UNIVERSITY OF LJUBLJANA
FACULTY OF MATHEMATICS AND PHYSICS
DEPARTMENT OF MATHEMATICS

Mathematics – 3rd cycle

Anja Petković Komel

META-ANALYSIS OF TYPE THEORIES WITH AN APPLICATION TO THE DESIGN OF FORMAL PROOFS

Doctoral thesis

Adviser: prof. dr. Andrej Bauer

Ljubljana, 2021

UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO
ODDELEK ZA MATEMATIKO

Matematika – 3. stopnja

Anja Petković Komel

META-ANALIZA TEORIJ TIPOV Z UPORABO V OBLIKOVANJU FORMALNIH DOKAZOV

Doktorska disertacija

Mentor: prof. dr. Andrej Bauer

Ljubljana, 2021

Abstract

In this thesis we present a meta-analysis of a wide class of general type theories, focusing on three aspects: transformations of type theories, elaboration of type theories, and a general equality checking algorithm.

Type theories provide the mathematical foundations of many proof assistants. We build towards understanding of how they interact by studying their meta-theoretic properties and checking them against an implementation of the flexible proof assistant Andromeda 2, which supports user-specified type theories.

Our meta-analysis is built on the definition of finitary type theories. We define *syntactic transformations* of type theories and prove they form a relative monad for the syntax. To account for the derivability structure, we upgrade the definition to *type-theoretic transformations* that cover some familiar examples, like propositions as types translation and the definitional extension. Once these definitions are accomplished we prove some meta-theorems. The usefulness of type-theoretic transformations is unveiled in the definition of an elaboration and we prove an elaboration theorem, saying that every finitary type theory has an elaboration.

To tackle the implementational side, we design a general and user-extensible equality checking algorithm, applicable to a finitary type theories. The algorithm is composed of a type-directed phase for applying extensionality rules and a normalization phase based on computation rules. Both kinds of rules are defined using the type-theoretic concept of object-invertible rules. We specify sufficient syntactic criteria for recognizing such rules and a simple pattern-matching algorithm for applying them. A third component of the algorithm is a suitable notion of principal arguments, which determines a notion of normal form. By varying these, we obtain known notions, such as weak head-normal and strong normal forms. We prove that our algorithm is sound. We implemented it in the Andromeda 2 proof assistant.

2020 Mathematics Subject Classification: 03B38, 03B70, 18C10, 68V15, 68V20, 03F50, 03F03, 03F07

Keywords: Dependent type theory, algebraic theory, proof assistant, type-theoretic elaboration equality checking

Izvleček

V doktorski disretaciji predstavimo meta-analizo širokega razreda splošnih teorij tipov. Osredotočimo se na tri vidike: transformacije teorij tipov, dopolnitev teorij tipov in splošen algoritem za preverjanje enakosti.

Teorije tipov zagotavljajo matematično osnovo za mnoge dokazovalne pomočnike. Da bi lažje razumeli interakcije med teorijami, študiramo njihove meta-teoretične lastnosti in jih preverjamo z implementacijo fleksibilnega dokazovalnega pomočnika Andromeda 2, ki omogoča, da uporabnik sam poda teorijo tipov.

Naša meta-analiza je zgrajena na definiciji končnih teorij tipov. Definiramo *sintaktične transformacije* teorij tipov in dokažemo, da tvorijo relativno monado za sintakso. Da bi vključili strukturo izpeljivosti nadgradimo definicijo v *transformacije teorij tipov*, ki pokrije nekatere znane primere, kot sta Curry-Howardova korespondenca in razširitev z definicijo. Nato dokažemo nekaj meta-izrekov o transformacijah. Uporabnost transformacij teorij tipov se pokaže v definiciji dopolnitve. Dokažemo izrek o dopolnitvi, ki pravi, da ima vsaka končna teorija tipov dopolnitev.

Na strani implementacije oblikujemo splošen algoritem za preverjanje enakosti, ki ga lahko uporabimo na končnih teorijah tipov. Algoritem je sestavljen iz faze ekstenzionalnosti, kjer uporabljamo pravila ekstenzionalnosti, in faze normalizacije, ki temelji na pravilih za izračun. Obe vrsti pravil sta definirani s pomočjo pogoja objektne obrnljivosti. Podamo tudi zadosten sintaktični kriterij za prepoznavanje teh pravil in algoritem s preprostimi vzorci, ki pravila uporablja. Tretja komponenta algoritma je primeren pojem glavnih argumentov, ki določa normalno obliko. S spreminjanjem glavnih argumentov dobimo znane pojme, kot sta šibka in močna normalna oblika. Dokažemo, da algoritem zadošča izreku skladnosti. Algoritem je implementiran v dokazovalnem pomočniku Andromeda 2.

2020 Mathematics Subject Classification: 03B38, 03B70, 18C10, 68V15, 68V20, 03F50, 03F03, 03F07

Ključne besede: odvisna teorija tipov, algebrajska teorija, dokazovalni pomočnik, dopolnitev teorij tipov, preverjanje enakosti.

Acknowledgements

I would like to thank my mentor Andrej Bauer for taking me on this exciting journey, for his scientific guidance, for helping me find my course in the research community, and for the enthusiasm with which he selflessly shares his ideas and knowledge. I can honestly say that every time I left his office, I felt inspired and eager to see what is to come next.

I thank the members of my thesis committee for taking on this role. I am grateful to Marko Petkovšek for his inspirational lectures when I was an undergraduate student which have sparked my interest in logic. Matija Pretnar shared his passion for programming languages and with his whimsical sense of humor always found a way to keep my spirits up. I thank him for all the advice and kind words he offered. Thanks to Bas Spitters for helping me navigate an academic career and always keeping an eye out for possible opportunities to expand my horizon. The contagious excitement for research he radiates and his open mind are always motivators to start (or keep working on) a project. A special thanks goes to Sandi Klavžar for joining the committee in the last phase and for his input on slovenian summary.

This thesis would not have been possible without the many enlightening conversations I had with the members of the type theory community. Philipp Georg Haselwarter played the role of my academic older brother, helping me with my initial attempts to tackle the theoretic and programming sides of research. He was a great studying partner and fun to collaborate with. I would like to thank Peter LeFanu Lumsdaine for the numerous times he shared his insights. I am grateful to Guillaume Brunerie for the countless hours we spent discussing the ideas of general type theories and for the friendship that formed along the way. It was fun to talk to, formalize (and run) with Dominik Kirst who always made me feel welcome in the community. Théo Winterhalter never hesitated to offer his help and start a discussion about type-theoretic translations, software and life. Kira Kutcher and I share numerous views on computer science and the world. I could always count on Anders Mörtberg to share his insights, ideas and explanations, keeping a light atmosphere and a smile. There were many others who shaped my experience: Martin Escardo, Ingo Blechschmidt, Matthieu Sozeau, Mike Shulman, Thorsten Altenkirch, Assia Mahboubi, Steve Awodey, Ohad Kammar, Fredrik Nordvall Forsberg and numerous people I met in the community.

Being a part of the foundations of mathematics and theoretical computer science research group in Ljubljana has been a delight. I am especially thankful to Žiga Lukšič for being and incredible office-mate and friend, for solving research and life problems

with me, making my days at the office fun and a thousand times better. I thank Alex Simpson for all the time and effort we spent trying to tackle some difficult problems. Collaborating with Alex has taught me a great deal and was a wonderful experience. Egbert Rijke and Danel Ahman were always kind in sharing their insights and knowledge. Work was more fun when Brett Chenoweth, Niels Voorneveld, Riccardo Ugolini, Filip Koprivec and Gavin Bierman were around. Their jokes often made my day. Taking a walk or a cup of coffee with Davorin Lešnik or Katja Berčič was always a pleasure. Over the years our group had many visitors who contributed to my knowledge, of whom Gaïa Loutchmia and Simona Kašterović were especially friendly.

I want to thank Jure Slak, Vesna Iršič, Žiga and Tatiana Primožič for our weekly lunch meetings. Being in your company is always an amazing time. Thanks to Vida Vukašinovič for being my role model all these years, for her support in work and life, providing me with encouragement whenever I needed it. I further thank her for our common projects in tackling gender balance issues. I thank Renata Dacinger for the opportunity to communicate science to the general public, for her guidance and support.

I wish to thank my friends Ožbe Pečar, Lidija Pečar, Ajda Demšar Luzar, Jernej Luzar, Maša Rožanc, Jože Jeraj, Sara Pia Marinček, Tomaž Nahtigal, Nino Bašič, Sara Bizjak, Andreja Novak and Brigita Pinter for keeping my spirits up when life got difficult and for all the good times we shared.

Words are hardly good enough to express my gratitude to my family – my parents Marjan and Mirjana; and my sisters Bojana and Darja with their families Marko, Igor and Peter. They supported me in my vision to be a scientist, encouraged me in my doubts and stood by my side every step of the way. I am grateful to have them in my life.

A big thanks goes to my other family as well, Jani, Sneža, Petra and in the recent years Ivan, for supporting me on this path. A special thanks goes to the extended Komel family too, especially to Anča for her mathematics lessons.

And most importantly, I want to thank my husband Luka for supporting me through the best and the worst of times, for keeping me sane when I was overwhelmed, for his patience and thoughtfulness when I was overworked, for sharing with me the joy of finishing a proof, and for the life we share.

This material is based upon work supported by the U.S. Air Force Office of Scientific Research under award number FA9550-17-1-0326, grant number 12595060, and award number FA9550-21-1-0024.

A quick guide to reading this thesis

In this day and age a document of this size is usually read on an electronic media. Thanks to Théo Winterhalter, who inspired me to use the kaobook class to generate this kind of document, the thesis is adapted for a convenient use electronically, without compromising the experience on the paper version.

The document contains a wide margin to accommodate accompanying comments, citations, side notes etc. The reader may also use the margin to scribble their own notes should they wish – this feature is actually even more enjoyable on paper. The references to definitions (like Definition 9.1.2), theorems (like Theorem 10.2.1), lemmas (like Lemma 13.2.10), propositions (like Proposition 5.1.2), examples (like Example 9.3.1) and even type-theoretic rules (like TT-Tm-Sym) are clickable so one can quickly travel to the relevant part of the document.

[24]: Bauer et al. (2021), *An extensible equality checking algorithm for dependent type theories*

Citations like [24] appear in the margin (also clickable) in a short version listing just the first author, as well as in the Bibliography where the full citations can be found. Instead of the footnotes we use side notes[1] so one does not have to scroll (or look) at the bottom of the page.

1: Just like this one :)

The margin will also contain margin notes, comments accompanying the main text. These notes are not placed automatically, but are adjusted so they are close to the relevant paragraph.

This is supposedly relevant to what is on the left.

The margin also contains reminders.

Reminder: of a concept

Sometimes we will have a reminder in the margin if a concept is not used very often.

Theorem 0.1 Theorems will stand out in these red boxes, so they are easy to spot.

Definition 0.2 Definitions will appear in the yellow boxes. The defining notions will appear in *red*.

Lemma 0.3 Lemmas and propositions are in green boxes.

If a reminder is a definition, it will appear in a yellow box. Sometimes we will introduce a notion without paying too much attention to it. In such cases we will also use the yellow boxes on the right.

This is a *side definition*.

Contents

Prologue: The story of Carla

Meet Carla.

Carla. Source: Pngwing.

Carla is a young mathematician, ready to take on the world. One day she was sitting in her office, re-reading a paper she had written and was just published, when she suddenly almost spilled the cup of chamomile tea (or at least that is what she claims it was) in her hand. There was a glaring mistake in one of her proofs. How could she miss that?! Hmmmmm, how could all the reviewers miss that? Well, she had been pretty tired in the days before the submission deadline, as she (as usually) procrastinated with writing the paper to the last minute, so she had probably not been as focused as she would have wanted. And if she was completely honest with herself, every time she had reviewed a paper thus far, she had read some parts more thoroughly than others. To calm her nerves she tried to fix the mistake, setting down the cup in her shaking hands first. Of course, since the mistake was not just a typo, it was not so easy.

When hours passed with no luck, she left the office frustrated and tired, thanking herself she took the train to work that morning instead of driving herself. Observing the changing scenery from the train windows, her mind drifted to the events of that day. Naturally, she will publish the corrected version of the proof – if she ever finds a way to fix it. But what bothered her the most was that the mistake slipped through the cracks and made its way to being published at all.

Seeking consolation and a little bit of schadenfreude, Carla opened up her internet browser and searched for similar stories. She stumbled upon the story of Vladimir Voevodsky, a Russian mathematician with an impressive résumé, after all he did receive the prestigious Field's medal for his work. It turns out that after Voevodsky was awarded the medal, he wrote another paper [81] which was published and only after a mistake was found [131]. Concerned with the drawbacks of the human-reviewing process, he grew increasingly convinced that proofs should be checked by computers, as well as humans. He used his mathematical influence to accelerate the research in the area of proof assistants, computer software that does what the names suggests - assists humans in proving and checking the proofs. The evolution of the proof assistants goes hand in hand with the development of mathematical foundations, of which type theories, as

[81]: Kapranov et al. (1991), "∞-groupoids and homotopy types"
[131]: Simpson (1998), *Homotopy types of strict 3-groupoids*

it seems to Carla, are the most popular candidates on which proof assistants are based.

Carla was intrigued. She heard people mentioning proof assistants before, but so far she brushed it off as a special niche for the fanatics of constructive mathematics. A little disappointed with her own prejudice, yet curious of this new world, she decided to look into the matter a little further. She found out that there are many proof assistants based on type theories [5, 43, 45, 78, 108, 141, 146] and that their underlying theories differ. As a mathematician, she was dazzled by the fact that Georges Gonthier and his team machine-checked the proof of the odd order theorem [64], a proof renowned to be very long and difficult.

What further fascinated her was how using proof assistants one could not only machine-check mathematical proofs, but also formalize correctness of computer programs. What a revelation! As a fairly experienced programmer she was well aware of the usual way people test correctness of computer programmed functions: they choose a set of inputs and check if the function computes the outputs they expect. Of course they try to cover as many different examples of input as they can think of, but well, who trusts the programmers to catch all the cases? The situation is a bit better if one uses typed programming languages. If a function is supposed to return a list of integers, but suddenly it tries to output a string, the compiler will already catch the error and impose some discipline. But even then, errors that respect the type-system will not get caught (unless there is a specific input-output test for it somewhere). This (to Carla's eyes new) paradigm of proving correctness of programs was a player on a whole other level. We do not just *test* the correctness, we can *prove* a program fits its specification in *every* possible case. Carla could quickly imagine use cases for such a tool, when we want to be absolutely super duper sure the software is correct, like sending a rocket on Mars and knowing the navigation system is bug-free, or having a laser operation on your eyes when every millimeter makes a difference. Of course not every computer bug is lethal, in the majority of cases it just causes some frustration when an application on the phone unexpectedly crashes. With her experience, Carla soon realized that it takes a substantial amount of knowledge, work, time, and energy to verify program correctness for a sizable project, but a price worth paying for the "lethal" cases. A bit of further investigation into the subject led Carla to learn that an entire compiler for a version of programming language C was verified by Xavier Leroy and his team in a project called CompCert [90]. This is definitely a game changer as it eliminates the possibility of a bug occurring in the compiler itself, so a programmer who translates her programs to byte-code using this verified compiler has only but herself to blame for the possible mistakes in the code.

Carla was overwhelmed with relief, joy and all this new knowledge. She decided then and there to try proof assistants out herself. But where to start? The gods of the internet guided her to the Curry-Howard correspondence [48, 49, 74, 149] relating her knowledge of logic with type theory. She never thought of proofs as objects before, to her they were justifications that theorems and propositions were correct, but now proofs are thought of elements of types written as

[45]: (2021), *The Coq proof assistant, version 2021.02.2*
[5]: (2021), *The Agda proof assistant*
[108]: Moura et al. (2015), "The Lean Theorem Prover (System Description)"
[146]: Vezzosi et al. (2019), "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types"
[141]: The RedPRL Development Team (2020), *The 'redtt' theorem prover.*
[43]: Cohen et al. (2018), *The 'cubicaltt' theorem prover.*
[78]: Isaev (2021), *Arend Standard Library*

[64]: Gonthier et al. (2013), "A Machine-Checked Proof of the Odd Order Theorem"

[90]: Leroy (2009), "A formally verified compiler back-end"

[149]: Wadler (2015), "Propositions as Types"
[48]: Curry (1934), "Functionality in Combinatory Logic"
[49]: Curry et al. (1958), *Combinatory logic. Vol. I*
[74]: Howard (1980), "The Formulae-as-Types Notion of Construction"

$p : A$. To Carla this was a little unusual. If proofs are objects, how do we use logical connectives? Let us see, if Carla wants to prove that a and b holds, then she needs to provide two proofs, one for a and one for b, so a *pair* of proofs.

$$\text{proving} \qquad a \wedge b \qquad \text{corresponds to} \qquad (p_1, p_2) : A \times B$$

With an implication $a \implies b$ we know that if we have a proof of a, then we have a proof of b, so it is like a function that takes a proof of a and provides a proof of b.

$$\text{proving} \qquad a \implies b \qquad \text{corresponds to} \qquad p : A \to B$$

Easy peasy lemon squeezy.[1]

1: It is not actually that trivial. We need to introduce dependent product types and dependent sums and have a whole discussion about the law of excluded middle. But this is the story for another time, posssibly a textbook.

Before Carla could start playing around with a proof assistant, she was faced with a decision of which one to use. Since she had no previous experience with proof assistants, all the factors that influence experts, like the nature of the problem to be formalized, expressivity of the underlying type theory, availability of the necessary libraries, performance of a proof assistant etc., were not yet relevant to her. At the end, she decided to start with Coq [45], claiming it was because it was so widely used (as she remains in denial that it was because of the name).

[45]: (2021), *The Coq proof assistant, version 2021.02.2*

Over the course of time, with grit, perseverance, and a lot of help from the kind-hearted type-theory experts, Carla gradually gained experience with formalizing proofs. One day she was finally ready to tackle the proof from her paper – the one that almost spilled her chamomile tea. Now that she enriched her mathematical intuition, it was easy to spot the mistake and, what's more, the proof assistant helped her identify and solve the problem. But there was another upside: Carla was now confident that her proof was correct and she will not have to go through the same emotional turmoil again. She felt like a superwoman!

Carla as a superwoman. Source: Pngwing and TopPng.

While formalizing (or at least trying to fomalise) proofs, Carla learned some of the intricacies of type theories and proof assistants. For example, one of the convenient features of Coq[2] is that Carla (and the other users) can declare some of the arguments of functions implicit like in the definition of the identity function below.

2: Implicit arguments are a common techique also in proof assistants other than Coq.

```
Definition id {A : Type} (x : A) : A := x.
```

The function as defined takes two arguments: a type argument `A` and a term argument `x` of type `A`. But the first argument is implicit (denoted

by the curly braces {}), as it can be inferred if we compute the type of the second argument. Thus Carla does not have to write the type of the domain (and codomain) of the identity function every time she uses it. What a relief!

Another important feature, Carla learned, is that Coq knows how to compute with terms. The command

```
Eval compute in double (double (S 0)).
```

computes to S(S(S(S(0)))) which is a unary representation of the number 4, where `double` is suitably defined function that doubles a natural number and S is the successor. Behind the scenes is an equality checking algorithm[3] with a normalization component that enables such (and much more complex) computations.

3: Equality checking algorithms are are also an essential part of other proof assistants.

Carla also observed there is a lot of mathematical knowledge already formalized in several libraries, like UniMath [148] for example, and even more still missing formalization. However, what troubled Carla is that the libraries, and what is worse the underlying type theories, between proof assistants are not always compatible. What can be proven in a type theory depends on its rules and axioms, which are specific to the theory itself. But Carla knows that not every proof relies on *all* the rules and axioms of the theory, just the ones that are used in deriving it, after all there are theorems proven using many different type theories. Maybe if a fragment of the theory used to prove a theorem is compatible with another theory, we could translate the proof to another proof assistant?

[148]: Voevodsky et al. *UniMath — a computer-checked library of univalent mathematics*

Carla's sharp mathematical intuition screamed that if we are going to fiddle with the proofs and translate them into another type theory, there better be a precise mathematical (meta-)theory that allows us to do just that. Looking for a general and possibility syntactic definition of a type theory and transformations of type theories, she found some very deep and elaborate notions [15, 22, 34, 52, 66, 69, 144, 151], but unfortunately none of them analyzed syntactic transformations to the extent she had hoped for.

