} \quad \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ (\Pi^{l,l'}(x : s).t) : \square(\Gamma \vdash_{\mathrm{C}} \mathtt{Ty}_{l \sqcup l'} \mathbin{@} 1 + (l \sqcup l'))) \\ \rightsquigarrow t'_\Pi[l/\ell, l'/\ell, \Gamma/u_\Gamma, s/u_s, t/u_t, \\ \qquad \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ s : \square(\Gamma \vdash_{\mathrm{C}} \mathtt{Ty}_l \mathbin{@} 1 + l))/x_s, \\ \qquad \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ t : \square(\Gamma, x : \mathtt{El}^l\ s \mathbin{@} l \vdash_{\mathrm{C}} \mathtt{Ty}_{l'} \mathbin{@} 1 + l'))/x_t] \end{array}$$

F.2.5 Zero Case

The typing rule is

$$\frac{L \mid \Psi \vdash_{\Uparrow(i)} \Gamma}{L \mid \Psi; \Gamma \vdash_i \mathsf{zero} : \mathtt{Nat} \mathbin{@} 0}$$

The rule for the branch t_{zero} is

$$L \mid \Psi, u_\Gamma : \mathsf{Ctx}; \Gamma \vdash_{\mathrm{M}} t_{\mathsf{zero}} : M_{\mathsf{Trm}}[0/\ell, u_\Gamma/u_\Gamma, \mathtt{Nat}/u_T, \mathtt{box}\ \mathsf{zero}/x_t] \mathbin{@} l_2$$

The reduction rule is

$$\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ \mathsf{zero} : \square(\Gamma \vdash_{\mathrm{C}} \mathtt{Nat} \mathbin{@} 0)) \rightsquigarrow t_{\mathsf{zero}}[\Gamma/u_\Gamma]$$

F.2.6 Successor Case

The typing rule is

$$\frac{L \mid \Psi; \Gamma \vdash_i t : \texttt{Nat} \mathbin{@} 0}{L \mid \Psi; \Gamma \vdash_i \mathsf{succ}\ t : \texttt{Nat} \mathbin{@} 0}$$

The rule for the branch t_{succ} is

$$\begin{aligned}
&L \mid \Psi, u_\Gamma : \mathsf{Ctx}, u_t : (u_\Gamma \vdash_{\mathrm{C}} \texttt{Nat} \mathbin{@} 0)\\
&; \Gamma, x_t : M_{\mathsf{Trm}}[0/\ell, u_\Gamma/u_\Gamma, \texttt{Nat}/u_T, \texttt{box}\ u_t^{\mathsf{id}}/x_t] \mathbin{@} l_2\\
\vdash_{\mathrm{M}}\ & t_{\mathsf{succ}} : M_{\mathsf{Trm}}[0/\ell, u_\Gamma/u_\Gamma, \texttt{Nat}/u_T, \texttt{box}\ (\mathsf{succ}\ u_t^{\mathsf{id}})/x_t] \mathbin{@} l_2
\end{aligned}$$

The reduction rule is

$$\begin{aligned}
&\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\texttt{box}\ (\mathsf{succ}\ t) : \Box(\Gamma \vdash_{\mathrm{C}} \texttt{Nat} \mathbin{@} 0))\\
\rightsquigarrow\ & t_{\mathsf{succ}}[\Gamma/u_\Gamma, t/u_t, \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\texttt{box}\ t : \Box(\Gamma \vdash_{\mathrm{C}} \texttt{Nat} \mathbin{@} 0))/x_t]
\end{aligned}$$

F.2.7 Function Abstractions

The typing rule is

$$\frac{L \mid \Psi; \Gamma \vdash_i S \mathbin{@} l \qquad L \mid \Psi; \Gamma, x : S \mathbin{@} l \vdash_i t : T \mathbin{@} l'}{L \mid \Psi; \Gamma \vdash_i \lambda^{l,l'}(x : S).t : \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l'}$$