[22]: Bauer et al. (2020), *A general definition of dependent type theories*
[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*
[144]: Uemura (2019), *A General Framework for the Semantics of Type Theory*
[66]: Harper (2021), *An Equational Logical Framework for Type Theories*
[52]: Dybjer (1995), "Internal Type Theory"
[34]: Cartmell (1986), "Generalised algebraic theories and contextual categories"
[15]: Awodey (2018), "Natural models of homotopy type theory"
[151]: Winterhalter (2020), "Formalisation and Meta-Theory of Type Theory"

With a playful mind and an enthusiastic heart, inspired by all the meta-theoretic mathematical definitions of type theory Carla turned to proof assistants to experiment. So where better to start than in a general-type-theory purpose proof assistant Andromeda 2 [9]. While happy that she can input the rules of the desired type theory herself, she was soon getting frustrated with the fact that the stupid machine did not know how to compute anything. Andromeda 2 was lacking the support of an equality checking algorithm. But how do we design an equality-checking algorithm when we don't even know what equality rules will appear in the theory?

[9]: Bauer et al. *The Andromeda proof assistant*

Carla's heart sank. Yet again the lack of meta-theoretic results obstructed her experiments. It was definitely the time to make another chamomile tea.

1. Introduction

Despite there being several instances of type theories [26, 36, 42, 47, 97–100, 142], their general (syntactic) definitions are a fruit of recent work [22, 66, 69, 144]. These definitions opened up a new meta-level on which we can analyze properties of type theories, what all the instances of type theories have in common and how they interact. In this thesis we develop a meta-analysis of type theories as syntactically defined in [69]. We investigate interactions between type theories by defining several notions of transformations and proving their meta-theoretic properties. The usefulness of said transformations culminates in the definition of an elaboration of a type theory and the proof of the elaboration theorem. Since a big part of the motivating machinery for the development of type theories is their use in proof assistants, we also address one meta-theoretic aspect that closely relates to the implementations, namely the general equality-checking algorithm which we also implemented in the Andromeda 2 proof assistant [9].

[98]: Martin-Löf (1982), "Constructive mathematics and computer programming"
[100]: Martin-Löf (1998), "An intuitionistic theory of types"
[97]: Martin-Löf (1975), "An intuitionistic theory of types: predicative part"
[99]: Martin-Löf (1984), *Intuitionistic type theory*
[47]: Coquand et al. (1988), "Inductively defined types"
[36]: Church (1940), "A Formulation of the Simple Theory of Types"
[142]: The Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*
[42]: Cohen et al. (2015), "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom"
[26]: Bezem et al. (2019), "The Univalence Axiom in Cubical Sets"
[22]: Bauer et al. (2020), *A general definition of dependent type theories*
[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*
[144]: Uemura (2019), *A General Framework for the Semantics of Type Theory*
[66]: Harper (2021), *An Equational Logical Framework for Type Theories*
[9]: Bauer et al. *The Andromeda proof assistant*

There are three integral parts of this thesis. The first formally describes the type theories we work with, the second part analyzes transformations and the third part gives the equality checking algorithm. Instead of introducing all the concepts at once, each part is given their own introduction (Chapter 2, Chapter 6, Chapter 12). We thus invite the readers, who are afraid of drowning in technicalities, to read those introductory chapters first.

1.1. Aims of the thesis

The aim of the thesis is to provide a meta-analysis of interactions of type theories in the form of their transformations which satisfy the following criteria

- they work in the fully general setting of finitary type theories of [69],
- transformations are syntactic in nature, so they are pertinent to possible implementations in proof assistants,
- preservation of derivability is inherent to transformations of type theories,
- we can exhibit some useful examples,

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

and to develop the meta-theory of type theories for designing a general equality checking algorithm which

- works in the fully general setting of finitary type theories of [69],
- is sound,
- permits possible implementations,
- works as expected on well-behaved theories seen in practice.

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

We approach the first goal by defining notions of a *syntactic transformation*, *type-theoretic transformation* and an *elaboration map*, prove they satisfy the desired properties and exhibit their usefulness in the *elaboration theorem*. The second goal is achieved by proposing the condition of *object-invertibility* which identifies the rules suitable to be used in an equality checking algorithm, and by designing a sound algorithm and implementing it in Andromeda 2.

1.2. Overview and structure of the thesis

The thesis is split into three parts: 'Finitary Type Theories', 'Transformations of type theories' and 'An equality checking algorithm'. The purpose of the first part is to give the background and notations used throughout the thesis. It describes what we mean by a type theory and provides the necessary meta-theorems. The second part is dedicated to transformations of type theories, their definitions and meta-theoretic properties. In the third part we develop a general equality checking algorithm for dependent type theories and prove that it is sound.

Instead of having a long introduction and an even longer conclusion, each part has its own introduction (Chapter 2, Chapter 6, Chapter 12) with a specific section that describes *contributions* (Section 2.4, Section 6.1, Section 12.1). There is also a related work section for every part individually (Section 2.3, Section 11.1, Chapter 17). Thus the reader gets introduced to the concepts right before they are being used and can discuss about them while they are still fresh in mind.

The thesis ends with some concluding remarks (Chapter 18).

Finitary type theories

We start in Chapter 2 with a brief description of the development of type theories, especially focusing on the general definitions given in the recent years. We then proceed by describing one such definition, namely the *finitary type theories* proposed by Haselwarter and Bauer [69] following the presentations that appears in [24]. We first describe the syntactic entities (Chapter 3) that appear in the four Martin-Löf style judgement forms and then add the deductive system (Section 4.1) given by the *raw rules* (Section 4.2) to define a *raw type theory* (Section 4.3). Since such theories can have some circularities in their derivations, we give definitions of *finitary type theories* and *standard type theories* (Section 4.4). In the Chapter 5 we list some basic meta-theorems about type theories and prove some more (Section 5.3) that are needed in the rest of the thesis.

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

[24]: Bauer et al. (2021), *An extensible equality checking algorithm for dependent type theories*

Transformations of type theories

This part of the thesis is dedicated to studying meta-theoretic properties of transformations between type theories. We begin with an introduction (Chapter 6) that describes the stipulations we try to adhere

to. Since the syntactic transformations we define in Chapter 8 form a relative monad for syntax, we first review the definition of relative monads in Chapter 7 and describe the shape of relative monads that arise from substitutions of variables and instantiations of metavariables. In Chapter 8 we also prove that the syntactic transformations indeed satisfy the conditions for a relative monad and we thus organize their (somewhat expected) meta-theoretic properties. Transformations are then upgraded to *type-theoretic transformations* to account for the derivability structure of the type theories. In Chapter 9 we also prove some meta-theorems about the type-theoretic transformations, such as that they preserve derivability (Theorem 9.1.3) and how judgementally equal transformations interact. As defined, type-theoretic transformations are morphisms in the category of type theories (Section 9.2) which we also prove has an initial object and coproducts.

In Section 9.3 we exhibit the scope of our definition on some examples (and non-examples) of type-theoretic transformations, such as the *propositions as types* translation (details of which are in the Appendix Chapter A) and *definitional extension*. However, the major use case for our notion of type-theoretic transformations is *elaboration*. In Chapter 10 we precisely define what is an elaboration of a finitary type theory and prove it is universal. We also state and prove the *elaboration theorem* (Theorem 10.2.1) stating that every finitary type theory can be elaborated, and inspect some algorithmic properties of elaboration in Section 10.3.

We conclude the part in Chapter 11 with a discussion about related work and future directions.

Equality checking algorithm

The third part of the thesis is published in [24]. In the introductory chapter (Chapter 12) we comment on some design decision made in the development of the equality checking algorithm as well as what are our original contributions (Section 12.1). Chapter 13 describes *object-invertibility*, a meta-theoretic condition on equality rules which ensures the rules can be used in the algorithm. We also provide a syntactic condition in the form of *patterns*. We then formally define the *computation rules* and *extensionality rules* in Chapter 14 and provide some examples.

[24]: Bauer et al. (2021), *An extensible equality checking algorithm for dependent type theories*

The algorithm itself in given in Chapter 15. We start with an overview and describe the components of the algorithm. The *normalization* phase is given in Section 15.1 and the *type-directed phase* in Section 15.2. We prove the algorithm is sound (Section 15.3) and then discuss the design of the algorithm in Section 15.4.

The algorithm was implemented in the Andromeda 2 proof assistant. We describe the implementation in Chapter 16 and give examples in Section 16.1 to showcase the scope and interesting use-cases of the algorithm.

The part concludes with a chapter on related work Chapter 17.

Finitary Type Theories

2. The era of type theories

Type theories are formal systems, which serve many purposes ranging from foundations of mathematics to applications in computer science, usually in the form of proof assistants. They are presented as sets of reasoning rules that govern types, terms, and computations with these. In this part of the thesis we describe what precisely we mean by a *type theory*, starting with some of the work our definition is based on.

2.1. Development of type theories

The history of development of type theories is a very broad subject, so we only mention a few crucial points in their development.

Bertrand Russell [129] conceived type theory as a response to the crisis in foundations of mathematics that was caused by the discovery of his and other paradoxes of naïve set theory. Russell and Whitehead epitomized formal type theory in their famous work Principia Mathematica [150]. Influenced by Russel's idea, Church [36, 37] developed his theory of *simple types* in combination with λ-calculus, which paved the way for development of *computational type theory* as Church, Turing and Kleene showed that λ-calculus is a universal model of computation which can be used to simulate any Turing machine. The calculus was later on applied in computer science in the theory of programming languages [121].

[129]: Russell (1908), "Mathematical Logic as Based on the Theory of Types"

[150]: Whitehead et al. (1925), *Principia Mathematica*

[36]: Church (1940), "A Formulation of the Simple Theory of Types"

[37]: Church (1941), *The Calculi of Lambda Conversion. (AM-6)*

[121]: Pierce (2002), *Types and Programming Languages*

An important step in development of proof assistants is the *Curry-Howard correspondence* which connected types and propositions, logic and programming, computations and natural deduction. In 1976 with De Bruijn and his Automath, the first proof assistant that enabled manipulations of derivations, dependent types were introduced. The evolution of dependent type theories became tightly intertwined with development of proof assistants.

These concepts resonated with Per Martin-Löf [97–100], when he pioneered his own type system, variants of which are nowadays known as *Martin-Löf type theories* (MLTT). They were originally designed as a formal foundation of constructive mathematics. Through years MLTT evolved and passed through several changes. At first it postulated a type of all types and it had no identity types. Then there was *Extensional Type Theory* (ETT), which employs equality reflection, and later *Intentional Type Theory* (ITT) which is the foundation of the proof assistant Agda [5].

[100]: Martin-Löf (1998), "An intuitionistic theory of types"

[97]: Martin-Löf (1975), "An intuitionistic theory of types: predicative part"

[98]: Martin-Löf (1982), "Constructive mathematics and computer programming"

[99]: Martin-Löf (1984), *Intuitionistic type theory*

[5]: (2021), *The Agda proof assistant*

In parallel with Martin-Löf, Milner worked on the LCF proof checker [103], which lead to development of HOL proof assistants [76] and the ML programming language[65]. Around the same time Girard and Reynolds [60, 128] independently designed *System F*.

[103]: Milner (1972), *Logic for Computable Functions: description of a machine implementation*

[76]: (2016), *Isabelle*

[65]: Gordon et al. (1978), "A Metalanguage for Interactive Proof in LCF"

[60]: Girard (1972), "Interprétation Fonctionelle et Élimination Des Coupures de l'arithmétique d'ordre Supérieur"

[128]: Reynolds (1974), "Towards a theory of type structure"

Based on Martin-Löf's work a number of other type theories came to life. In 1986 Constable and his group invented the proof assistant Nuprl and with it *Computational Type Theory* [6, 44]. Thierry Coquand and Gérard Huet defined *Calculus of Constructions* [46] that Coquand together with Christine Paulin upgraded to *Calculus of Inductive Constructions* [47] which is the basis of the widely used proof assistants *Coq* [45] and Lean [88, 106, 109].

In the early 2000s it was observed that identity types of ITT have homotopical content: identifications between terms can be thought of as paths. This new outlook gave rise to *Homotopy Type Theory* (HoTT) [142, 147], which inspired a lot of new research. The need for use of HoTT resulted in implementations in proof assistants, such as the HoTT Library [21] and UniMath [148].

The lack of explicit computational content in the univalence axiom motivated researchers to investigate *Cubical Type Theories* [25, 42, 91] that are still being intensively studied to this day. There are also several implementations of variants of cubical type theories, such as cubicaltt [43], redTT [141], Cubical Agda [146] and Arend [78].

2.2. Semantic approach to type theories

To be able to rely on a type theory, one has to prove that it does not derive unsound judgments. While in this thesis we use syntactic arguments to prove the correctness of our constructions, the semantic approach is often more efficient and powerful. In sematic approach we construct a model of type theory, a mathematical structure that interprets its inference rules, and study its properties. The internal language is then a part of the correspondence between the theory and the model.

There are several examples of such correspondence between type theories and models. The first is commonly attributed to Lambek and Scott [87] for showing that *cartesian closed categories* are a model of simply-typed λ-calculus. In 1978 Cartmell introduced *generalised algebraic theories* and *contextual categories* in his thesis [33]. A model of MLTT was provided by Seely in 1984 in the form of *locally cartesian closed categories* [130] and Martin Hofmann and Thomas Streicher constructed the *groupoid model* [73]. *Simplicial sets* provided the first known model of univalent type theory [82, 137] and *cubical sets* for cubical type theories [25, 26, 42, 92, 111].

There are also numerous models of a type theory such as Bart Jacobs' *comprehension categories* [79], *display map categories* [139] by Paul Taylor, *categories with attributes* [34, 105], Peter Dybjer's *categories with families* [52] (which are equivalent to categories with attributes) and Steve Awodey's *natural models* [15] (which, as Awodey noted, are also equivalent to categories with families of Dybjer [52]). Among these notions of a model, the closest to the syntax of type theories are categories with families and for a while they were considered a canonical notion of a model of a type theory. All these models are modeling a particular instance of a type theory. If we extend

[44]: Constable et al. (1986), *Implementing mathematics with the Nuprl proof development system*
[6]: Allen et al. (2006), "Innovations in computational type theory using Nuprl"
[46]: Coquand et al. (1988), "The Calculus of Constructions"
[47]: Coquand et al. (1988), "Inductively defined types"
[147]: Voevodsky (2011), "Univalent Foundations of Mathematics"
[142]: The Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*
[42]: Cohen et al. (2015), "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom"
[25]: Bezem et al. (2014), "A Model of Type Theory in Cubical Sets"
[91]: Licata et al. (2015), "A Cubical Approach to Synthetic Homotopy Theory"
[87]: Lambek et al. (1986), *Introduction to Higher Order Categorical Logic*
[33]: Cartmell (1978), "Generalised algebraic theories and contextual categories"
[130]: Seely (1984), "Locally cartesian closed categories and type theory"
[73]: Hofmann et al. (1994), "The Groupoid Model Refutes Uniqueness of Identity Proofs"
[82]: Kapulkin et al. (2021), "The simplicial model of Univalent Foundations (after Voevodsky)"
[137]: Streicher (2014),
[25]: Bezem et al. (2014), "A Model of Type Theory in Cubical Sets"
[26]: Bezem et al. (2019), "The Univalence Axiom in Cubical Sets"
[42]: Cohen et al. (2015), "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom"
[111]: Orton et al. (2018), "Axioms for Modelling Cubical Type Theory in a Topos"
[92]: Licata et al. (2018), "Internal Universes in Models of Homotopy Type Theory"
[79]: Jacobs (1993), "Comprehension categories and the semantics of type dependency"
[139]: Taylor (1986), "Internal Completeness of Categories of Domains"
[52]: Dybjer (1995), "Internal Type Theory"
[15]: Awodey (2018), "Natural models of homotopy type theory"
[105]: Moggi (1991), "A category-theoretic account of program modules"
[34]: Cartmell (1986), "Generalised algebraic theories and contextual categories"

the theory with some constructors, we need to update the model as well.

2.3. General definitions of type theories and related work

With increasing number of instances of type theories, the need for a general definition of a type theory gradually surfaced. Proof assistants strived to extend the underlying type theory in various ways. One such approach are syntactic models [28, 31, 71, 115] in which one extends type theories via shallow embeddings. To extend judgemental equality, some proof assistants allow limited rewrite rules as for example in Coq Modulo Theory [18], Agda's rewrite rules [38, 39, 41] or Dedukti [50]. On the semantic side, categories with families were considered the canonical model for type theories, but they still need to be adapted for possible extensions.

[71]: Hofmann (1997), "Extensional Constructs in Intensional Type Theory"
[28]: Boulier et al. (2017), "The next 700 syntactical models of type theory"
[31]: Boulier (2018), "Extending type theory with syntactic models. (Etendre la théorie des types à l'aide de modèles syntaxiques)"
[115]: Pédrot et al. (2017), "An effectful way to eliminate addiction to dependence"

A general, mathematically precise definition of type theories is a fruit of recent work. In 2016 Valery Isaev [77] proposed a definition of a general type theory as a special essentially algebraic theory. A model of said type theory is then a special case of a model of the underlying essentially algebraic theory. His work is equivalent to a generalization of a contextual category.

[77]: Isaev (2017), *Algebraic Presentations of Dependent Type Theories*

In 2020 Andrej Bauer, Peter LeFanu Lumsdaine and Philipp Georg Haselwarter proposed a variant of *general type theories* [22]. The definition is syntactical and proceeds in stages, from raw syntax to well-formed rules. General type theories have been modified and extended by Bauer and Haselwarter in the form of *finitary type theories* [69] that are also presented without contexts as *context-free type theories*. The two definitions are very similar, the most notable difference being the treatment of metavariables: in general type theories metavariables extend the signature, while in finitary type theories they are given separately in a metavariable context.

[22]: Bauer et al. (2020), *A general definition of dependent type theories*
[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*
[9]: Bauer et al. *The Andromeda proof assistant*
[67]: Harper et al. (1993), "A Framework for Defining Logics"
[118]: Pfenning (2001), "Logical Frameworks"

We use finitary type theories [69] as our definition of type theories. We summarize it in Chapter 3 and Chapter 4. The context-free representation of finitary type theories has also been implemented in the Andromeda 2 proof assistant [9].

[120]: Pientka et al. (2010), "Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)"

For representing logics, the *logical frameworks* [67, 118] are extensively used. Numerous implementations of logical frameworks such as Twelf [119] and Beluga [120] have been used to study the metatheoretic properites of formal deductive systems and programming languages. In this spirit have Taichi Uemura [144, 145] in 2019 and Rober Harper [66] in 2021 provided logical frameworks for representing general dependent type theories. Uemura also presents another (more semantic) definition of type theories, namely (small) categories with representable maps, and describes the models of such type theories. Uemura's models are a generalisation of natural models [15]. Unlike with the definition of Bauer, Haselwarter and Lumsdaine that gives the four judgement forms in the style of Martin-Löf, Uemura's definition of type theory allows for other judgement forms as well

[119]: Pfenning et al. (1999), "System Description: Twelf — A Meta-Logical Framework for Deductive Systems"
[66]: Harper (2021), *An Equational Logical Framework for Type Theories*
[144]: Uemura (2019), *A General Framework for the Semantics of Type Theory*
[145]: Uemura (2021), "Abstract and Concrete Type Theories"
[15]: Awodey (2018), "Natural models of homotopy type theory"

and can thus express special judgement forms of cubical type theory, pure type systems [17], two-level type theory [7, 10] and polymorphic type theory [59, 61, 127].

The two definitions of type theories; the one of Bauer, Haselwarter and Lumsdaine; and the one proposed by Uemura; have been developed in parallel, so a more detailed comparison can be found in [69] and [145].

On the side of proof assistants we should also mention the MetaCoq project [132, 133], which implements and certifies the type-checker for the Coq proof assistant, tackling the practical issues when the implementation and specification differ.

[17]: Barendregt (1993), "Lambda Calculi with Types"
[10]: Annenkov et al. (2019), *Two-Level Type Theory and Applications*
[7]: Altenkirch et al. (2016), "Extending Homotopy Type Theory with Strict Equality"

[59]: Girard (1971), "Une Extension De L'Interpretation De Gödel a L'Analyse, Et Son Application a L'Elimination Des Coupures Dans L'Analyse Et La Theorie Des Types"
[127]: Reynolds (1974), "Towards a Theory of Type Structure"
[61]: Girard et al. (1989), *Proofs and Types*
[132]: Sozeau et al. (2020), "The MetaCoq Project"
[133]: Sozeau et al. (2019), "Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq"

2.4. Contributions

The purpose of this part of the thesis is to give the necessary background and notations used throughout the thesis. The definitions of the syntactic entities (Chapter 3) and type theories (Chapter 4, Chapter 5) are mainly the work of Philipp Georg Haselwarter and Andrej Bauer, and are taken from [69], more or less in the form published in [24] (extended with some additional explanations).

The original contributions are some additional meta-theorems, specifically Theorem 5.1.5 about judgementally equal instantiations which was developped jointly with Haselwarter and Bauer, and Section 5.3 on meta-theorems which contains

- meta-theorems about the natural type: Proposition 5.3.1, Corollary 5.3.2, Corollary 5.3.3,
- meta-theorems about judgementally equal instantiations: Lemma 5.3.4, Lemma 5.3.5, Proposition 5.3.6.

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

[24]: Bauer et al. (2021), *An extensible equality checking algorithm for dependent type theories*

3. Syntax of Finitary Type Theories

We give here only an overview of the syntax of such theories and refer the reader to [69] for a complete exposition. Following [69], we study syntactic presentations of type theories, in the sense that theories are seen as syntactic constructions, and the meta-theorems are obtained by analyzing abstract syntax. The motivation for such an approach is implementation in proof assistants, specifically in Andromeda 2. It is expected that the syntactic presentations will match nicely with some of the modern semantic accounts of type theories, and finitary type theories will be useful beyond being the theoretical support for proof assistants.

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

The definition captures dependent type theories in the Martin-Löf style, i.e., theories that have four judgement forms (for terms, types, type equations, and typed term equations), and hypothetical judgements standing in contexts of metavariables and variables.

This representation subsumes a wide class of type theories, including intensional and extensional Martin-Löf type theory, possibly with Tarski-style universes, homotopy type theory, Church's simple type theory, simply typed λ-calculi, and many others. And as mentioned in Section 2.3 it is also easy to find counter-examples: in cubical type theory the interval type has a special judgement form, impredicative polymorphic λ-calculi quantify over all types, pure type systems organise judgement forms differently, etc.

3.1. Signatures and arities

In a raw type theory there are four *judgement forms*:

- "A type" asserting that A is a type,
- "$t : A$" asserting that t is a term of type A,
- "$A \equiv B$ by $\star_{\mathsf{Ty}}$" asserting that types A and B are equal, and
- "$s \equiv t : A$ by $\star_{\mathsf{Tm}}$" asserting that terms s and t are equal at type A.

We indicate these with tokens Ty, Tm, EqTy and EqTm respectively. To each token there also corresponds a syntactic class. Expressions of class Ty are the *type expressions*, and those of class Tm are the *term expressions*. These are formed using *(primitive) symbols* and *metavariables*, see Section 3.2, each of which has an associated arity, as explained below. The symbols should be thought of as the primitive type and term formers, while the metavariables shall be used to refer to the premises of a rule, and as pattern variables in the equality checking algorithm. The only expressions of syntactic classes EqTy and EqTm are the dummy expressions $\star_{\mathsf{Ty}}$ and $\star_{\mathsf{Tm}}$, which we both write as $\star$ when no confusion can arise. These are formality, to be used where one would normally record a proof term witnessing a premise, but the premise is a judgemental equality, which is proof irrelevant.

The *arity* of a metavariable M is a pair (c, n), where the *syntactic class* $\mathsf{c} \in \{\mathsf{Ty}, \mathsf{Tm}, \mathsf{EqTy}, \mathsf{EqTm}\}$ indicates whether M is respectively a type, term, type equality, or term equality metavariable, and $n \in \mathbb{N}$ is the number of term arguments it accepts. The metavariables of syntactic classes Ty and Tm are the *object metavariables*, and they participate in formation of expressions, while those of syntactic classes EqTy and EqTm are the *equality metavariables*, and are used to refer to equational premises.

Metavariable arities are collected in *metavariable shapes*, lists of the form $[\mathsf{M}_1{:}(\mathsf{c}_1, m_1), \ldots, \mathsf{M}_n{:}(\mathsf{c}_n, m_n)]$, where (c_i, m_i) is the metavariable arity of M_i.

The *symbol arity* $(\mathsf{c}, [(\mathsf{c}_1, n_1), \ldots, (\mathsf{c}_k, n_k)])$ of a symbol S tells us that

1. the syntactic class of expressions built with S is $\mathsf{c} \in \{\mathsf{Ty}, \mathsf{Tm}\}$,
2. S accepts $k \in \mathbb{N}$ arguments,
3. the i-th argument has syntactic class $\mathsf{c}_i \in \{\mathsf{Ty}, \mathsf{Tm}, \mathsf{EqTy}, \mathsf{EqTm}\}$ and binds $n_i \in \mathbb{N}$ variables.

Symbol arity is a pair of a syntactic class c and a metavariable shape without the names of metavariables[1] $[(\mathsf{c}_1, n_1), \ldots, (\mathsf{c}_k, n_k)]$.

1: Sometimes we need to speak about the metavariables from the arity of a symbol (Section 8.2). In those cases we can generate a metavariable shape with metavariables $\mathsf{M}_1, \ldots, \mathsf{M}_n$ from the arity of the symbol.

Example 3.1.1 The arity of a type constant such as bool is $(\mathsf{Ty}, [])$, the arity of a binary term operation such as $+$ is $(\mathsf{Tm}, [(\mathsf{Tm}, 0), (\mathsf{Tm}, 0)])$, and the arity of a quantifier such as the dependent product Π is $(\mathsf{Ty}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1)])$ because it is a type former taking two type arguments, with the second one binding one variable.

The information about arities is collected in a *signature*, which maps each symbol to its arity.

When discussing syntax, it is understood that such a signature and a metavariable shape have been given, even if we do not mention them explicitly. In the presence of a metacontext (Section 3.3), the metavariable shape is deduced from there.

3.2. Expressions

The syntax of finitary type theories is summarized in the top part of Figure 3.1. There are three kinds: type expressions, term expressions, and arguments.

A *type expression* is an application $\mathsf{S}(e_1, \ldots, e_n)$ of a *primitive symbol* S to arguments, or an application $\mathsf{M}(t_1, \ldots, t_n)$ of a *metavariable* M to terms. We write S and M instead of $\mathsf{S}()$ and $\mathsf{M}()$.

A *term expression* is a variable, an application of a primitive symbol to arguments, or an application of a metavariable to terms. We strictly separate free variables $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$ from the bound ones $x, y, z, \ldots$, a choice fashioned after the locally nameless syntax [35, 102], a common implementation technique in which free variables are represented as names and the bound ones as de Bruijn indices.

[102]: McKinna et al. (1993), "Pure Type Systems Formalized"
[35]: Charguéraud (2012), "The Locally Nameless Representation"

Type expression A, B	$::= \mathsf{S}(e_1, \ldots, e_n)$	type symbol application
	$\mid \mathsf{M}(t_1, \ldots, t_n)$	type metavariable application
Term expression s, t	$::= \mathsf{a}$	free variable
	$\mid x$	bound variable
	$\mid \mathsf{S}(e_1, \ldots, e_n)$	term symbol application
	$\mid \mathsf{M}(t_1, \ldots, t_n)$	term metavariable application
Argument e	$::= A$	type argument
	$\mid t$	term argument
	$\mid \star_{\mathsf{Ty}}$	dummy type equality argument
	$\mid \star_{\mathsf{Tm}}$	dummy term equality argument
	$\mid \{x\}e$	abstraction (x bound)
Judgement thesis $\mathcal{J}$	$::= A$ type	A is a type
	$\mid t : A$	t has type T
	$\mid A \equiv B$ by $\star_{\mathsf{Ty}}$	A and B are equal types
	$\mid s \equiv t : A$ by $\star_{\mathsf{Tm}}$	s and t are equal terms at A
Abstracted judgement: $\mathcal{J}$	$::= \mathcal{J}$	judgement thesis
	$\mid \{x{:}A\}\mathcal{J}$	abstracted judgement (x bound)
Boundary thesis $\mathcal{b}$	$::= \square$ type	a type
	$\mid \square : A$	a term of type A
	$\mid A \equiv B$ by $\square$	type equation boundary
	$\mid s \equiv t : B$ by $\square$	term equation boundary
Abstracted boundary $\mathcal{B}$	$::= \mathcal{b}$	boundary thesis
	$\mid \{x{:}A\}\mathcal{B}$	abstracted boundary (x bound)
Variable context Γ	$::= [\mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n]$	
Metacontext Θ	$::= [\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n]$	
Hypothetical judgement	$\Theta; \Gamma \vdash \mathcal{J}$	
Hypothetical boundary	$\Theta; \Gamma \vdash \mathcal{B}$	

Figure 3.1.: The syntax of expressions, boundaries and judgements.

An *argument* is a type expression, a term expression, a dummy argument $\star_{\mathsf{Ty}}$ or $\star_{\mathsf{Tm}}$, or an abstracted argument $\{x\}e$ binding x in e. Note that we take abstraction to be a basic syntactic operation. For instance, we do not construe a λ-abstraction as a variable-binding construct $\lambda x{:}A\,.\,t$, but rather an application $\lambda(A, \{x\}t)$ of the primitive symbol λ to two separate arguments A and $\{x\}t$. We may abbreviate an iterated abstraction $\{x_1\}\cdots\{x_n\}e$ as $\{\vec{x}\}e$, and similarly use the vector notation elsewhere when appropriate. We permit $\vec{x}$ to be empty, in which case $\{\vec{x}\}e$ is just e. To an argument we assign the metavariable arity

$$\mathsf{ar}(\{x_1\}\cdots\{x_n\}e) = (\mathsf{c}, n),$$

where $\mathsf{c} \in \{\mathsf{Ty}, \mathsf{Tm}, \mathsf{EqTy}, \mathsf{EqTm}\}$ is the syntactic class of the non-abstracted argument e.

For an expression to be syntactically valid, all bound variables must be bound by abstractions, and all symbol and metavariable applications respect their arities. That is, if the arity of S is $(\mathsf{c}, [(\mathsf{c}_1, n_1), \ldots, (\mathsf{c}_k, n_k)])$ then it must be applied to k arguments $e_1, \ldots, e_k$ with $\mathsf{ar}(e_i) = (\mathsf{c}_i, n_i)$,

and the expression $\mathsf{S}(e_1, \ldots, e_k)$ has syntactic class c. Similarly, an object metavariable M of arity (c, n) must be applied to n term expressions to yield an expression of syntactic class c.

We write $e[t/x]$ for capture-avoiding ***substitution*** of t for x in e, and $e[\vec{t}/\vec{x}]$ or $e[t_1/x_1, \ldots, t_n/x_n]$ for simultaneous substitution of $t_1, \ldots, t_n$ for $x_1, \ldots, x_n$. Expressions which only differ in the choice of names of bound variables are considered syntactically identical (alternatively, we could use de Bruijn indices for bound variables).

Given an expression e, let $\mathsf{mv}(e)$ and $\mathsf{fv}(e)$ be the sets of metavariables and free variables occurring in e, respectively. A ***renaming*** of an expression e is an injective map ρ with domain $\mathsf{mv}(e) \cup \mathsf{fv}(e)$ that takes metavariables to metavariables and free variables to free variables. The renaming acts on e to yield an expression $\rho_* e$ by replacing each occurrence of a metavariable M and a free variable a with $\rho(\mathsf{M})$ and $\rho(\mathsf{a})$, respectively. We similarly define renamings of metacontexts, variable contexts, judgements and boundaries, which are defined below.

The renamings are also morphisms in the relative monad for syntax in Section 7.1.

3.3. Judgements and boundaries

We next discuss the syntax of judgements and boundaries, see the bottom part of Figure 3.1.

To each of the judgement forms corresponds a ***judgement thesis***:

- "A type" asserts that A is a type,
- "$t : A$" that t is a term of type A,
- "$A \equiv B$ by $\star_{\mathsf{Ty}}$" that types A and B are equal, and
- "$s \equiv t : A$ by $\star_{\mathsf{Tm}}$" that terms s and t of type A are equal.

The latter two have "by $\star$" attached so that all boundaries can be filled with a head, as we shall explain shortly. We normally write just "$A \equiv B$" and "$s \equiv t : A$".

A ***boundary*** is a fundamental notion of type theory, although perhaps less familiar. Whereas a judgement is an assertion, a boundary is a *goal* to be accomplished:

- "$\square$ type" asks that a type be constructed,
- "$\square : A$" that the type A be inhabited, and
- "$A \equiv B$ by $\square$" and "$s \equiv t : A$ by $\square$" that equations be proved.

An ***abstracted judgement*** has the form $\{x{:}A\}\ \mathcal{J}$, where A is a type expression and $\mathcal{J}$ is a (possibly abstracted) judgement. The variable x is bound in $\mathcal{J}$ but not in A. As before, we write $\{\vec{x}{:}\vec{A}\}\ \mathcal{J}$ for an iterated abstraction $\{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\ \mathcal{J}$. Similarly, an ***abstracted boundary*** has the form $\{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\ \mathcal{b}$, where $\mathcal{b}$ is a ***boundary thesis***, i.e., it takes one of the four (non-abstracted) boundary forms.

To an abstracted boundary we assign a metavariable arity by

$$\mathsf{ar}(\{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\mathcal{B}) = (\mathsf{c}, n)$$

where $\mathsf{c} \in \{\mathsf{Ty}, \mathsf{Tm}, \mathsf{EqTy}, \mathsf{EqTm}\}$ is the syntactic class of the non-abstracted boundary $\mathscr{b}$.

The placeholder $\square$ in a boundary $\mathscr{B}$ may be filled with an argument e, called the ***head***, to give a judgement $\mathscr{B}\boxed{e}$, as follows:

$$\begin{aligned}
(\square\ \mathsf{type})\boxed{A} &= (A\ \mathsf{type}),\\
(\square : A)\boxed{t} &= (t : A),\\
(A \equiv B\ \mathsf{by}\ \square)\boxed{e} &= (A \equiv B\ \mathsf{by}\ \star),\\
(s \equiv t : A\ \mathsf{by}\ \square)\boxed{e} &= (s \equiv t : A\ \mathsf{by}\ \star),\\
(\{x{:}A\}\mathscr{B})\boxed{\{x\}e} &= (\{x{:}A\}\mathscr{B}\boxed{e}).
\end{aligned}$$

We also define the operation $\mathscr{B}\boxed{e \equiv e'}$ which turns an object boundary $\mathscr{B}$ into an equation:

$$\begin{aligned}
(\square\ \mathsf{type})\boxed{A \equiv B} &= (A \equiv B\ \mathsf{by}\ \star),\\
(\square : A)\boxed{s \equiv t} &= (s \equiv t : A\ \mathsf{by}\ \star),\\
(\{x{:}A\}\mathscr{B})\boxed{\{x\}e \equiv \{x\}e'} &= (\{x{:}A\}\mathscr{B}\boxed{e \equiv e'}).
\end{aligned}$$

Example 3.3.1 If the symbols A and Id have arities

$$(\mathsf{Ty}, []), \quad \text{and} \quad (\mathsf{Ty}, [(\mathsf{Ty}, 0), (\mathsf{Tm}, 0), (\mathsf{Tm}, 0)]),$$

respectively, then the boundaries

$$\{x{:}\mathsf{A}\}\{y{:}\mathsf{A}\}\ \square : \mathsf{Id}(\mathsf{A}, x, y) \qquad \text{and} \qquad \{x{:}\mathsf{A}\}\{y{:}\mathsf{A}\}\ x \equiv y : \mathsf{A}\ \mathsf{by}\ \square$$

may be filled with heads $\{x\}\{y\}x$ and $\{x\}\{y\}\star$ to yield abstracted judgements

$$\{x{:}\mathsf{A}\}\{y{:}\mathsf{A}\}\ x : \mathsf{Id}(\mathsf{A}, x, y) \qquad \text{and} \qquad \{x{:}\mathsf{A}\}\{y{:}\mathsf{A}\}\ x \equiv y : \mathsf{A}\ \mathsf{by}\ \star_{\mathsf{Tm}}.$$

In Section 4.2, Θ will provide typing of metavariable and premises of an inference rule, while at the level of raw syntax it just determines metavariable arities. That is, Θ assigns the metavariable arity $\mathsf{ar}(\mathscr{B}_i)$ to M_i.

A metavariable context $\Theta = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ may be *restricted* to a metavariable context $\Theta_{(i)} = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_{i-1}{:}\mathscr{B}_{i-1}]$.

The metavariable context Θ is syntactically well formed when each $\mathscr{B}_i$ is a syntactically well-formed boundary over Σ and $\Theta_{(i)}$. In addition each $\mathscr{B}_i$ must be closed, i.e., contain no free variables.

The ***domain*** of Θ is the set $|\Theta| = \{\mathsf{M}_1, \ldots, \mathsf{M}_n\}$. The metavariable shape of Θ is the list $[\mathsf{M}_1{:}\beta_1, \ldots, \mathsf{M}_n{:}\beta_n]$, where β_i is the arity of the metavariable M_i deduced from $\mathscr{B}_i$: if $\mathscr{B}_i = \{x_1\}\ldots\{x_m\}\mathscr{b}$ then

- if $\mathscr{b} = \square\ \mathsf{type}$ then $\beta_i = (\mathsf{Ty}, m)$,
- if $\mathscr{b} = \square : A$ then $\beta_i = (\mathsf{Tm}, m)$,
- if $\mathscr{b} = A \equiv B\ \mathsf{by}\ \square$ then $\beta_i = (\mathsf{EqTy}, m)$,
- if $\mathscr{b} = a \equiv b : A\ \mathsf{by}\ \square$ then $\beta_i = (\mathsf{EqTm}, m)$.

We also define the set of the object metavariables of Θ to be

$$|\Theta|_{\text{obj}} = \{\mathsf{M}_i \mid \mathscr{B}_i \text{ is an object boundary}\}.$$

A *variable context* $\Gamma = [\mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n]$ over a metavariable context Θ is a finite list of pairs written as $\mathsf{a}_i{:}A_i$. It is considered syntactically valid when the variables $\mathsf{a}_1, \ldots, \mathsf{a}_n$ are all distinct, and for each i the type expression A_i is valid with respect to the signature and the metavariable arities assigned by Θ, and the free variables occurring in A_i are among $\mathsf{a}_1, \ldots, \mathsf{a}_{i-1}$. A variable context Γ yields a finite map, also denoted Γ, defined by $\Gamma(\mathsf{a}_i) = A_i$. The *domain* of Γ is the set $|\Gamma| = \{\mathsf{a}_1, \ldots, \mathsf{a}_n\}$.

A *context* is a pair $\Theta;\Gamma$ consisting of a metavariable context Θ and a variable context Γ over Θ. A syntactic entity is considered syntactically valid over a signature and a context $\Theta;\Gamma$ when all symbol and metavariable applications respect the assigned arities, the free variables are among $|\Gamma|$, and all bound variables are properly abstracted. It goes without saying that we always require all syntactic entities to be valid in this sense.

A *(hypothetical) judgement* has the form

$$\Theta;\Gamma \vdash \mathscr{J},$$

where $\Theta;\Gamma$ is a context and $\mathscr{J}$ is an abstracted judgement over $\Theta;\Gamma$.

In a hypothetical judgement

$$\Theta;\mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n \vdash \{x_1{:}B_1\} \cdots \{x_m{:}B_m\}\mathcal{J}$$

the hypotheses are split between the variable context $\mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n$ on the left of $\vdash$, and the abstraction $\{x_1{:}B_1\} \cdots \{x_m{:}B_m\}$ on the right. The former lists the *global* hypotheses that interact with other judgements, and the latter the hypotheses that are *local* to the judgement. In our experience such a separation is quite useful, because it explicitly marks the part of the context that is abstracted when a variable-binding symbol is applied to its arguments.

The context is split into three parts: metacontext, variable context and abstraction. When deriving judgements we can pass between the contexts using promotion from Proposition 4.3.6 and Proposition 4.3.7.

A *(hypothetical) boundary* is formed in the same fashion, as

$$\Theta;\Gamma \vdash \mathscr{B}.$$

We read it as asserting that $\mathscr{B}$ is a well-typed boundary in the context $\Theta;\Gamma$.

3.4. Instantiations

Let us spell out how how to instantiate metavariables with arguments. An *instantiation* of a metacontext $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ over a context $\Theta;\Gamma$ is a list representing a map

$$\langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle,$$

where e_i's are arguments over $\Theta;\Gamma$ such that $\mathsf{ar}(\mathscr{B}_i) = \mathsf{ar}(e_i)$. We sometimes write $I \in \mathbf{Inst}(\Xi, \Theta, \Gamma)$ when I is such an instantiation.

For $k \leq n$, define the *restriction*

$$I_{(k)} = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_{k-1} \mapsto e_{k-1} \rangle.$$

We sometimes write $I_{(\mathsf{M})}$ to indicate the initial segment up to the given metavariable $\mathsf{M} \in |I|$. We use the same notation for initial segments of sequences in general, e.g., if $\vec{x} = (x_1, \ldots, x_n)$ then $\vec{x}_{(k)} = (x_1, \ldots, x_{k-1})$.