The rule for the branch t_λ is

$$\begin{aligned}
&L, \ell, \ell'\\
&\mid \Psi, u_\Gamma : \mathsf{Ctx}\\
&,\ \ u_S : (u_\Gamma \vdash_{\mathrm{C}}\ \mathbin{@} \ell)\\
&,\ \ u_T : (u_\Gamma, x : u_S^{\mathsf{id}} \mathbin{@} \ell \vdash_{\mathrm{D}}\ \mathbin{@} \ell')\\
&,\ \ u_t : (u_\Gamma, x : u_S^{\mathsf{id}} \mathbin{@} \ell \vdash_{\mathrm{C}} u_T^{\mathsf{id}} \mathbin{@} \ell')\\
&; \Gamma, x_S : M_{\mathsf{Typ}}[\ell/\ell, u_\Gamma/u_\Gamma, \texttt{box}\ u_S^{\mathsf{id}}/x_T] \mathbin{@} l_1\\
&,\ \ x_t : M_{\mathsf{Trm}}[\ell'/\ell, (u_\Gamma, x : u_S^{\mathsf{id}} \mathbin{@} \ell)/u_\Gamma, u_T^{\mathsf{id}}/u_T, \texttt{box}\ u_t^{\mathsf{id}}/x_t] \mathbin{@} l_2\\
\vdash_{\mathrm{M}}\ & t_\lambda : M_{\mathsf{Trm}}[\ell \sqcup \ell'/\ell, u_\Gamma/u_\Gamma, \Pi^{\ell,\ell'}(x : u_S^{\mathsf{id}}).u_T^{\mathsf{id}}/u_T, \texttt{box}\ (\lambda^{\ell,\ell'}(x : u_S^{\mathsf{id}}).u_t^{\mathsf{id}})/x_t] \mathbin{@} l_2
\end{aligned}$$

In the meta-context, other than u_S and u_t which represent corresponding sub-structures, I also have u_T which represents the return type of the function in an extended regular context. Looking at the typing rule, however, the return type T is not ascribed and is not a sub-structure. Therefore, u_T lives at layer D and there is not a recursive call for it, i.e. no x_T in the regular context. If in the core syntax, I choose to ascribe T as part of the syntax of λ, then I can change this layer from D to C and add a recursive call in the regular assumption. This is a flexibility that I can take; however, in this thesis, I simply choose not to include T as a sub-structure.

The reduction rule is

$$\begin{aligned}
&\mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ (\lambda^{l,l'}(x:S).t) : \square(\Gamma \vdash_{\mathrm{C}} \Pi^{l,l'}(x:S).T \mathbin{@} l \sqcup l'))\\
\rightsquigarrow\ & t_\Pi[l/\ell, l'/\ell, \Gamma/u_\Gamma, S/u_S, T/u_T, t/u_t,\\
&\quad \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ S : \square(\Gamma \vdash_{\mathrm{C}} \mathbin{@} l))/x_S,\\
&\quad \mathsf{elim}^{l_1,l_2}\ \overrightarrow{M}\ \overrightarrow{b}\ (\mathtt{box}\ t : \square(\Gamma, x : S \mathbin{@} l \vdash_{\mathrm{C}} T \mathbin{@} l'))/x_t]
\end{aligned}$$

G

Semantic Judgments for Code

The semantic judgment for code of types is defined as

$$\frac{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\geq\text{D}}} \texttt{Ty}_l \mathbin{@} 1+l}{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\text{C}}} \texttt{Ty}_l \mathbin{@} 1+l} \qquad \frac{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\geq\text{D}}} \texttt{Nat} \mathbin{@} 0}{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\text{C}}} \texttt{Nat} \mathbin{@} 0} \qquad \frac{\begin{array}{c} L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\text{C}}} t : \texttt{Ty}_l \mathbin{@} 1+l \\ L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\geq\text{D}}} \texttt{El}^l\ t \mathbin{@} l \end{array}}{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\text{C}}} \texttt{El}^l\ t \mathbin{@} l}$$

$$\frac{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\text{C}}} S \mathbin{@} l \qquad L \mid \Psi; \Gamma, x : S \mathbin{@} l \mathrel{\Vert\!\!\vDash^{\text{D}}_{\text{C}}} T \mathbin{@} l' \qquad L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\geq\text{D}}} \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l'}{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\text{C}}} \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l'}$$

$$\frac{u : (\Delta \vdash_{\text{C}} \ \mathbin{@} l) \in \Psi \qquad L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\text{C}}} \delta : \Delta \qquad L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\geq\text{D}}} u^{\delta} \mathbin{@} l}{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{\text{C}}} u^{\delta} \mathbin{@} l}$$

When I define the semantic judgments for code, I am also concerned about the semantics at layer V. Indeed, the semantics for code of variables at layer V is just a special case of the semantic judgments for code. When I define these judgments, I actually parameterize both $\boxed{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{i}} T \mathbin{@} l}$ and $\boxed{L \mid \Psi; \Gamma \mathrel{\Vert\!\!\vDash^{\text{D}}_{i}} t : T \mathbin{@} l}$, where $i \in \{\text{V}, \text{C}\}$, so when $i = \text{V}$, I also give the semantics for code of variables.