An *instantiation I acts* on an expression e to give an expression $I_* e$ by:

$$I_*\mathsf{a} = \mathsf{a}, \qquad I_* x = x, \qquad I_* \star = \star,$$
$$I_*(\{x\}e) = \{x\}(I_* e), \qquad I_*(\mathsf{M}_i(\vec{t})) = e[I_*\vec{t}/\vec{x}] \text{ where } I(\mathsf{M}) = \{\vec{x}\}e,$$
$$I_*(\mathsf{S}(\vec{e}')) = \mathsf{S}(I_*\vec{e}'),$$

The action on abstracted judgements is given by

$$I_*(A \text{ type}) = (I_* A \text{ type}),$$
$$I_*(t : A) = (I_* t : I_* A),$$
$$I_*(A \equiv B \text{ by } \star) = (I_* A \equiv I_* B \text{ by } \star),$$
$$I_*(s \equiv t : A \text{ by } \star) = (I_* s \equiv I_* t : I_* A \text{ by } \star),$$
$$I_*(\{x{:}A\}\ \mathcal{J}) = (\{x{:}I_* A\}\ I_* \mathcal{J}).$$

We can just imagine that $I_*\square = \square$ and instantiate boundaries similarly to judgements.

An abstracted boundary may be instantiated analogously.

Given I of Ξ over $\Theta;\Gamma$, and $\Delta = [x_1{:}A_1, \ldots, x_n{:}A_n]$ over Θ such that $|\Gamma| \cap |\Delta| = \emptyset$, we define $\Gamma, I_*\Delta$ to be the variable context

$$\Gamma, x_1{:}I_* A_1, \ldots, x_n{:}I_* A_n$$

Note that $I_*\Delta$ by itself is not a valid variable context. A judgement $\Xi; \Delta \vdash \mathcal{J}$ may be instantiated to $\Theta; \Gamma, I_*\Delta \vdash I_*\mathcal{J}$. A hypothetical boundary can be instantiated analogously.

4. Type Theories

4.1. Deductive systems

We first recall the general notion of a deductive system. A (finitary) *closure rule* on a carrier set S is a pair $([p_1, \ldots, p_n], q)$ where $p_1, \ldots, p_n, q \in S$. The elements $p_1, \ldots, p_n$ are the *premises* and q is the *conclusion* of the rule. A rule may be displayed as

$$\frac{p_1 \qquad \cdots \qquad p_n}{q}.$$

A *deductive system* on a set S is a family C of closure rules on S. We say that $T \subseteq S$ is *deductively closed* for C when the following holds: for every rule $C_i = ([p_1, \ldots, p_n], q)$, if $\{p_1, \ldots, p_n\} \subseteq T$ then $q \in T$. A *derivation* with *conclusion* $q \in S$ is a well-founded tree whose root is labeled by an index i of a closure rule $C_i = ([p_1, \ldots, p_n], q)$, and whose subtrees are derivations with conclusions $p_1, \ldots, p_n$. We say that $q \in S$ is *derivable* if there exists a derivation with conclusion q. The derivable elements of S form precisely the least deductively closed subset.

All deductive systems that we shall consider will have as their carriers the set of hypothetical judgements and boundaries, as described in Section 3.3.

4.2. Raw rules

An inference rule in type theory is a template that generates a family of closure rules constituting a deductive system. In our setting, a *raw rule* is a hypothetical judgement of the form $\Theta; [] \vdash \mathcal{J}$, which we display[1] as

$$\Theta \Longrightarrow \mathcal{J}.$$

1: Sometimes we still use the traditional fraction-like presentation for clarity.

It is an *object rule* when $\mathcal{J}$ is an object judgement, and an *equality rule* when $\mathcal{J}$ is an equality judgement. The designation "raw" signals that, even though a raw rule is syntactically sensible, it may be quite unreasonable from a type-theoretic point of view.

Given a raw rule $R = (\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{b}\boxed{e})$ and an instantiation $I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$ of its premises over $\Theta; \Gamma$, the *rule instantiation* $I_* R$ is the closure rule $([p_1, \ldots, p_n, q], r)$ where p_i is

$$\Theta; \Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i},$$

q is $\Theta; \Gamma \vdash I_*\mathscr{b}$, and r is $\Theta; \Gamma \vdash I_*(\mathscr{b}\boxed{e})$. In this way a raw rule generates a family of closure rules, indexed by instantiations. The premise q is needed only in various meta-theoretic inductive arguments, as it

ensures the well-formedness of the boundary of the conclusion. In practice, we use the "economic" variant $([p_1, \ldots, p_n], r)$, which is easily seen to be admissible once Theorem 5.1.6 is established.

Example 4.2.1 We may translate traditional ways of presenting rules to raw rules easily. For example, consider the formation rule for dependent products, which might be written as

$$\frac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma, x{:}A \vdash B \ \mathsf{type}}{\Gamma \vdash \Pi(A, \{x\}B) \ \mathsf{type}}$$

To be quite precise, the above is a *family* of closure rules, indexed by meta-level parameters Γ, A, and B ranging over suitable syntactic entities. The template which generates such a family might be written as

$$\frac{\vdash \mathsf{A} \ \mathsf{type} \qquad x{:}\mathsf{A} \vdash \mathsf{B}(x) \ \mathsf{type}}{\vdash \Pi(\mathsf{A}, \{x\}\mathsf{B}(x)) \ \mathsf{type}}$$

Indeed, there is no need to mention Γ because it is always present, and we have replaced the parameters A and B with metavariables A and B (notice the change of fonts) to obtain bona-fide syntactic expressions. Next, observe that the premises amount to specifying an abstracted boundary for each metavariable, which brings us to

$$\frac{\mathsf{A}{:}(\Box \ \mathsf{type}) \qquad \mathsf{B}{:}(\{x{:}\mathsf{A}\} \ \Box \ \mathsf{type})}{\Pi(\mathsf{A}, \{x\}\mathsf{B}(x)) \ \mathsf{type}}$$

By writing everything in a single line we obtain a raw rule

$$\mathsf{A}{:}(\Box \ \mathsf{type}), \mathsf{B}{:}(\{x{:}\mathsf{A}\} \ \Box \ \mathsf{type}) \Longrightarrow \Pi(\mathsf{A}, \{x\}\mathsf{B}(x)) \ \mathsf{type}.$$

The original family of closure rules is recovered when the above raw rule is instantiated with $\langle \mathsf{A} \mapsto A, \mathsf{B} \mapsto \{x\}B \rangle$ where A and B are type expressions over (a metacontext and) a variable context Γ.

The raw rules act as templates for closure rules, so they have empty context. Once we instantiate a raw rule, we get the context Γ.

We next define congruence and metavariable rules. These feature in every type theory.

Definition 4.2.2 The *congruence rules* associated with a raw object rule R

$$\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{b}\boxed{e}$$

are closure rules, with

$$I = \langle \mathsf{M}_1 \mapsto f_1, \ldots, \mathsf{M}_n \mapsto f_n \rangle \quad \text{and} \quad J = \langle \mathsf{M}_1 \mapsto g_1, \ldots, \mathsf{M}_n \mapsto g_n \rangle,$$

of the form

$$\frac{\begin{array}{ll}\Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{f_i} & \text{for } i = 1,\dots,n\\ \Theta;\Gamma \vdash (J_{(i)*}\mathscr{B}_i)\boxed{g_i} & \text{for } i = 1,\dots,n\\ \Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{f_i \equiv g_i} & \text{for object boundary } \mathscr{B}_i\\ \Theta;\Gamma \vdash I_*B \equiv J_*B & \text{if } \mathscr{b} = (\square : B)\end{array}}{\Theta;\Gamma \vdash (I_*\mathscr{b})\boxed{I_*e \equiv J_*e}}$$

The last premise applies only if we have a rule for a term judgement rather than a type judgement.

Metavariables have their own formation and congruence rules, akin to specific and congruence rules.

Definition 4.2.3 Given a context $\Theta;\Gamma$ over Σ with

$$\Theta = [\mathsf{M}_1{:}\mathscr{B}_1,\dots,\mathsf{M}_n{:}\mathscr{B}_n],$$

and $\mathscr{B}_k = (\{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\ \mathscr{b})$, the *metavariable rules* for M_k are the closure rules of the form

$$\frac{\Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1,\dots,m \qquad \Theta;\Gamma \vdash \mathscr{b}[\vec{t}/\vec{x}]}{\Theta;\Gamma \vdash (\mathscr{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_k(\vec{t})}}$$

The congruence rules for metavariables are repeated in Figure 4.1 in order to have all the structural rules gathered in the same place.

where $\vec{x} = (x_1,\dots,x_m)$ and $\vec{t} = (t_1,\dots,t_m)$. Furthermore, if $\mathscr{b}$ is an object boundary, then the *metavariable congruence rules* for M_k are the closure rules of the form

$$\frac{\begin{array}{ll}\Theta;\Gamma \vdash s_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\dots,m\\ \Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\dots,m\\ \Theta;\Gamma \vdash s_j \equiv t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\dots,m\\ \Theta;\Gamma \vdash C[\vec{s}/\vec{x}] \equiv C[\vec{t}/\vec{x}] & \text{if } \mathscr{b} = (\square : C)\end{array}}{\Theta;\Gamma \vdash (\mathscr{b}[\vec{s}/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}) \equiv \mathsf{M}_k(\vec{t})}}$$

Recall that in the conclusion of the metavariable rule the notation $(\mathscr{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_k(\vec{t})}$ acually means $(\mathscr{b}[\vec{t}/\vec{x}])\boxed{\star}$ if $\mathscr{b}$ is an equational boundary.

where $\vec{s} = (s_1,\dots,s_m)$ and $\vec{t} = (t_1,\dots,t_m)$.

In a finitary type theory (Definition 4.4.3) presuppositions may be elided safely from the above rules to yield the following admissible "economic" versions:

$$\frac{\begin{array}{ll}\Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{f_i} & \text{for equation boundary } \mathscr{B}_i\\ \Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{f_i \equiv g_i} & \text{for object boundary } \mathscr{B}_i\end{array}}{\Theta;\Gamma \vdash (I_*\mathscr{b})\boxed{I_*e \equiv J_*e}}$$

TT-Meta-Ec

$$\frac{\Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1,\dots,m}{\Theta;\Gamma \vdash (\mathscr{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_k(\vec{t})}}$$

TT-Meta-Congr-Ec

$$\frac{\Theta;\Gamma \vdash s_j \equiv t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1,\dots,m}{\Theta;\Gamma \vdash (\mathscr{b}[\vec{s}/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}) \equiv \mathsf{M}_k(\vec{t})}}$$

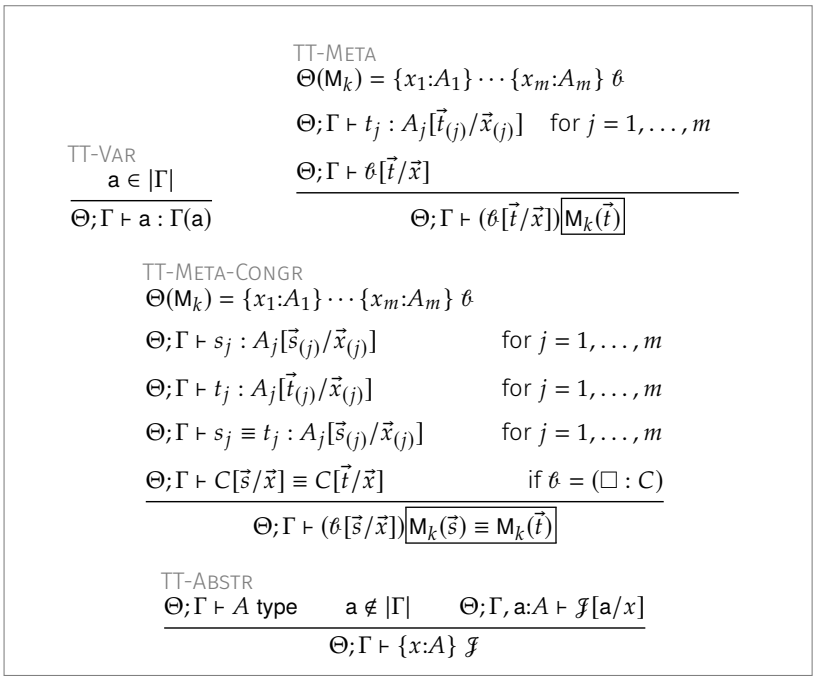


Figure 4.1.: Variable, metavariable and abstraction closure rules.

4.3. Raw type theories

A type theory in its basic form is a collection of rules that generate a deductive system. While this is too permissive a notion from a type-theoretic standpoint, it is nevertheless useful enough to deserve a name.

Definition 4.3.1 A *raw type theory* $\mathcal{T}$ over a signature Σ is a family of raw rules over Σ, called the *specific rules* of $\mathcal{T}$. The *associated deductive system* of $\mathcal{T}$ consists of:

1. the *structural rules* over Σ:
   a) the *variable*, *metavariable*, and *abstraction* rules (Definition 4.2.3, Figure 4.1),
   b) the *equality* rules, (Figure 4.2),
   c) the *boundary* rules (Figure 4.3);
2. the instantiations of the specific rules of $\mathcal{T}$;
3. for each specific object rule of $\mathcal{T}$, the instantiations of the associated congruence rule (Definition 4.2.2).

The rules of a raw type theory do not impose any conditions on the metacontexts and variable contexts, although they only ever extend variable contexts with well-formed types. When a well-formed metavariable or variable context extension is needed, the auxiliary rules in Figure 4.4 are employed. We call a judgement *strongly derivable*, when also its metacontext and context are well-formed.

The notion of storngly derivable judgements is especially used in Chapter 10, where we can only elaborate strongly derivable judgements.

Definition 4.3.2 A judgement $\Theta;\Gamma \vdash \mathcal{J}$ is *strongly derivable*, if it is derivable and $\vdash \Theta\ \mathsf{mctx}$ and $\Theta \vdash \Gamma\ \mathsf{vctx}$ are also derivable.

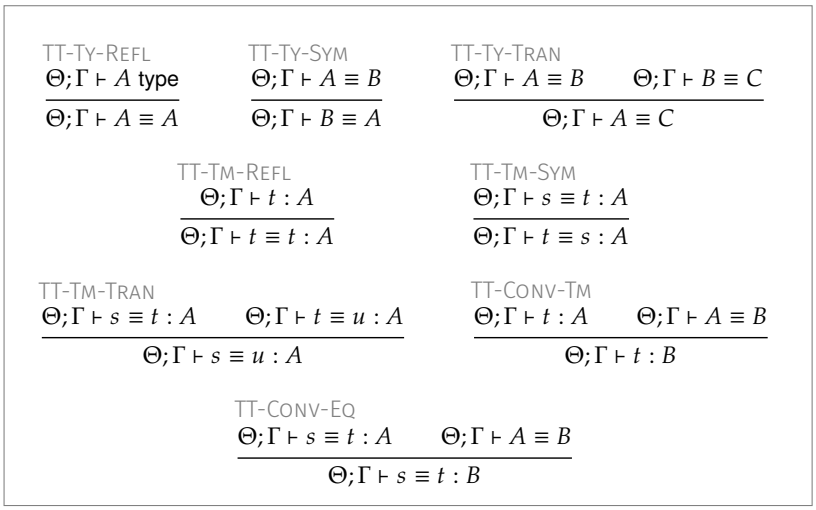


Figure 4.2.: Equality closure rules.

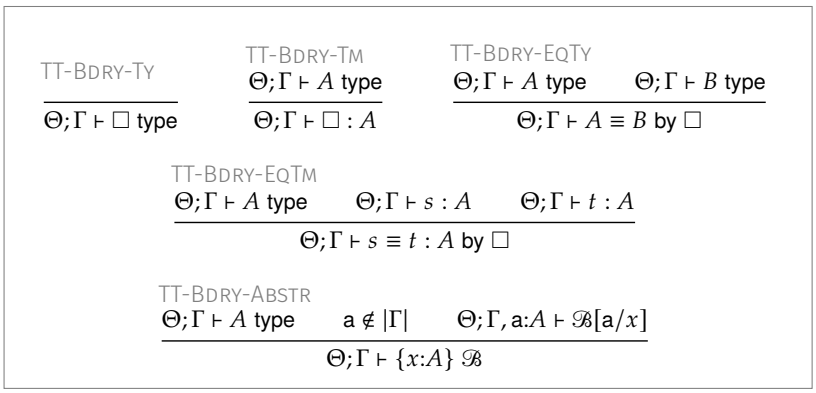


Figure 4.3.: Well-formed abstracted boundaries.

With the notion of raw type theory in hand, we may define concepts that employ derivability.

Definition 4.3.3 An instantiation $I = \langle M_1 \mapsto e_1, \ldots, M_n \mapsto e_n \rangle$ of a metacontext $\Xi = [M_1{:}\mathscr{B}_1, \ldots, M_n{:}\mathscr{B}_n]$ over $\Theta;\Gamma$ is *derivable* when $\Theta;\Gamma \vdash (I_{(k)*}\mathscr{B}_k)\boxed{e_k}$ for $k = 1, \ldots, n$.

Definition 4.3.4 Instantiations

$$I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle \qquad \text{and} \qquad J = \langle \mathsf{M}_1 \mapsto f_1, \ldots, \mathsf{M}_n \mapsto f_n \rangle$$

over Θ and Γ are *judgementally equal* when, for $k = 1, \ldots, n$, if $\mathscr{B}_k$ is an object boundary then $\Theta;\Gamma \vdash (I_{(k)*}\mathscr{B}_k)\boxed{e_k \equiv f_k}$.

Definition 4.3.5 A raw rule $\Xi \Longrightarrow \mathcal{J}$ is *derivable* when it is derivable qua judgement. It is *admissible* when, for every derivable instantiation $I = \langle M_1 \mapsto e_1, \ldots, M_n \mapsto e_n \rangle$ of Ξ over $\Theta;\Gamma$ we have $\Theta;\Gamma \vdash I_*\mathcal{J}$.

A derivation of a judgement is called *a generic derivation of a rule* if it is just a rule of the type theory (instantiated with the identity instantiation).

If I is an instantiation of $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_m{:}\mathscr{B}_m]$ over Θ and Δ, and

$$\textsc{MCtx-Empty}\ \frac{}{\vdash []\ \mathsf{mctx}} \qquad \textsc{MCtx-Extend}\ \frac{\vdash \Theta\ \mathsf{mctx} \qquad \Theta;[] \vdash \mathscr{B} \qquad \mathsf{M} \notin |\Theta|}{\vdash \langle \Theta, \mathsf{M}{:}\mathscr{B}\rangle\ \mathsf{mctx}}$$

$$\textsc{VCtx-Empty}\ \frac{\vdash \Theta\ \mathsf{mctx}}{\Theta \vdash []\ \mathsf{vctx}} \qquad \textsc{VCtx-Extend}\ \frac{\Theta \vdash \Gamma\ \mathsf{vctx} \qquad \Theta, \Gamma \vdash A\ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma|}{\Theta \vdash \langle \Gamma, \mathsf{a}{:}A\rangle\ \mathsf{vctx}}$$

Figure 4.4.: Well-formed metacontexts and variable context extensions.

J is an instantiation of Θ over $\Psi;\Gamma$ such that $|\Gamma| \cap |\Delta| = \emptyset$, their *composition* $J \circ I$ is the instantiation of Ξ over $\Psi;\Gamma, J_*\Delta$ defined by

$$(J \circ I)(\mathsf{M}) = J_*(I(\mathsf{M})).$$

Composition of instantiations is associative. It also preserves derivability, which can be proved easily once Theorem 5.1.4 is established.

It will be useful to know that derivability only needs to be checked for instantiations over the empty variable context. For this purpose, define the *promotion* of

$$\Theta;\Gamma \vdash \mathcal{J}$$

to be the judgement

$$(\Theta, \Gamma);[] \vdash \mathcal{J},$$

in which the free variables are promoted to metavariables. Note that $\vdash (\Theta, \Gamma)\ \mathsf{mctx}$ is derivable if, and only if, both $\vdash \Theta\ \mathsf{mctx}$ and $\Theta \vdash \Gamma\ \mathsf{vctx}$ are derivable.

We could obfuscate what we just said by being more precise: if

$$\Gamma = [\mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n],$$

the promotion is the judgement

$$(\Theta, \mathsf{a}'_1{:}A'_1, \ldots, \mathsf{a}'_n{:}A'_n);[] \vdash \mathcal{J}[\vec{\mathsf{a}}'/\vec{\mathsf{a}}]$$

in which $\mathsf{a}'_1, \ldots, \mathsf{a}'_n$ are fresh metavariables and $A'_i = A_i[\vec{\mathsf{a}}'_{(i)}/\vec{\mathsf{a}}_{(i)}]$.

Proposition 4.3.6 A raw type theory derives $\Theta;\Gamma \vdash \mathcal{J}$ if, and only if, it derives the promotion $(\Theta, \Gamma);[] \vdash \mathcal{J}$.

Proof. To pass between the original variable context and its promotion, swap applications of TT-Var with corresponding applications of TT-Meta. □

Another useful way of promoting variables is promoting the abstracted variables to the context. Let

$$\Theta;\Gamma \vdash \{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\mathcal{J}$$

be an abstracted judgement. Its *abstraction promotion* is the judgement

$$\Theta;(\Gamma, \mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n) \vdash \mathcal{J}[\mathsf{a}_1/x_1, \ldots, \mathsf{a}_n/x_n]$$

given that $\mathsf{a}_1, \ldots, \mathsf{a}_n \notin |\Gamma|$.

Proposition 4.3.7 A raw type theory strongly derives

$$\Theta;\Gamma \vdash \{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\mathcal{J}$$

if, and only if, it strongly derives the abstraction promotion

$$\Theta;(\Gamma, \mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n) \vdash \mathcal{J}[\mathsf{a}_1/x_1, \ldots, \mathsf{a}_n/x_n]$$

Proof. To prove the equivalence we prove both directions. First suppose

$$\Theta; \Gamma \vdash \{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\jmath$$

is strongly derivable. We proceed by induction on the derivation. In the base case the two judgements are identical. By inversion the derivation ends with TT-Abstr:

$$\frac{\Theta; \Gamma \vdash A_1 \text{ type} \qquad \mathsf{a}_1 \notin |\Gamma| \qquad \Theta; \Gamma, \mathsf{a}_1{:}A_1 \vdash (\{x_2{:}A_2\} \cdots \{x_n{:}A_n\}\jmath)[\mathsf{a}_1/x_1]}{\Theta; \Gamma \vdash \{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\jmath}$$

Using VCtx-Extend on the first premise (and the derivation of Γ being well-formed that is given by strong derivability), we obtain a derivation that $(\Gamma, \mathsf{a}_1{:}A_1)$ is also a well-formed context. We can now use induction hypothesis on the third premise to get the desired derivation.

For the other way round, suppose

$$\Theta; (\Gamma, \mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n) \vdash \jmath[\mathsf{a}_1/x_1, \ldots, \mathsf{a}_n/x_n] \tag{4.1}$$

is strongly derivable. We proceed by induction on the derivation that $(\Gamma, \mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n)$ is a well-formed context. The derivation ends with VCtx-Extend

$$\frac{\Theta \vdash \langle \Gamma, \mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_{n-1}{:}A_{n-1}\rangle \text{ vctx} \qquad \Theta, \Gamma, \mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_{n-1}{:}A_{n-1} \vdash A_n \text{ type} \qquad \mathsf{a}_n \notin |\Gamma, \mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_{n-1}{:}A_{n-1}|}{\Theta \vdash \langle \Gamma, \mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n\rangle \text{ vctx}}$$

We can use TT-Abstr on the second premise and (4.1). We then inductively proceed with the first premise, to get the desired result. □

Sometimes we need to speak of a *sub-theory* of a type theory $\mathcal{T}$: a type theory over the same signature $\Sigma_{\mathcal{T}}$ where we only take some of the specific rules. Formally we say that a ***fragment of a type theory*** $\mathcal{T}$ as a family of specific rules indexed by the set I is a type theory $\mathcal{T}^{\mathrm{fr}}$ given the restriction $\mathcal{T}\restriction_{\mathrm{I}^{\mathrm{fr}}}$ of the family $\mathcal{T}$ to a subset of the index set $\mathrm{I}^{\mathrm{fr}} \subseteq \mathrm{I}$. The derivations in the fragment are implicitly embedded in the entire type theory.

4.4. Finitary and standard type theories

Raw rules do not impose any well-typedness conditions on the premises and the conclusion. We may rectify this by requiring that their boundaries be derivable. To make this precise we recall well-founded orders.

Definition 4.4.1 A ***well-founded order*** on a set I is an irreflexive transitive relation $\sqsubset$ for which the following holds: for every subset

$A \subseteq I$

$$(\forall i \in I.(\forall j \sqsubset i.j \in A) \implies i \in A) \implies A = I.$$

We read the condition in the definition of a well-founded order as an induction principle: to show that $\forall x \in I.\Phi(x)$ we need to show for any $i \in I$ that $\Phi(i)$ holds if we assume $\Phi(y)$ for all $y \sqsubset i$.

Classically, the condition for well-founded orders is equivalent to the fact that the order contains no countable infinite descending chains. However, the formulation we use gives the inductive principle we use later on. We will also need the following lemma about chains of well-founded orders.

Lemma 4.4.2 Let $(A_n, \sqsubset_n)_{n\in\mathbb{N}}$ be well-founded orders such that for every $n \in \mathbb{N}$, it holds that $A_n \subseteq A_{n+1}$, the order $\sqsubset_n$ is included in $\sqsubset_{n+1}$ and $\sqsubset_n$ is an initial segment of $\sqsubset_{n+1}$, i.e.

$$\forall x, y \in A_{n+1}.\ (x \sqsubset_{n+1} y \wedge y \in A_n) \implies x \sqsubset_n y.$$

Then $A_\infty = \cup_{n\in\mathbb{N}} A_n$ ordered by $\sqsubset$ with

$$\forall x, y \in A_\infty.\ x \sqsubset y \Leftrightarrow \exists n \in \mathbb{N}.\ x, y \in A_n \wedge x \sqsubset_n y$$

is also a well-founded order.

Proof. In Appendix Chapter B. □

We can now finally give the definition of finitary type theories.

Definition 4.4.3 A raw rule $\Theta \implies \mathcal{b}\boxed{\mathcal{e}}$ is a *finitary rule* with respect to a raw type theory $\mathcal{T}$ when $\vdash \Theta$ mctx and $\Theta; [] \vdash \mathcal{b}$ are derivable. A *finitary type theory* is a raw type theory $\mathcal{T} = (T_i)_{i\in \mathrm{I}}$ for which there exists a well-founded order $(\mathrm{I}, \sqsubset)$ such that each T_i is finitary with respect to the fragment $(T_j)_{j\sqsubset i}$.

We use the well-founded order on the rules in the above definition to avoid possible circularities, otherwise the justification that a rule is finitary may rely on the rule itself, see [22] for further details.

[22]: Bauer et al. (2020), *A general definition of dependent type theories*

Example 4.4.4 A finitary type theory is well behaved in many respects, but may still be "non-standard". Assuming N, O and S are respectively a type constant, a term constant, and a unary term symbol, the rules

$$[] \implies \mathsf{N}\ \mathsf{type}, \qquad [] \implies \mathsf{O} : \mathsf{N}, \qquad \mathsf{n}{:}(\square : \mathsf{N}) \implies \mathsf{S}(\mathsf{S}(\mathsf{n})) : \mathsf{N}$$

constitute a finitary type theory. However, the third rule is troublesome because it posits a compound term S(S(n)).

We avoid such anomalies by requiring that object rules only ever introduce generically applied symbols. For this purpose, define a *rule-boundary* to be a hypothetical boundary of the form $\Theta; [] \vdash \mathcal{b}$, notated as $\Theta \implies \mathcal{b}$. The elements of Θ are the *premises* and $\mathcal{b}$ is the

conclusion boundary. We say that the rule-boundary is an *object rule-boundary* when $\mathcal{b}$ is a type or a term boundary, and an *equality rule-boundary* when $\mathcal{b}$ is an equality boundary. Next, given an object rule-boundary

$$\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathcal{b},$$

its *associated symbol arity* is $(c, [\mathsf{ar}(\mathscr{B}_1), \ldots, \mathsf{ar}(\mathscr{B}_n)])$, where $c \in \{\mathsf{Ty}, \mathsf{Tm}\}$ is the syntactic class of $\mathcal{b}$. Given a fresh symbol S, we assign it the associated arity and define the *associated symbol rule* to be

$$\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathcal{b}[\mathsf{S}(\widehat{\mathsf{M}}_i, \ldots, \widehat{\mathsf{M}}_n)],$$

where $\widehat{\mathsf{M}}_i$ is the *generic application* of the metavariable M_i, defined as, assuming $\mathsf{ar}(\mathscr{B}_i) = (c_i, n_i)$:

1. $\widehat{M}_i = \{x_1, \ldots, x_{n_i}\}\mathsf{M}_i(x_1, \ldots, x_{n_i})$ when $c_i \in \{\mathsf{Ty}, \mathsf{Tm}\}$,
2. $\widehat{M}_i = \{x_1, \ldots, x_{n_i}\}\star$ when $c_i \in \{\mathsf{EqTy}, \mathsf{EqTm}\}$.

Here then is our final notion of type theory.

Definition 4.4.5 A finitary type theory is *standard* if its specific object rules are symbol rules, and each symbol has precisely one associated rule.

We can also build a standard type theory iteratively, starting with the empty fragment (over the empty signautre) and then adding symbols with symbol rules, and equality rules.

The examples of type theories listed at the beginning of Chapter 3 are all standard type theories. A typical example of a finitary type theory that is not also standard is the simply typed λ-calculus, where the domain and codomain of functions are not explicitly annotated.

5. Meta-theorems

We recall from [69] meta-theorems that establish desirable structural properties of type theories. The theorems are rather expected thus verifying that our definition of a type theory is sensible. We only include the statements of the theorems, the reader can consult [69] for the proofs. In the next section we prove several additional meta-theorems that we rely on subsequently in the Part 'An equality checking algorithm'.

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

5.1. Meta-theorems about raw type theories

First we have meta-theorems about raw type theories that are concerned with syntactic manipulations. We start with derivability of renamings.

A *renaming* of an expression e is an injective map ρ with domain $\mathsf{mv}(e) \cup \mathsf{fv}(e)$ that takes metavariables to metavariables and free variables to free variables.

Proposition 5.1.1 (Renaming) If a raw type theory derives a judgement or a boundary, then it also derives its renaming.

Once derivability of a judgement is established, additional hypothesis do not break it. This materializes in ability to weaken metacontexts and variable contexts.

Proposition 5.1.2 (Weakening) For a raw type theory:

1. If $\Theta; \Gamma_1, \Gamma_2 \vdash \mathcal{J}$ then $\Theta; \Gamma_1, \mathsf{a}{:}A, \Gamma_2 \vdash \mathcal{J}$.
2. If $\Theta_1, \Theta_2; \Gamma \vdash \mathcal{J}$ then $\Theta_1, \mathsf{M}{:}\mathscr{B}, \Theta_2; \Gamma \vdash \mathcal{J}$.

An analogous statement holds for boundaries.

It is understood that in the above statements, and the subsequent ones, we tacitly assume whatever syntactic conditions are needed to ensure that all entities are well-formed. For example, in Proposition 5.1.2 we require $\mathsf{a} \notin |\Gamma_1, \Gamma_2|$ and that A be a syntactically valid type expression for $\Theta; \Gamma_1$.

A well-trained eye surely notices that there are no substitution rules in Chapter 3. That is because substitution rules are admissible as shown in the following theorem. [1]

1: Excluding substitution rules from the definition of a type theory has antoher importatnt benefit: it saves us from proving substitution lemmas for every meta-theroem we state, as Theorem 5.1.3 proves them once and forall.

Theorem 5.1.3 (Admissibility of substitution) In a raw type theory the substitution rules from Figure 5.1 are admissible.

While substitutions substitute variables for expressions, instantiations substitute metavariables. Further more, instantiations of specific rules

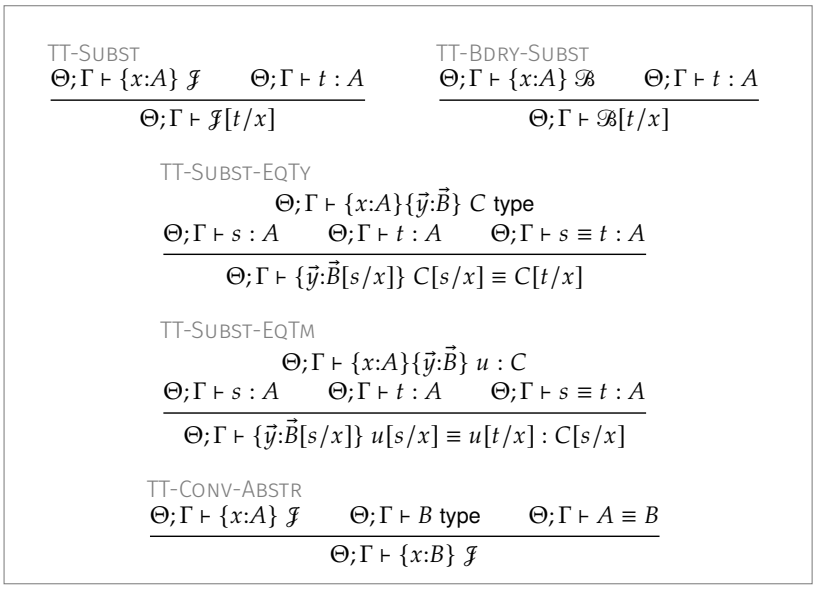


Figure 5.1.: Admissible substitution rules.

give us closure rules for our type theory. If we instantiate any meta-context with derivable expressions, i.e. if instantiation is derivable, then derivability of the judgement is preserved.

Theorem 5.1.4 (Admissibility of instantiations) In a raw type theory, let I be a derivable instantiation of Ξ over $\Theta;\Delta$. If $\Xi;\Gamma \vdash \mathcal{J}$ is derivable and $|\Delta| \cap |\Gamma| = \emptyset$ then $\Theta;\Delta, I_*\Gamma \vdash I_*\mathcal{J}$ is derivable, and similarly for boundaries.

Another important structural property of instantiations is that instantiating by judgementally equal instantiations induces judgemental equality. We make this precise in the following theorem.

Theorem 5.1.5 In a raw type theory, let I and J be judgementally equal derivable instantiations of Ξ over $\Theta;\Gamma$. Suppose $\Xi \vdash \Delta$ vctx and $|\Gamma| \cap |\Delta| = \emptyset$. If $\Xi;\Delta \vdash \mathcal{B}\boxed{e}$ is a derivable object judgement then $\Theta;\Gamma, I_*\Delta \vdash (I_*\mathcal{B})\boxed{I_*e \equiv J_*e}$ is derivable.

The condition $|\Gamma| \cap |\Delta| = \emptyset$ is inessential, as we can always apply a renaming to ensure it holds.

If a judgement is derivable, so are its presuppositions. In the language of raw type theories this materializes in the boundary being well-formed.

Theorem 5.1.6 (Presuppositivity) If a raw type theory derives $\vdash \Theta$ mctx, $\Theta \vdash \Gamma$ vctx, and $\Theta;\Gamma \vdash \mathcal{B}\boxed{e}$ then it derives $\Theta;\Gamma \vdash \mathcal{B}$.

Note that for presuppositions to hold we need to have a strongly derivable judgement. This is not surprising, as if the context is not well formed, we could derive some ill-formed presuppositions.

5.2. Meta-theorems about standard type theories

The next two theorems apply to standard type theories. The first one provides an inversion principle, and the second one guarantees that

a term has at most one type, up to judgemental equality. Both rely on a candidate type that may be read off directly from the term.

Definition 5.2.1 Let $\mathcal{T}$ be a standard type theory. The *natural type* $\tau_{\Theta;\Gamma}(t)$ of a term expression t with respect to a context $\Theta;\Gamma$ is defined by:

$$\begin{aligned}
\tau_{\Theta;\Gamma}(\mathsf{a}) &= \Gamma(a), \\
\tau_{\Theta;\Gamma}(\mathsf{M}(t_1,\ldots,t_m)) &= A[t_1/x_1,\ldots,t_m/x_m] \\
&\qquad \text{where } \Theta(\mathsf{M}) = (\{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\ \square : A) \\
\tau_{\Theta;\Gamma}(\mathsf{S}(e_1,\ldots,e_n)) &= \langle \mathsf{M}_1 \mapsto e_1,\ldots,\mathsf{M}_n \mapsto e_n\rangle_* B \\
&\qquad \text{where the symbol rule for } \mathsf{S} \text{ is} \\
&\qquad \mathsf{M}_1{:}\mathscr{B}_1,\ldots,\mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathsf{S}(\widehat{\mathsf{M}}_1,\ldots,\widehat{\mathsf{M}}_n) : B.
\end{aligned}$$

The following theorem is an inversion principle that recovers the "stump" of a derivation of a derivable object judgement.

Theorem 5.2.2 (Inversion theorem) If a standard finitary type theory derives a term judgement then it does so by a derivation which concludes with precisely one of the following rules:

1. the variable rule TT-Var,
2. the metavariable rule TT-Meta,
3. an instantiation of a symbol rule,
4. the abstraction rule TT-Abstr,
5. the term conversion rule TT-Conv-Tm of the form

$$\frac{\Theta;\Gamma \vdash t : \tau_{\Theta;\Gamma}(t) \qquad \Theta;\Gamma \vdash \tau_{\Theta;\Gamma}(t) \equiv A}{\Theta;\Gamma \vdash t : A}$$

   where $\tau_{\Theta;\Gamma}(t) \neq A$.

Since inversion is a principle that applies to more than just the term judgements in the sense of Theorem 5.2.2, we use the word "inversion" to mean the general principle and explicitly refer to the inversion theorem when we apply it in our proofs.

Finally, in a standard type theory a term has at most one type, up to judgemental equality.

Theorem 5.2.3 (Uniqueness of typing) For a standard finitary type theory:

1. If $\Theta;\Gamma \vdash t : A$ and $\Theta;\Gamma \vdash t : B$ then $\Theta;\Gamma \vdash A \equiv B$.
2. If $\Theta;[] \vdash \Gamma$ vctx and $\Theta;\Gamma \vdash s \equiv t : A$ and $\Theta;\Gamma \vdash s \equiv t : B$ then $\Theta;\Gamma \vdash A \equiv B$.

5.3. More meta-theorems

We state and prove several further meta-theorems used mostly in Part 'An equality checking algorithm'. We include the proofs as the theorems were developped for the equality checking algorithm and they are mostly not included in [69], but they are part of [24].

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

[24]: Bauer et al. (2021), *An extensible equality checking algorithm for dependent type theories*

We start by the instantiation of a natural type.

Proposition 5.3.1 Let $\mathcal{T}$ be a standard type theory and I an instantiation of Ξ over $\Theta;\Gamma$. For a term expression $\mathsf{S}(\vec{e})$ it holds that

$$I_*(\tau_{\Xi;\Delta}(\mathsf{S}(\vec{e}))) = \tau_{\Theta;\Gamma,I_*\Delta}(I_*\mathsf{S}(\vec{e})).$$

Proof. Let $\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathsf{S}(\widehat{\mathsf{M}}_1, \ldots, \widehat{\mathsf{M}}_n) : B$ be the symbol rule for S. By unfolding the definition of the natural type we have

$$\begin{aligned} I_*(\tau_{\Xi;\Delta}(\mathsf{S}(\vec{e}))) &= I_*(\langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle_* B) = \langle \mathsf{M}_1 \mapsto I_* e_1, \ldots, \mathsf{M}_n \mapsto I_* e_n \rangle_* B \\ &= \tau_{\Theta;\Gamma,I_*\Delta}(\mathsf{S}(I_*\vec{e})) = \tau_{\Theta;\Gamma,I_*\Delta}(I_*(\mathsf{S}(\vec{e}))) \end{aligned}$$

□

Note that I acts purely syntactically and needs not be derivable for the equation to hold. It is also worth pointing out that the equation does not hold for metavariable term expressions.

We now explicate two common usages of Theorem 5.2.2.

Corollary 5.3.2 In a standard type theory, suppose the rule for S is

$$\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{C}\,\boxed{\mathsf{S}(\widehat{\mathsf{M}}_1, \ldots, \widehat{\mathsf{M}}_n)}.$$

If the theory derives $\Theta;\Gamma \vdash \mathscr{C}\,\boxed{\mathsf{S}(\vec{e})}$ then it derives $\Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\,\boxed{e_i}$ for all $i = 1, \ldots, n$, where $I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$.

Proof. By Theorem 5.2.2 the judgement is derived by an application of the symbol rule for S, possibly followed by a conversion, whose premises are precisely the judgements of interest. □

Corollary 5.3.3 If a standard type theory derives $\Theta;\Gamma \vdash t : A$ then it also derives $\Theta;\Gamma \vdash t : \tau_{\Theta;\Gamma}(t)$.