The semantic judgment for code of regular substitutions is

$$\frac{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} \cdot : \cdot}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{i} \cdot : \cdot} \qquad \frac{L \mid \Psi; u, \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} \mathsf{wk} : u}{L \mid \Psi; u, \Gamma \Vert\!\vDash^{\mathrm{D}}_{i} \mathsf{wk} : u} \qquad \frac{\begin{array}{c} L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{i} \delta : \Delta \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{i} t : T[\delta] \mathrel{@} l \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} \delta, t/x : \Delta, x : T \mathrel{@} l \end{array}}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{i} \delta, t/x : \Delta, x : T \mathrel{@} l}$$

Finally, I give the semantic judgment for code of terms. The following two rules are the only cases where V is a possible value for i:

$$\frac{\begin{array}{c} x : T \mathrel{@} l \in \Gamma \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} T' \approx T \mathrel{@} l \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} x : T' \mathrel{@} l \end{array}}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{i} x : T' \mathrel{@} l} \qquad \frac{\begin{array}{c} u : (\Delta \vdash_{i'} T \mathrel{@} l) \in \Psi \qquad i' \leq i \qquad L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{i} \delta : \Delta \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} T' \approx T[\delta] \mathrel{@} l \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} u^{\delta} : T' \mathrel{@} l \end{array}}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{i} u^{\delta} : T' \mathrel{@} l}$$

All other rules only take $i = \mathrm{C}$, as they are variables:

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} T \approx \mathtt{Ty}_{1+l} \mathrel{@} 2 + l \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} \mathtt{Ty}_{l} : T \mathrel{@} 2 + l \end{array}}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} \mathtt{Ty}_{l} : T \mathrel{@} 2 + l} \qquad \frac{\begin{array}{c} L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} T \approx \mathtt{Ty}_{0} \mathrel{@} 1 \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} \mathtt{Nat} : T \mathrel{@} 1 \end{array}}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} \mathtt{Nat} : T \mathrel{@} 1}$$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} T \approx \mathtt{Nat} \mathrel{@} 0 \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} \mathsf{zero} : T \mathrel{@} 0 \end{array}}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} \mathsf{zero} : T \mathrel{@} 0} \qquad \frac{\begin{array}{c} L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} t : \mathtt{Nat} \mathrel{@} 0 \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} T \approx \mathtt{Nat} \mathrel{@} 0 \qquad L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} \mathsf{succ}\ t : T \mathrel{@} 0 \end{array}}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} \mathsf{succ}\ t : T \mathrel{@} 0}$$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} s : \mathtt{Ty}_{l} \mathrel{@} 1 + l \qquad L \mid \Psi; \Gamma, x : \mathtt{El}^{l}\ S \mathrel{@} l \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} t : \mathtt{Ty}_{l'} \mathrel{@} 1 + l' \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} T \approx \mathtt{Ty}_{l \sqcup l'} \mathrel{@} 1 + (l \sqcup l') \qquad L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} \Pi^{l,l'}(x : s).t : T \mathrel{@} 1 + (l \sqcup l') \end{array}}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} \Pi^{l,l'}(x : s).t : \mathtt{Ty}_{l \sqcup l'} \mathrel{@} 1 + (l \sqcup l')}$$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} S \mathrel{@} l \qquad L \mid \Psi; \Gamma, x : S \mathrel{@} l \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} t : T \mathrel{@} l' \\ L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} T' \approx \Pi^{l,l'}(x : S).T \mathrel{@} l \sqcup l' \qquad L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\geq \mathrm{D}} \lambda^{l,l'}(x : S).t : T' \mathrel{@} l \sqcup l' \end{array}}{L \mid \Psi; \Gamma \Vert\!\vDash^{\mathrm{D}}_{\mathrm{C}} \lambda^{l,l'}(x : S).t : T' \mathrel{@} l \sqcup l'}$$

$$\frac{\begin{array}{c} L \mid \Psi; \Gamma \Vdash\!\models^{\mathrm{D}}_{\mathrm{C}} S \mathbin{@} l \qquad L \mid \Psi; \Gamma, x : S \mathbin{@} l \Vdash\!\models^{\mathrm{D}}_{\mathrm{C}} T \mathbin{@} l' \\ L \mid \Psi; \Gamma \Vdash\!\models^{\mathrm{D}}_{\mathrm{C}} t : \Pi^{l,l'}(x : S).T \mathbin{@} l \sqcup l' \qquad L \mid \Psi; \Gamma \Vdash\!\models^{\mathrm{D}}_{\mathrm{C}} s : S \mathbin{@} l \\ L \mid \Psi; \Gamma \Vdash\!\models^{\mathrm{D}}_{\geq \mathrm{D}} T' \approx T[s/x] \mathbin{@} l' \qquad L \mid \Psi; \Gamma \Vdash\!\models^{\mathrm{D}}_{\geq \mathrm{D}} (t : \Pi^{l,l'}(x : S).T)\ s : T' \mathbin{@} l' \end{array}}{L \mid \Psi; \Gamma \Vdash\!\models^{\mathrm{D}}_{\mathrm{C}} (t : \Pi^{l,l'}(x : S).T)\ s : T' \mathbin{@} l'}$$