Proof. By Theorem 5.2.2, either $A = \tau_{\Theta,\Gamma}(t)$ and there is nothing to prove, or the derivations ends with

$$\frac{\Theta;\Gamma \vdash t : \tau_{\Theta;\Gamma}(t) \qquad \Theta;\Gamma \vdash \tau_{\Theta;\Gamma}(t) \equiv A}{\Theta;\Gamma \vdash t : A}$$

which contains the desired equality as a subderivation. □

We next prove a statement about instantiations that needs a couple of preparatory lemmas.

Lemma 5.3.4 If a raw type theory derives $\Theta \vdash \Gamma$ vctx and $\Theta;\Gamma \vdash \mathscr{B}\boxed{e \equiv e'}$ then it derives $\Theta;\Gamma \vdash \mathscr{B}\boxed{e'}$.

Proof. We proceed by induction on the number of abstractions in the object boundary $\mathscr{B}$.

Case $\mathscr{B} = (\square$ type$)$, $e = A$ *and* $e' = B$: Theorem 5.1.6 applied to the assumption $\Theta;\Gamma \vdash A \equiv B$ by $\star$ gives $\Theta;\Gamma \vdash A \equiv B$ by $\square$, from which $\Theta;\Gamma \vdash B$ type follows by inversion.

Case $\mathscr{B} = (\square : A)$: Similar to the previous case.

Case $\mathscr{B} = (\{x{:}A\}\ \mathscr{B}')$: Inversion on the assumption $\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}'\boxed{e \equiv e'}$ gives

$$\Theta;\Gamma \vdash A \text{ type} \qquad \text{and} \qquad \Theta;\Gamma, \mathsf{a}{:}A \vdash (\mathscr{B}'[\mathsf{a}/x])\boxed{e[\mathsf{a}/x] \equiv e'[\mathsf{a}/x]}.$$

By induction hypothesis, the second judgement entails

$$\Theta;\Gamma, \mathsf{a}{:}A \vdash (\mathscr{B}'[\mathsf{a}/x])\boxed{e'[\mathsf{a}/x]},$$

which we may abstract to the desired form. □

Lemma 5.3.5 In a raw type theory, consider judgementally equal derivable instantiations I and J of Ξ over $\Theta;\Gamma$, and suppose $\Xi \vdash \Delta$ vctx and $\Xi;\Delta \vdash \mathscr{B}$ such that $|\Delta| \cap |\Gamma| = \emptyset$. If $\Theta;\Gamma, I_*\Delta \vdash (I_*\mathscr{B})\boxed{e}$ is derivable then so is $\Theta;\Gamma, I_*\Delta \vdash (J_*\mathscr{B})\boxed{e}$.

Proof. We proceed by structural induction on the derivation of $\Xi;\Delta \vdash \mathscr{B}$.

Case TT-Bdry-Ty: Trivial because $I_*\mathscr{B} = (\square$ type$) = J_*\mathscr{B}$.

Case TT-Bdry-Tm: If the derivation ends with

$$\frac{\Xi;\Delta \vdash A \text{ type}}{\Xi;\Delta \vdash \square : A}$$

then $\Theta;\Gamma, I_*\Delta \vdash I_*A \equiv J_*A$ by Theorem 5.1.5 applied to the premise, hence we may convert $\Theta;\Gamma, I_*\Delta \vdash e : I_*A$ to $\Theta;\Gamma, I_*\Delta \vdash e : J_*A$.

Case TT-Bdry-EqTy: If the derivation ends with

$$\frac{\Xi;\Delta \vdash A \text{ type} \qquad \Xi;\Delta \vdash B \text{ type}}{\Xi;\Delta \vdash A \equiv B \text{ by } \square}$$

then Theorem 5.1.5 applied to the premises gives us

$$\Theta;\Gamma, I_*\Delta \vdash I_*A \equiv J_*A \qquad \text{and} \qquad \Theta;\Gamma, I_*\Delta \vdash I_*B \equiv J_*B.$$

Together with the assumption $\Theta; \Gamma, I_*\Delta \vdash I_*A \equiv I_*B$, these suffice to derive $\Theta; \Gamma, I_*\Delta \vdash J_*A \equiv J_*B$.

Case TT-Bdry-EqTm: similar to TT-Bdry-EqTy.

Case TT-Bdry-Abstr: Suppose $e = \{x\}e'$ and the derivation ends with

$$\frac{\Xi; \Delta \vdash A \ \mathsf{type} \qquad \mathsf{a} \notin |\Delta| \qquad \Xi; \Delta, \mathsf{a}{:}A \vdash \mathscr{B}'[\mathsf{a}/x]}{\Xi; \Delta \vdash \{x{:}A\}\ \mathscr{B}'}$$

where we may assume $\mathsf{a} \notin |\Gamma|$ without loss of generality. Theorem 5.1.5 applied to the first premise derives

$$\Theta; \Gamma, I_*\Delta \vdash I_*A \equiv J_*A. \tag{5.1}$$

By inverting the assumption $\Theta; \Gamma, I_*\Delta \vdash \{x{:}I_*A\}\ (I_*\mathscr{B}')\boxed{e}$, and possibly renaming a free variable to a, we obtain

$$\Theta; \Gamma, I_*\Delta \vdash I_*A \ \mathsf{type} \quad \text{and} \quad \Theta; \Gamma, I_*\Delta, \mathsf{a}{:}I_*A \vdash ((I_*\mathscr{B}')\boxed{e})[\mathsf{a}/x].$$

Then the induction hypothesis for the second premise yields

$$\Theta; \Gamma, I_*\Delta, \mathsf{a}{:}I_*A \vdash ((J_*\mathscr{B}')\boxed{e})[\mathsf{a}/x],$$

which we may abstract to $\Theta; \Gamma, I_*\Delta \vdash \{x{:}I_*A\}\ (J_*\mathscr{B}')\boxed{e}$ and apply TT-Conv-Abstr to convert it to the desired judgement

$$\Theta; \Gamma, I_*\Delta \vdash \{x{:}J_*A\}\ (J_*\mathscr{B}')\boxed{e}.$$

The premise $\Theta; \Gamma, I_*\Delta \vdash J_*A \ \mathsf{type}$ is derived by Theorem 5.1.6 from (5.1). □

Proposition 5.3.6 In a raw type theory, consider instantiations I and J of Ξ over $\Theta; \Gamma$, such that $\vdash \Xi \ \mathsf{mctx}$ and $\Theta \vdash \Gamma \ \mathsf{vctx}$. If I is derivable and I and J are judgementally equal then J is derivable.

Proof. We prove the claim by induction on the length of Ξ. The base case is trivial. For the induction step we assume the statement, and show that is still holds when we extend Ξ, I and J by one more entry. Specifically, assume that

$$\Xi; [] \vdash \mathscr{B}, \qquad \text{and} \qquad \Theta; \Gamma \vdash (I_*\mathscr{B})\boxed{e}, \tag{5.2}$$

and if $\mathscr{B}$ is an object boundary also that

$$\Theta; \Gamma \vdash (I_*\mathscr{B})\boxed{e \equiv e'}. \tag{5.3}$$

Then we must demonstrate $\Theta; \Gamma \vdash (J_*\mathscr{B})\boxed{e'}$.

If $\mathscr{B}$ is an equality boundary then applying Lemma 5.3.5 to (5.2) gives $\Theta; \Gamma \vdash (J_*\mathscr{B})\boxed{e}$, and we are done because e and e' are the same[2].

If $\mathscr{B}$ is an object boundary then applying Lemma 5.3.5 to $\Xi; [] \vdash \mathscr{B}$ and (5.3) gives $\Theta; \Gamma \vdash (J_*\mathscr{B})\boxed{e \equiv e'}$. The derivability of $\Theta; \Gamma \vdash (J_*\mathscr{B})\boxed{e'}$ now follows from Lemma 5.3.4. □

2: Both e and e' are equal to $\star$.

Transformations of type theories

6. Overview

Setting oneself to develop a formal proof, one has to face a dilemma of which proof assistant to use. Even amongst proof assistants based on type theories there are several options [2, 5, 43, 45, 58, 78, 108, 133, 141, 146]. The choice can depend on various factors such as one's previous experience with a proof assistant, the nature of the problem to be formalised, expressivity of the underlying type theory, availability of the necessary libraries, performance of a proof assistant etc. Once the choice is made and the formalization begins it is usually not easy to change one's mind and continue the formalization in another proof assistant, as all the proof development has to be manually translated to the new type-theoretic foundation, if such a translation is even possible.

A step towards analyzing compatibility of formalizations is studying compatibility of the underlying type theories. Often a proof relies on only a fragment of the type-theoretic foundation and when such a fragment is common to another type theory, the translation could be made. Studying translations of formal systems is an age-old tale with far too many contributions to cover in this thesis. In Section 11.1 we summarize some key points relevant to our work.

In Part 'Transformations of type theories' we propose mathematical definitions of transformations of type theories, prove some basic meta-theoretical properties and exhibit their use on a few examples, the particularly substantial one being the elaboration from Chapter 10.

To accommodate for the use in proof assistants, the transformations should be syntactic in nature and general enough to be applicable to a class of type theories. We aim to adhere to the following stipulations:

1. We work in the fully general setting of raw type theories (Definition 4.3.1).
2. Transformations are syntactic in nature, so they are pertinent to possible implementations in proof assistants.
3. Preservation of derivability is inherent to transformations of type theories.
4. We can exhibit some interesting and useful examples.

Addressing the issue of translating syntax of type theories immediately stumbles upon the question of what stage the translation is performed on. Do we transform terms, types, judgements or derivations?

A trivial definition of a transformation $f : \mathcal{T} \to \mathcal{U}$ would be a map between derivations of type theories $\mathcal{T}$ and $\mathcal{U}$, without additional constraints. Such a map certainly adheres to (3) and covers (too) many examples, but is not usable in practice: in order to specify a transformation, one has to provide a derivation in the target theory $\mathcal{U}$ for

[45]: (2021), *The Coq proof assistant, version 2021.02.2*
[5]: (2021), *The Agda proof assistant*
[108]: Moura et al. (2015), "The Lean Theorem Prover (System Description)"
[133]: Sozeau et al. (2019), "Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq"
[58]: Gilbert et al. (2019), "Definitional proof-irrelevance without K"
[2]: Abel et al. (2017), "Decidability of Conversion for Type Theory in Type Theory"
[146]: Vezzosi et al. (2019), "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types"
[141]: The RedPRL Development Team (2020), *The 'redtt' theorem prover.*
[43]: Cohen et al. (2018), *The 'cubicaltt' theorem prover.*
[78]: Isaev (2021), *Arend Standard Library*

every derivation of $\mathcal{T}$. If the transformation does not preserve additional structure, this is an impossible and rather useless task.

Instead, we propose three notions of transformations of type theories. First is the *syntactic transformation* that operates purely syntactically and is specified by mapping symbols from the signature of the source theory to expressions in the target theory. Syntactic transformations preserve some of the syntactic structure, such as variable shapes, judgement forms and syntactic classes of expressions. To organize the expected properties of syntactic transformations, we describe the concept of relative monads for syntax in Chapter 7 and in Section 8.3 show that syntactic transformations are indeed an instance of such relative monad. The syntactic transformations and their relation to substitutions and instantiations were also formalized by Andrej Bauer [19].

[19]: Bauer (2021), *Syntax of dependent type theories*

We upgrade syntactic transformations to *type-theoretic transformations* to accommodate for (3). With (4) in mind we show that the *propositions as types* translation [48, 49, 74, 149] is an example of type-theoretic transformation (Example 9.3.1, Chapter A). We also prove enough meta-theoretic properties of type-theoretic transformations to show that a definitional extension of a type theory (Example 9.3.5) is conservative.

[149]: Wadler (2015), "Propositions as Types"
[48]: Curry (1934), "Functionality in Combinatory Logic"
[49]: Curry et al. (1958), *Combinatory logic. Vol. I*
[74]: Howard (1980), "The Formulae-as-Types Notion of Construction"

A major use case for our notion of type-theoretic transformations is *elaboration*. Proof assistants often allow the users to omit some typing information which can be automatically reconstructed by an elaborator. In Chapter 10 we precisely define what is an elaboration of a finitary type theory and prove the *elaboration theorem* stating that every finitary type theory can be elaborated. We also inspect some algorithmic properties of elaboration in Section 10.3.

The definition of elaboration gives rise to the third notion of a transformation of type theories, an *elaboration map*. Unlike a type-theroetic transformation which is given by a map on symbols and specific rules, an elaboration map takes as input derivations of (strongly derivable) judgements and yields the elaborated judgements, while still preserving derivability and thus adhering to (3).

6.1. Contributions

The main two contributions are defining a notion of type-theoretic transformation and proving the elaboration theorem (Theorem 10.2.1). All constructions and proofs are carried out in a constructive fashion.

To build towards a definition of type-theoretic transformation, we

- describe a general schema of relative monads for syntax (Section 7.1),
- define a notion of a *symbol renaming* (Definition 8.1.1),
- define *syntactic transformations* (Definition 8.2.1) and prove that they form a relative monad over the category of signatures (Section 8.3),

- define *type-theoretic transformations* (Definition 9.1.2) and prove that they preserve derivability (Theorem 9.1.3),
- prove some meta-theoretic properties of judgementally equal transformations (Proposition 9.1.6, Corollary 9.1.10),
- define a category of type theories and type-theoretic transformations and show that it has an initial object (Proposition 9.2.2) and coporoducts (Proposition 9.2.3),
- exhibit that *propositions as types* is an example of a type-theoretic transformation (Example 9.3.1, Chapter A) and
- using meta-theoretic properties of type-theoretic transformations prove that definitional extension is conservative (Example 9.3.5).

We give a mathematically precise definition of an *elaboration* (Definition 10.1.3) and prove that elaboration has a universal property (Theorem 10.1.7). We state and prove the *elaboration theorem* (Theorem 10.2.1, Subsection 10.2.1, Subsection 10.2.2). To analyze algorithmic properties of elaboration we

- define the *elaborator*, an algorithm for elaboration (Definition 10.3.2),
- relate decidable checking with decidable equality checking in a standard type theory (Proposition 10.3.5),
- relate computable elaboration with decidable checking (Theorem 10.3.9) and
- relate (equality) checking of a finitary type theory with (equality) checking of its elaboration (Theorem 10.3.10, Corollary 10.3.11).

7. Relative monads

Before we dig into defining transformations of type theories, we recall the categorical notion of *relative monad* [8]. The syntactic transformations in Chapter 8 can be neatly organized as a relative monad over the category of signatures. This gives us the categorical structure we expect: we get the Kleisli category over the relative monad. In order to be able to express transformations in this sense, we first remind ourselves of the definition of relative monad and then express the syntax of finitary type theories in those terms.

[8]: Altenkirch et al. (2015), "Monads need not be endofunctors"

Definition 7.0.1 Let $\mathbb{J}$ and $\mathbb{C}$ be categories. A *relative monad* on a functor $J : \mathbb{J} \to \mathbb{C}$ is given by:

- an object mapping $T : |\mathbb{J}| \to |\mathbb{C}|$,
- for any $X \in |\mathbb{J}|$ a map $\eta_X \in \mathbb{C}(J\ X, T\ X)$ (the unit),
- for any $X, Y \in |\mathbb{J}|$ and $f \in \mathbb{C}(J\ X, T\ Y)$, a map $f^\dagger \in \mathbb{C}(T\ X, T\ Y)$ (the Kleisli extension)

such that the following conditions hold:

1. for any $X, Y \in |\mathbb{J}|$ and $f \in \mathbb{C}(J\ X, T\ Y)$ it holds that $f^\dagger \circ \eta_X = f$,
2. for any $X \in |\mathbb{J}|$ it holds that $\eta_X^\dagger = \mathrm{id}_{TX}$,
3. for any $X, Y, Z \in |\mathbb{J}|$, $f \in \mathbb{C}(J\ X, T\ Y)$ and $g \in \mathbb{C}(J\ Y, T\ Z)$ it holds that $(g^\dagger \circ f)^\dagger = g^\dagger \circ f^\dagger$.

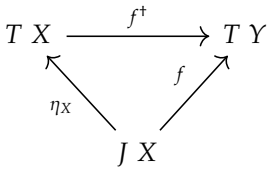
We think of the functor J as a sort of embedding of $\mathbb{J}$ into $\mathbb{C}$.

For the relation to the monads and more examples please consult [8]

.

We think of the functor J as a sort of embedding of $\mathbb{J}$ into $\mathbb{C}$. The relative monad laws are then quite natural:

1. For the first law we can draw the commutative diagram



2. For the second law:

$$T\ X \xrightarrow{\eta_X^\dagger = \mathrm{id}_{TX}} T\ X$$

$$J\ X \xrightarrow{\eta_X} T\ X$$

   Lifting the unit gives us the identity.
3. For the third equation:

$$J\ T \xrightarrow{f^\dagger} T\ Y \xrightarrow{g^\dagger} T\ Z$$

$$J\ X \xrightarrow{f} T\ Y \xrightarrow{g^\dagger} T\ Z$$

$$J\ Y \xrightarrow{g} T\ Z$$

From the top row of the diagram we get the composition of lifts $g^\dagger \circ f^\dagger$. However we can also lift the second row to get $(g^\dagger \circ f)^\dagger$ and the second law makes sure we get the same map.

The following proposition is stated in [8], but for completeness we include the proof as well.

[8]: Altenkirch et al. (2015), "Monads need not be endofunctors"

Proposition 7.0.2 Let T be a relative monad on a functor $J\colon \mathbb{J} \to \mathbb{C}$. The condition for the relative monad further ensure that:

1. $T\colon \mathbb{J} \to \mathbb{C}$ is functorial: T maps $f \in \mathbb{J}(X, Y)$ to $T\ f \in \mathbb{C}(T\ X, T\ Y)$ defined by $T\ f = (\eta_Y \circ J\ f)^\dagger$,
2. η is natural,
3. $(-)^\dagger$ is natural.

Proof. To verify, that T is functorial we check functor laws:

- The identity law: We need to verify that $T\ \mathrm{id}_X = \mathrm{id}_{TX}$. Compute

$$T\ \mathrm{id}_X = (\eta_X \circ J\ \mathrm{id})^\dagger = (\eta_X \circ \mathrm{id}_{JX})^\dagger = \eta_X^\dagger = \mathrm{id}_X$$

  Where the last equation holds by the second equation for relative monads.
- The composition law: let $X, Y, Z \in \mathbb{J}$ and $f \in \mathbb{J}(X, Y)$ and $g \in \mathbb{J}(Y, Z)$. We want to show $T\ g \circ T\ f = T\ (g \circ f)$, which computes to

$$(\eta_Z \circ (J\ g))^\dagger \circ (\eta_Y \circ (J\ f))^\dagger = (\eta_Z \circ (J\ (g \circ f)))^\dagger. \tag{7.1}$$

  We can draw the following diagram:

$$\begin{array}{ccccc} J\ X & \xrightarrow{J\ f} & J\ Y & \xrightarrow{J\ g} & J\ Z \\ \downarrow{\scriptstyle \eta_X} & & \downarrow{\scriptstyle \eta_Y} & & \downarrow{\scriptstyle \eta_Z} \\ T\ X & \xrightarrow{(\eta_Y \circ J\ f)^\dagger} & T\ Y & \xrightarrow{(\eta_Z \circ J\ g)^\dagger} & T\ Z \end{array}$$

  The diagram clearly commutes due to the third equation of relative monads. We can now compute $(\eta_Z \circ J\ g)^\dagger \circ (\eta_Y \circ J\ f)^\dagger$, which by the third law for relative monads equals

$$(\eta_Z \circ J\ g)^\dagger \circ (\eta_Y \circ J\ f)^\dagger = ((\eta_Z \circ J\ g)^\dagger \circ (\eta_Y \circ J\ f))^\dagger.$$

  We can now apply associativity of composition and the first law of relative monads to get

$$((\eta_Z \circ J\ g)^\dagger \circ (\eta_Y \circ J\ f))^\dagger = (\eta_Z \circ J\ g \circ J\ f)^\dagger$$

which by functoriality of J gives us

$$((\eta_Z \circ J\ g)^\dagger \circ (\eta_Y \circ J\ f))^\dagger = (\eta_Z \circ J\ (g \circ f))^\dagger.$$

By transitivity we get the desired equation.

Now we can check that η is indeed natural transformation. Let $X, Y \in \mathbb{J}$ and $f \in \mathbb{J}(X, Y)$. We need to check that the following diagram commutes:

$$\begin{array}{ccc} T\ X & \xrightarrow{(\eta_Y \circ J\ f)^\dagger} & T\ Y \\ {\scriptstyle \eta_X}\uparrow & & \uparrow{\scriptstyle \eta_Y} \\ J\ X & \xrightarrow{J\ f} & J\ Y \end{array}$$

The equation

$$(\eta_Y \circ J\ f)^\dagger \circ \eta_X = \eta_Y \circ J\ f$$

holds by applying the first law for relative monads to the morphism $\eta_Y \circ J\ f$.

To check that the Kleisli extension $(-)^\dagger$ is functorial, let $X, Y \in \mathbb{J}$ and $k \in \mathbb{C}(J\ X, T\ Y)$. The diagram

$$\begin{array}{ccc} T\ X & \xrightarrow{k^\dagger} & T\ Y \\ {\scriptstyle \eta_X}\uparrow & & \uparrow{\scriptstyle \mathrm{id}_{T\ Y}} \\ J\ X & \xrightarrow{k} & T\ Y \end{array}$$

commutes by the first law for relative monads. □

7.1. Relative monads for syntax

Since we would like to study the relative monad of syntactic expressions, we take a look at a special form of monads that covers our examples. These relative monads for syntax (for the case of substitutions and instantiations) were studied and formalized by Gaïa Loutchmia, Jure Taslak, Danel Ahman and Andrej Bauer [93].

[93]: Loutchmia et al. (2021), *Formalization of simple type theory*

We take substitution as the motivating example. The substitution is usually a map $\sigma\colon \gamma \to \delta$, where γ determines variables and δ determines expressions. But we also have other parameters, like information about sorts, arities of variables, etc. We form two categories:

- $\mathbb{C}$ for the category of objects that change. In the case of substitution that would be objects like γ and δ, and renaming of variables as morphisms,
- $\mathbb{D}$ for the rest of the parameters.

Just like metacontexts can be reduced to metavariable shapes, variable contexts Γ can be reduced to *variable shapes* of the form $\gamma = |\Gamma|$.

Together $\gamma \in |\mathbb{C}|$ and $\vartheta \in |\mathbb{D}|$ should suffice to determine a free algebra of expressions $\mathsf{Expr}(\gamma, \vartheta)$. We then formulate a functor

$$J : \mathbb{C} \to \mathbf{Set}^{\mathbb{D}}$$

That maps γ to a functor, which for $\vartheta \in \mathbb{D}$ gives us the *set of variables in γ adhering to restrictions of ϑ*. The relative monad over J is then

$$\begin{aligned} T : \mathbb{C} &\to \mathbf{Set}^{\mathbb{D}} \\ \gamma &\mapsto (\vartheta \mapsto \mathsf{Expr}(\gamma, \vartheta)) \end{aligned}$$

The functorial action of T tells how renamings act on expressions.

For every $\gamma \in \mathbb{C}$ the unit η of the relative monad T is a natural transformation

$$\eta_\gamma : J\ \gamma \to T\ \gamma$$

between functors from $\mathbb{D} \to \mathbf{Set}$. For every $\vartheta \in \mathbb{D}$ we get a morphism

$$\eta_\gamma^\vartheta : J\ \gamma\ \vartheta \to T\ \gamma\ \vartheta$$

that maps a name x to an expression x. In the case of substitution this would map a variable name to the variable of that name as a syntactic expression. The Kleisli extension then captures the action of substitution: for a substitution $\sigma : J\ \gamma \to T\ \delta$, the lifting $\sigma^\dagger : T\ \gamma \to T\ \delta$ is the action of σ on expressions.

We now have the general idea, what is the shape of relative monads for syntax. To give concrete examples that arise from the syntax of type theory, we first need to specify the following categories:

- $\mathbf{Class} = \{\mathsf{Ty}, \mathsf{Tm}, \mathsf{EqTy}, \mathsf{EqTm}\}$ is the discrete category of syntactic classes.
- $\mathbf{VShape}$ is the category of variable shapes with variable renamings. The category has coproducts (for variable context extension).
- $\mathbf{MShape}$ is the category of metavariable shapes and renamings.
- $\mathbf{Sig}$ is the category of signatures. The objects are lists of the form $[\mathsf{S}_1{:}\alpha_1, \ldots, \mathsf{S}_n{:}\alpha_n]$, where α_i is the arity of symbol S_i. The morphisms are symbol renamings (Definition 8.1.1).

Recall that a *metavariable renaming*

$$r : \vartheta \to \Xi$$

maps a metavariable M of ϑ to a metavariable in Ξ so that

$$\mathsf{ar}(r(\mathsf{M})) = \mathsf{ar}(\mathsf{M})$$

We then have a functor

$$\begin{aligned} \mathbf{Expr} &: \mathbf{Sig} \times \mathbf{Class} \times \mathbf{MShape} \times \mathbf{VShape} \to \mathbf{Set} \\ \mathbf{Expr}\ (\Sigma, \mathsf{c}, \vartheta, \gamma) &= \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma) \end{aligned}$$

A *symbol renaming* is a map between signatures $f : \Sigma_1 \to \Sigma_2$ that preserves arities of symbols: for every $\mathsf{S} \in \Sigma_1$ it holds that

$$\mathsf{ar}(\mathsf{S}) = \mathsf{ar}(f(\mathsf{S})).$$

that gives the set of syntactic expressions. For readability we shall use notation $\mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma)$ for the functor as well. It's easy to see this really is a functor: since $\mathbf{Class}$ is a discrete category a morphism in the product category $\mathbf{Sig} \times \mathbf{Class} \times \mathbf{MShape} \times \mathbf{VShape}$ is a quadruple $(r_s, \mathrm{id}_\mathsf{c}, r_m, r_v)$, where r_s, r_m and r_v are renamings of symbols, metavariables and variables respectively. Since renamings preserve arities, the action of the functor on morphism $(r_s, \mathrm{id}_\mathsf{c}, r_m, r_v)$ is just a function between sets of expressions $\mathsf{Expr}_{\Sigma_1}(\mathsf{c}, \vartheta_1; \gamma_1) \to \mathsf{Expr}_{\Sigma_2}(\mathsf{c}, \vartheta_2; \gamma_2)$ that acts by renaming according to r_s, r_m and r_v.

By varying the choice for the categories $\mathbb{C}$ and $\mathbb{D}$ we can now build the relative monads for the syntactic structures we are interested in.

7.2. The relative monad of substitutions

For the relative monad of substitutions take $\mathbb{C} = \mathbf{VShape}$ and $\mathbb{D} = \mathbf{Sig} \times \mathbf{Class} \times \mathbf{MShape}$. We define the functor J as

$$J : \mathbf{VShape} \to \mathbf{Set}^{\mathbf{Sig}\times\mathbf{Class}\times\mathbf{MShape}}$$
$$J\ \gamma\ (\Sigma, \mathsf{c}, \vartheta) = \begin{cases} \{x \mid x \in |\gamma|\} & \text{if } \mathsf{c} = \mathsf{Tm} \\ \emptyset & \text{otherwise} \end{cases}$$

so it produces the set of all variables in the given variable context (if we are looking at the correct class Tm). The functorial action on the variable renamings (the morphisms of $\mathbf{VShape}$) gives the action of those renamings on sets of variables. The relative monad T is defined by

$$T : \mathbf{VShape} \to \mathbf{Set}^{\mathbf{Sig}\times\mathbf{Class}\times\mathbf{MShape}}$$
$$T\ \gamma\ (\Sigma, \mathsf{c}, \vartheta) = \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma)$$

For $\gamma \in |\mathbf{VShape}|$ the unit η_γ is given by

$$\eta_\gamma : J\ \gamma \to T\ \gamma$$
$$\eta_\gamma\ (\Sigma, \mathsf{c}, \vartheta) = x \mapsto x$$

Since $J\ \gamma$ and $T\ \gamma$ are elements of $\mathbf{Set}^{\mathbf{Sig}\times\mathbf{Class}\times\mathbf{MShape}}$, the unit is a natural transformation between these two functors, so a function between sets $J\ \gamma\ (\Sigma, \mathsf{c}, \vartheta)$ and $T\ \gamma\ (\Sigma, \mathsf{c}, \vartheta)$. If $\mathsf{c} \neq \mathsf{Tm}$ the set $J\ \gamma\ (\Sigma, \mathsf{c}, \vartheta)$ is empty, therefore we get the empty map. Otherwise the unit is the function from $\{x \mid x \in |\gamma|\}$ to $\mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma)$ that takes a variable name and makes the variable expression of that name.

The Kleisli extension for a substitution $\sigma : J\ \gamma \to T\ \delta$ is given by the action of substitution on expressions:

$$\sigma^\dagger : T\ \gamma \to T\ \delta$$
$$\sigma^\dagger\ (\Sigma, \mathsf{c}, \vartheta) : \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma) \to \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \delta)$$
$$\sigma^\dagger\ (\Sigma, \mathsf{c}, \vartheta)\ e = \sigma_* e$$

We will not show the conditions that this is indeed relative monad. Instead this has been formalized by Loutchmia, Taslak, Ahman and Bauer [93].

[93]: Loutchmia et al. (2021), *Formalization of simple type theory*

7.3. The relative monad of instantiations

There are many similarities between substitutions and instantiations. While substitutions replace variables with expressions, instantiations replace metavariables. They both act on expressions in a similar way.

We can also promote variables to metavariables according to Proposition 4.3.6 and then use instantiations instead of substitutions. We get the relative monad of instantiations similarly to the relative monad of substitutions, we just now have to pay attention to the arities of metavariables.

For the category $\mathbb{C}$ we take $\mathbf{MShape}$ and for $\mathbb{D} = \mathbf{Sig} \times \mathbf{Class} \times \mathbf{VShape}$. The functor J is given by

$$J : \mathbf{MShape} \to \mathbf{Set}^{\mathbf{Sig}\times\mathbf{Class}\times\mathbf{VShape}}$$
$$J\ \vartheta\ (\Sigma, \mathsf{c}, \gamma) = \{\mathsf{M} \mid \mathsf{M} : (\mathsf{c}, \gamma) \in \vartheta\}$$

The relative monad T is again the monad of expressions:

$$T : \mathbf{MShape} \to \mathbf{Set}^{\mathbf{Sig}\times\mathbf{Class}\times\mathbf{VShape}}$$
$$T\ \vartheta\ (\Sigma, \mathsf{c}, \gamma) = \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma)$$

The unit embeds metavariables in expression as their generic applications,

$$\eta_\vartheta : J\ \vartheta \to T\ \vartheta$$
$$\eta_\vartheta\ (\Sigma, \mathsf{c}, \gamma) : J\ \vartheta\ (\Sigma, \mathsf{c}, \gamma) \to T\ \vartheta\ (\Sigma, \mathsf{c}, \gamma)$$
$$\eta_\vartheta\ (\Sigma, \mathsf{c}, \gamma)\ \mathsf{M} = \widehat{\mathsf{M}}$$

and the Kleisli extension for an instantiation $I : J\ \vartheta \to T\ \Psi$ is the action of instantiation:

$$I^\dagger : T\ \vartheta \to T\ \Psi$$
$$I^\dagger\ (\Sigma, \mathsf{c}, \gamma) : T\ \vartheta\ (\Sigma, \mathsf{c}, \gamma) \to T\ \Psi\ (\Sigma, \mathsf{c}, \gamma)$$
$$I^\dagger\ (\Sigma, \mathsf{c}, \gamma)\ e = I_* e$$

The relative monad for instantiations has also been formalized by Loutchmia, Taslak, Ahman and Bauer [93].

[93]: Loutchmia et al. (2021), *Formalization of simple type theory*

8. Syntactic transformations

When considering transformations between type theories there are several options. We can map symbols to symbols, symbols to expressions or even judgements to other kinds of judgements as for example in [152]. We will not consider much the later kind, but will give a few observations in Section 9.3 and Chapter 11.

[152]: Winterhalter et al. (2019), "Eliminating Reflection from Type Theory"

8.1. Symbol renamings

We first consider transformations of the syntax. Following the example for variables and metavariables for which we have renamings, we can also rename symbols.

Definition 8.1.1 A *symbol renaming* is a map between signatures $f : \Sigma_1 \to \Sigma_2$ that preserves arities of symbols: for every $\mathsf{S} \in \Sigma_1$ it holds that

$$\mathsf{ar}(\mathsf{S}) = \mathsf{ar}(f(\mathsf{S})).$$

The composition $g \circ f$ of symbol renamings $f : \Sigma_1 \to \Sigma_2$ and $g : \Sigma_2 \to \Sigma_2$ as maps is again a symbol renaming, because the arities are preserved by both f and g. The composition is obviously associative and we have the identity symbol renaming that is just the identity map. This makes $\mathbf{Sig}$ into a category of signatures.

While specifying the arities, the natural numbers are used to denote the shapes of variables instead of $\mathbb{V}\mathsf{Shape}$. Since the variables are bound, the representations are equivalent.

Example 8.1.2 When we think of renamings, we usually see them as maps that justify our perceptions that the names of variables, metavariables or symbols are not important. For example, it does not matter what we call the type of natural numbers, as long as it has the correct rules. We can easily think of two signatures, like

$$\begin{aligned}
\Sigma_1 = [&\mathbb{N} : (\mathsf{Ty}, []),\\
&\mathsf{z} : (\mathsf{Tm}, []),\\
&\mathsf{s} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])]\\
\Sigma_2 = [&\mathsf{Nat} : (\mathsf{Ty}, []),\\
&\mathsf{zero} : (\mathsf{Tm}, []),\\
&\mathsf{succ} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])]
\end{aligned}$$

which both represent natural numbers. The symbol renaming

$$\begin{aligned}
\Sigma_1 &\to \Sigma_2\\
\mathbb{N} &\mapsto \mathsf{Nat}\\
\mathsf{z} &\mapsto \mathsf{zero}\\
\mathsf{s} &\mapsto \mathsf{succ}
\end{aligned}$$

exhibits the correspondence between the two signatures. Since this is a syntactic transformation between signatures, we are not yet concerned with the rules and derivations – we have not even given the rules that govern these symbols. However, in reasonable cases the rules given for the two signatures will also correspond to each other nicely, after all we are expecting to see two representations of natural numbers.

Since the symbol renaming is defined on the level of syntax, we are not dealing with the question of derivability yet. However, for that reason we can get some somewhat less intuitive examples of renamings that one does not encounter in practice.

Example 8.1.3 Suppose we have the following signatures:

$$\begin{aligned}
\Sigma_1 = &[\Pi : (\mathsf{Ty}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1)]),\\
&\mathbb{N} : (\mathsf{Ty}, []),\\
&\mathsf{zero} : (\mathsf{Tm}, []),\\
&\mathsf{succ} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])]\\
\Sigma_2 = &[\Sigma : (\mathsf{Ty}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1)]),\\
&\mathbb{Z} : (\mathsf{Ty}, []),\\
&\mathsf{zero} : (\mathsf{Tm}, []),\\
&\mathsf{succ} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])\\
&\mathsf{pred} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])]
\end{aligned}$$

Σ_1 is the signature with Π-types (dependent product types) and the type of natural numbers $\mathbb{N}$ with zero for zero and succ for successor. Σ_2 is the signature with Σ-types (dependent sum types) and the type of integers $\mathbb{Z}$ with zero for zero, succ for successor and pred for predecessor. We can form the symbol renaming

$$\begin{aligned}
\Sigma_1 &\longrightarrow \Sigma_2\\
\Pi &\longmapsto \Sigma\\
\mathbb{N} &\longmapsto \mathbb{Z}\\
\mathsf{zero} &\longmapsto \mathsf{zero}\\
\mathsf{succ} &\longmapsto \mathsf{succ}
\end{aligned}$$

that exploits the fact that Π and Σ have the same arities, and similarly for $\mathbb{N}$ and $\mathbb{Z}$. Note that in practice the signature Σ_1 would also contain symbols like λ of arity $(\mathsf{Tm}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1), (\mathsf{Tm}, 1)])$ and app with arity $(\mathsf{Tm}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1), (\mathsf{Tm}, 0), (\mathsf{Tm}, 0)])$, while Σ_2 would be extended by pair with arity $(\mathsf{Tm}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 0), (\mathsf{Tm}, 0), (\mathsf{Tm}, 0)])$ and the projections fst and snd of arity $(\mathsf{Tm}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1), (\mathsf{Tm}, 0)])$. Since the arities of these additional symbols do not match, we cannot extend the given symbol renaming.

Symbol renamings are a first definition of a syntactic transformation of type theories, but it is a very restrictive one. Since the arities need to be preserved, the useful renamings are namely the ones that preserve the entire structure and just take care of different names for the same symbols.

8.2. Syntactic transformation

If we again follow the example of variables and metavariables, the notions of substitution and instantiation are the ones that are useful in practice, because we can replace a (meta)variable with an arbitrary expression. We can define a similar concept for signatures of symbols as well, namely the *syntactic transformations*.

Definition 8.2.1 A *syntactic transformation* $f : \Sigma_1 \to \Sigma_2$ is a map that takes a symbol $\mathsf{S} \in \Sigma_1$ to an expression in $\mathsf{Expr}_{\Sigma_2}(\mathsf{c}, \vartheta; [])$, where $\mathsf{ar}(\mathsf{S}) = (\mathsf{c}, \vartheta)$.

The metavariable shape ϑ arises from the arity of the symbol S. Again if the symbol arity determines bound variables using natural numbers, we can get the metavariable shape that uses VShape.

With syntactic transformations we still preserve some of the structure. Symbols are mapped into expressions of the same class and the arity defines the metavariable shape of the expressions. Even though we map into the empty variable shape (empty variable context), we can always apply weakening to get the expression in the desired variable context.

Sometimes we need to speak about the metavariables from the arity of a symbol. In those cases we can generate a metavariable shape with metavariables $\mathsf{M}_1, \ldots, \mathsf{M}_n$ from the arity of the symbol.

Example 8.2.2 A special example of a syntactic transformation is the *identity transformation* $\mathrm{id}_\Sigma : \Sigma \to \Sigma$, which maps every symbol S to its generic application $\mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})$.

Example 8.2.3 We can also think of symbol renamings as syntactic transformations in the following sense: if $f : \Sigma_1 \to \Sigma_2$ is a symbol renaming, then $f' : \Sigma_1 \to \Sigma_2$ with

$$f'(\mathsf{S}) = (f(\mathsf{S}))(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})$$

where $\mathsf{M}_1, \ldots, \mathsf{M}_n$ are metavariables from the metavariable shape of the arity of the symbol $f(\mathsf{S})$.

Example 8.2.4 Extending Example 8.1.3 suppose we have the two signatures

$$\begin{aligned}\Sigma_1 = [&\Pi : (\mathsf{Ty}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1)]),\\ &\lambda : (\mathsf{Tm}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1), (\mathsf{Tm}, 1)])\\ &\mathsf{app} : (\mathsf{Tm}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1), (\mathsf{Tm}, 0), (\mathsf{Tm}, 0)])\\ &\mathbb{N} : (\mathsf{Ty}, []),\\ &\mathsf{zero} : (\mathsf{Tm}, []),\\ &\mathsf{succ} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])]\end{aligned}$$

and

$$\begin{aligned}\Sigma_2 = [&\Sigma : (\mathsf{Ty}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1)]),\\ &\mathsf{pair} : (\mathsf{Tm}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1), (\mathsf{Tm}, 0), (\mathsf{Tm}, 0)]),\\ &\mathsf{fst} : (\mathsf{Tm}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1), (\mathsf{Tm}, 0)]),\\ &\mathsf{snd} : (\mathsf{Tm}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1), (\mathsf{Tm}, 0)]),\\ &\mathbb{Z} : (\mathsf{Ty}, []),\\ &\mathsf{zero} : (\mathsf{Tm}, []),\\ &\mathsf{succ} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])\\ &\mathsf{pred} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])]\end{aligned}$$

Σ_1 is the signature with Π-types (dependent product types) with λ for forming dependent functions and app for function applications, and with the type of natural numbers $\mathbb{N}$ with zero for zero and succ for successor. Σ_2 is the signature with Σ-types (dependent sum types) with the usual symbols pair, fst and snd for forming pairs and projections and with the type of integers $\mathbb{Z}$ with zero for zero, succ for successor and pred for predecessor. We can extend the symbol renaming from Example 8.1.3 in the following way:

$$\begin{aligned}\Sigma_1 &\longrightarrow \Sigma_2\\ \Pi &\longmapsto \Sigma(\mathsf{M}_1, \{x\}\mathsf{M}_2(x))\\ \lambda &\longmapsto \mathsf{zero}\\ \mathsf{app} &\longmapsto \mathsf{pair}(\mathsf{M}_1, \{x\}\mathsf{M}_2(x), \mathsf{M}_4, \mathsf{M}_4)\\ \mathbb{N} &\longmapsto \mathbb{Z}\\ \mathsf{zero} &\longmapsto \mathsf{zero}\\ \mathsf{succ} &\longmapsto \mathsf{succ}(\mathsf{M}_1)\end{aligned}$$

where $\mathsf{M}_1, \ldots, \mathsf{M}_4$ are metavariables from the appropriate metavariable shape of the symbol arity. This is a valid syntactic transformation, but a rather unusual one. We expect λ to be mapped into a term expression of a Π-type, but instead it is mapped to (a weakened) zero, which we expect to have type $\mathbb{Z}$. Note that on the syntactic level there are no typing rules and we have not given any either. The syntactic transformation simply needs to respect syntactic classes, arities and scoping, which in our case it does.

Preserving derivability is a desired condition we use in the definiton of type-theoretic transformations (Definition 9.1.2).

Other interesting examples of syntactic transformations can be found in Section 9.3 as they are also type-theoretic transformations (Definition 9.1.2).

Every syntactic transformation acts on expressions in a natural way, lifting it to a map between expressions over signatures of type theories.

Definition 8.2.5 The *action* of a syntactic transformation $f : \Sigma_1 \longrightarrow \Sigma_2$ is a map

$$f_* : \mathsf{Expr}_{\Sigma_1}(\mathsf{c}, \vartheta; \gamma) \longrightarrow \mathsf{Expr}_{\Sigma_2}(\mathsf{c}, \vartheta; \gamma)$$

for every syntactic class c, metavariable shape ϑ and variable shape

γ given by

$$f_*\mathsf{a} = \mathsf{a}, \qquad f_* x = x, \qquad f_* \star = \star,$$
$$f_*(\{x\}e) = \{x\}(f_* e),$$
$$f_*(\mathsf{M}(t_1, \ldots, t_m)) = \mathsf{M}(f_* t_1, \ldots, f_* t_m)$$
$$f_*(\mathsf{S}(e_1, \ldots, e_n)) = \langle \mathsf{M}_1 \mapsto f_* e_1, \ldots \mathsf{M}_n \mapsto f_* e_n \rangle_* f(\mathsf{S})$$

where $\mathsf{M}_1, \ldots, \mathsf{M}_n$ are the metavariables from the metavariable shape ϑ in $\mathsf{ar}(\mathsf{S}) = (\mathsf{cl}(\mathsf{S}), \vartheta)$.

The action of syntactic transformation is similar to the action of substitutions and instantiations, but now the symbols are replaced instead of (meta)variables.

Definition 8.2.6 For syntactic transformations

$$f : \Sigma_1 \to \Sigma_2$$
and
$$g : \Sigma_2 \to \Sigma_3$$

the *composition* transformation $g \circ f : \Sigma_1 \to \Sigma_3$ is defined by

$$(g \circ f)(\mathsf{S}) = g_*(f\ \mathsf{S}).$$

The composition is well-defined, because the action preserves the metavariable shapes and variable shapes.

8.3. The relative monad of syntactic transformations

Just like the substitutions and instantiations were examples of relative monads for syntax described in Section 7.1, the newly defined syntactic transformations are as well. This yields many nice properties of syntactic transformations and organizes them in a structured concept. In the remainder of this section we explain how syntactic transformations form a relative monad and prove the necessary equations. On the way to this result we encounter many nice and somewhat expected lemmas about the interaction of syntactic transformations with the rest of the syntax (instantiations, substitutions, etc.). Some of these results were formalized by Andrej Bauer [19].

[19]: Bauer (2021), *Syntax of dependent type theories*

Following the schema from Section 7.1 we take $\mathbb{C} = \mathbf{Sig}$ and $\mathbb{D} = \mathbf{Class} \times \mathbf{MShape} \times \mathbf{VShape}$. We define the functor J as

$$J : \mathbf{Sig} \to \mathbf{Set}^{\mathbf{Class} \times \mathbf{MShape} \times \mathbf{VShape}} \tag{8.1}$$
$$J\ \Sigma\ (\mathsf{c}, \vartheta, \gamma) = \{\mathsf{S} \mid \mathsf{S} : (\mathsf{c}, \vartheta) \text{ appears in } \Sigma\}$$

The relative monad T is defined by

$$T : \mathbf{Sig} \to \mathbf{Set}^{\mathbf{Class} \times \mathbf{MShape} \times \mathbf{VShape}}$$
$$T\ \Sigma\ (\mathsf{c}, \vartheta, \gamma) = \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma)$$

Just like the unit η is the generic application of a metavariable in the case of instantiations, with syntactic transformations η is the generic

application of the symbols.

$$\eta_\Sigma : J\ \Sigma \to T\ \Sigma$$
$$\eta_\Sigma\ (\mathsf{c}, [\mathsf{M}_1{:}(\mathsf{c}_1, \beta_1), \ldots, \mathsf{M}_n{:}(\mathsf{c}_n, \beta_n)], \gamma)\ \mathsf{S} = \mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})$$

The unit η is in our case a natural transformation between functors J and T.

This is precisely the identity syntactic transformation id_Σ. To define the Kleisli extension, we take a syntactic transformation

$$f : J\ \Sigma \to T\ \Omega$$
$$f_{(\mathsf{c},\vartheta,\gamma)} : \mathsf{S} \mapsto f_{(\mathsf{c},\vartheta,\gamma)}\ \mathsf{S} \in \mathsf{Expr}_\Omega(\mathsf{c}, \vartheta; \gamma)$$

Syntactic transformation from definition Definition 8.2.1 maps to expressions in the empty variable shape. However, we implicitly weaken them into shape γ here.

and lift it with the action

$$f^\dagger : T\ \Sigma \to T\ \Omega$$
$$f^\dagger_{(\mathsf{c},\vartheta,\gamma)}\ e = f_* e.$$

To verify that this structure is indeed a relative monad, we need to check several facts that are gathered in the lemmas that follow. First, we establish that J from (8.1) is indeed a functor. Recall that $\mathbf{Sig}$ is a category with signatures and symbol renamings. We have not defined the action of J on a renaming $r : \Sigma \to \Omega$, but we take the obvious candidate

$$J\ r : J\ \Sigma \to J\ \Omega$$
$$(J\ r)_{(\mathsf{c},\vartheta,\gamma)} : \{\mathsf{S} \mid \mathsf{S} : (\mathsf{c}, \vartheta) \in \Sigma\} \to \{\mathsf{S}' \mid \mathsf{S}' : (\mathsf{c}, \vartheta) \in \Omega\}$$
$$(J\ r)_{(\mathsf{c},\vartheta,\gamma)}\ \mathsf{S} = r(\mathsf{S})$$

Since $J\ \Sigma$ and $J\ \Omega$ are elements of functor categories, $J\ r$ is a natural transformation, so an arrow for each point $(\mathsf{c}, \vartheta, \gamma)$.

We check the functor conditions:

$$(J\ \mathrm{id})_{(\mathsf{c},\vartheta,\gamma)}\ \mathsf{S} = \mathrm{id}(\mathsf{S})$$

so indeed $J\ \mathrm{id} = \mathrm{id}$. To get the equation for the composition

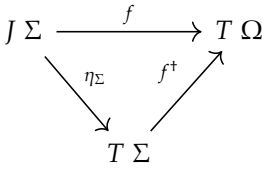
$$J\ (m \circ r) = (J\ m) \circ (J\ r)$$

we compute both sides:

$$(J\ (m \circ r))_{(\mathsf{c},\vartheta,\gamma)}\ \mathsf{S} = (m \circ r)(\mathsf{S}) = m(r(\mathsf{S}))$$
$$((J\ m) \circ (J\ r))_{(\mathsf{c},\vartheta,\gamma)}\ \mathsf{S} = (J\ m)_{(\mathsf{c},\vartheta,\gamma)}(r(\mathsf{S})) = m(r(\mathsf{S}))$$

and we get the same result. In the second case we used the definition of the vertical composition of natural transformations.

Now we check the equations for the relative monad from Definition 7.0.1. For the first equation we need to check that for all signatures $\Sigma, \Omega \in \mathbf{Sig}$ and for every syntactic transformation $f : J\ \Sigma \to T\ \Omega$ the following diagram commutes.



We compute for $(\mathsf{c}, \vartheta, \gamma) \in \mathbf{Class} \times \mathbf{MShape} \times \mathbf{VShape}$ and a symbol S with arity $\mathsf{ar}(\mathsf{S}) = (\mathsf{c}, \vartheta)$, where $\vartheta = [\mathsf{M}_1{:}\beta_1, \ldots, \mathsf{M}_n{:}\beta_n]$

$$\begin{aligned}(f^\dagger \circ \eta_\Sigma)_{(\mathsf{c},\vartheta,\gamma)}\ \mathsf{S} &= f^\dagger_{(\mathsf{c},\vartheta,\gamma)}(\eta_\Sigma\ (\mathsf{c}, \vartheta, \gamma)\ \mathsf{S}) \\ &= f^\dagger_{(\mathsf{c},\vartheta,\gamma)}(\mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})) \\ &= \langle \mathsf{M}_1 \mapsto f^\dagger(\widehat{\mathsf{M}_1}), \ldots, \mathsf{M}_n \mapsto f^\dagger(\widehat{\mathsf{M}_n})\rangle_* f(\mathsf{S}) \\ &= \langle \mathsf{M}_1 \mapsto \widehat{\mathsf{M}_1}, \ldots, \mathsf{M}_n \mapsto \widehat{\mathsf{M}_n}\rangle_* f(\mathsf{S}) \\ &= f(\mathsf{S})\end{aligned}$$

Note that $f^\dagger(\widehat{M}) = f^\dagger(\{\vec{x}\}\mathsf{M}(\vec{x})) = \{\vec{x}\}(f^\dagger(\mathsf{M}(\vec{x}))) = \{\vec{x}\}\mathsf{M}(f^\dagger(\vec{x})) = \widehat{\mathsf{M}}$

Since the unit η_Σ acts like the identity syntactic transformation, this equation proves that the identity transformation is the right neutral element for syntactic transformations.

The instantiation $\langle \mathsf{M} \mapsto \widehat{\mathsf{M}}\rangle$ is the identity instantiation and that it acts trivially on expressions. This is formalized in [19].

For the second equation of a relative monad we need to prove that for every signature $\Sigma \in \mathbf{Sig}$ it holds that

$$\eta^\dagger_\Sigma = \mathrm{id}_{T\Sigma}$$

where the righthand side is the identity map on expressions. Since the unit η acts like the identity syntactic transformation we first need the following lemma.

Lemma 8.3.1 Action preserves identity: the action of the identity transformation $\mathrm{id}_\Sigma : \Sigma \to \Sigma$ is the identity function on expressions: Let $e \in \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma)$. Then

$$\mathrm{id}_* e = e.$$

Proof. By structural induction on the expression.

Case $e = \mathsf{a}$ *or* $e = \star$ *or* $e = x$: Trivially, by the definition of action $\mathrm{id}_* e = e$.

Case $e = \{x\}e'$: By definition of action we have $\mathrm{id}_*(\{x\}e') = \{x\}(\mathrm{id}_* e')$, which by induction hypothesis equals $\{x\}e'$.

Case $e = \mathsf{M}(t_1, \ldots, t_m)$: By induction hypothesis $\mathrm{id}_* t_i = t_i$. We compute by the definition of the action on a metavariable $\mathrm{id}_*(\mathsf{M}(t_1, \ldots, t_m)) = \mathsf{M}(\mathrm{id}_* t_1, \ldots, \mathrm{id}_* t_m) = \mathsf{M}(t_1, \ldots, t_m)$.

Case $e = \mathsf{S}(e_1, \ldots, e_n)$: We compute

$$\mathrm{id}_* \mathsf{S}(e_1, \ldots, e_n) = \langle \mathsf{M}_1 \mapsto \mathrm{id}_* e_1, \ldots, \mathsf{M}_n \mapsto \mathrm{id}_* e_n\rangle_* \mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})$$

which by definition of instantiation and induction hypothesis equals $\mathsf{S}(e_1, \ldots, e_n)$. □

We can now conclude the proof of the second equation for relative monads by observing that

$$\begin{aligned}(\eta^\dagger_\Sigma)_{(\mathsf{c},\vartheta,\gamma)} &: \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma) \to \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma) \\ (\eta^\dagger_\Sigma)_{(\mathsf{c},\vartheta,\gamma)}\ e &= (\eta_\Sigma\ (\mathsf{c}, \vartheta, \gamma))_* e = \mathrm{id}_* e = e\end{aligned}$$

[19]: Bauer (2021), *Syntax of dependent type theories*

using Lemma 8.3.1 in the last step as the unit coincides with the identity syntactic transformation.

For the third equation of relative monads we first need a couple of lemmas.

Lemma 8.3.2 Action interacts with substitution: Let $f : \Sigma_1 \to \Sigma_2$ be a syntactic transformation and e and t well-formed expressions in the syntax over Σ_1. The following equation holds

$$f_*(e[t/x]) = (f_*e)[f_*t/x].$$

Proof. We proceed by induction on the expression e.

Case $e = \mathsf{a}$ *or* $e = y$ *or* $e = \star$: Since in this case $e[t/x] = e$ and the syntactic transformation f acts trivially, both sides of the equation compute to the same value e.

Case $e = x$: We compute

$$f_*(e[t/x]) = f_*t = x[f_*t/x] = (f_*e)[f_*t/x].$$

Case $e = \mathsf{M}(t_1, \ldots, t_m)$: Compute

$$\begin{aligned} f_*(\mathsf{M}(t_1, \ldots, t_m)[t/x]) &= f_*(\mathsf{M}(t_1[t/x], \ldots, t_m[t/x])) \\ &= \mathsf{M}(f_*(t_1[t/x]), \ldots, f_*(t_m[t/x])) \end{aligned}$$

we use the induction hypothesis to get

$$\begin{aligned} \mathsf{M}(f_*(t_1[t/x]), \ldots, f_*(t_m[t/x])) &= \mathsf{M}((f_*t_1)[f_*t/x], \ldots, (f_*t_m)[f_*t/x]) \\ &= (f_*(\mathsf{M}(t_1, \ldots, t_m)))[f_*t/x]. \end{aligned}$$

thus concluding the chain of equations that derives the desired result.

Case $e = \mathsf{S}(e_1, \ldots, e_n)$: Similarly to the metavariable case we will use the induction hypothesis. However, the action of the syntactic transformation f is a bit more involved in this case.

$$\begin{aligned} f_*(\mathsf{S}(e_1, \ldots, e_n)[t/x]) &= f_*(\mathsf{S}(e_1[t/x], \ldots, e_n[t/x])) \\ &= \langle \mathsf{N}_1 \mapsto f_*(e_1[t/x]), \ldots, \mathsf{N}_n \mapsto f_*(e_n[t/x]) \rangle_* f(\mathsf{S}) \end{aligned}$$

now we can use the induction hypothesis and get

$$\begin{aligned} &\langle \mathsf{N}_1 \mapsto f_*(e_1[t/x]), \ldots, \mathsf{N}_n \mapsto f_*(e_n[t/x]) \rangle_* f(\mathsf{S}) \\ &= \langle \mathsf{N}_1 \mapsto (f_*e_1)[f_*t/x], \ldots, \mathsf{N}_n \mapsto (f_*e_n)[f_*t/x] \rangle_* f(\mathsf{S}) \\ &= f_*(\mathsf{S}(e_1, \ldots, e_n))[f_*t/x] \end{aligned}$$

and we can conclude the proof by transitivity of equality. □

Instantiations commute with substitutions: For an appropriate instantiation of metavariables $\vec{\mathsf{N}}$, and expressions e' and e'' it holds that $\langle \vec{\mathsf{N}} \mapsto \vec{e}[e'/x] \rangle_* e''[e'/x] = (\langle \vec{\mathsf{N}} \mapsto \vec{e} \rangle_* e'')[e'/x]$. Since the bound variable x does not appear in $f(\mathsf{S})$ on the left it holds that $f(\mathsf{S})[f_*t/x] = f(\mathsf{S})$.

Lemma 8.3.3 Action interacts with instantiation: Let $f : \Sigma_1 \to \Sigma_2$ be a syntactic transformation. For every syntax class c, metavariable

shapes $[\mathsf{M}_1{:}\beta_1, \ldots, \mathsf{M}_n{:}\beta_n]$ and ϑ, variable shapes γ and δ, expression $t \in \mathsf{Expr}_{\Sigma_1}(\mathsf{c}, [\mathsf{M}_1{:}\beta_1, \ldots, \mathsf{M}_n{:}\beta_n]; \delta)$

$$f_*(\langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle_* t) = \langle \mathsf{M}_1 \mapsto f_* e_1, \ldots, \mathsf{M}_n \mapsto f_* e_n \rangle_* (f_* t)$$

where $\langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$ is an instantiation over $\vartheta; \gamma$.

Proof. By structural induction on the expression t. Let

$$I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$$

and

$$I^f = \langle \mathsf{M}_1 \mapsto f_* e_1, \ldots, \mathsf{M}_n \mapsto f_* e_n \rangle.$$

Cases $t = \mathsf{a}$, $t = \star$ or $t = x$: Because instantiations and syntactic transformations act trivially on t, both sides of the desired equation compute to t.

Case $t = \{x\}t'$: Action on the abstraction is trivial, then we use the induction hypothesis on t'.

Case $t = \mathsf{S}(t_1, \ldots, t_m)$: Compute

$$\begin{aligned} f_*(I_* \mathsf{S}(t_1, \ldots, t_m)) &= f_*(\mathsf{S}(I_* t_1, \ldots, I_* t_m)) \\ &= \langle \mathsf{N}_1 \mapsto f_*(I_* t_1), \ldots, \mathsf{N}_m \mapsto f_*(I_* t_m) \rangle_* \, f(\mathsf{S}) \end{aligned}$$

We then apply the induction hypothesis on $f_*(I_* t_1), \ldots, f_*(I_* t_m)$ to get

$$\langle \mathsf{N}_1 \mapsto I^f_* t_1, \ldots, \mathsf{N}_m \mapsto I^f_* t_n) \rangle_* \, f(\mathsf{S})$$

For appropriate instantiations of metavariables $\vec{\mathsf{N}}$ and $\vec{\mathsf{M}}$, and expression e' it holds that $\langle \vec{\mathsf{N}} \mapsto \langle \vec{\mathsf{M}} \mapsto \vec{e} \rangle_* \vec{t} \rangle_* e' = \langle \vec{\mathsf{M}} \mapsto \vec{e} \rangle_* (\langle \vec{\mathsf{N}} \mapsto \vec{t} \rangle_* e')$.

which can be factored as the composition of instantiations

$$I^f_* (\langle \mathsf{N}_1 \mapsto f_* t_1, \ldots, \mathsf{N}_m \mapsto f_* t_m \rangle_* (f(\mathsf{S})))$$

which is equal to $I^f_* (f_*(\mathsf{S}(t_1, \ldots, t_m)))$ by definition of action of f.

Case $t = \mathsf{M}_i(t_1, \ldots, t_m)$: We compute

$$f_*(I_* \mathsf{M}_i(t_1, \ldots, t_m)) = f_*(e_i[I_* t_1 / x_1, \ldots, I_* t_m / x_m])$$

Using Lemma 8.3.2 we get

$$f_*(e_i[I_* t_1 / x_1, \ldots, I_* t_m / x_m]) = (f_* e_i)[f_*(I_* t_1)/x_1, \ldots, f_*(I_* t_m)/x_m]$$

By induction hypothesis this equals

$$\begin{aligned} (f_* e_i)[f_*(I_* t_1)/x_1, \ldots, f_*(I_* t_m)/x_m] &= (f_* e_i)[I^f_* t_1 / x_1, \ldots, I^f_* t_m / x_m] \\ &= I^f_* (\mathsf{M}_i(t_1, \ldots, t_m)) \end{aligned}$$

and we conclude the proof by combining the equalities. □

The following lemma is the essence of the proof of the third equation for relative monads.

Lemma 8.3.4 Action preserves composition: Let $f\colon \Sigma_1 \to \Sigma_2$ and $g\colon \Sigma_2 \to \Sigma_3$ be syntactic transformations. Then for every syntax class c, metavariable shape ϑ, variable shape γ and expression $e \in \mathsf{Expr}_{\Sigma_1}(\mathsf{c}, \vartheta; \gamma)$ it holds that

$$(g \circ f)_* e = g_*(f_* e).$$

Proof. By structural induction on the expression e.

Cases $e = \mathsf{a}$, $e = \star$ *or* $e = x$: By definition of action

$$(g \circ f)_* e = e = g_*(f_* e)$$

which is the desired result.

Case $e = \{x\}e'$: By definition of action we have

$$(g \circ f)_*(\{x\}e') = \{x\}((g \circ f)_* e'),$$

which by induction hypothesis equals $\{x\}(g_*(f_* e))$. This by definition of action on abstraction equals $g_*(f_*(\{x\}e'))$.

Case $e = \mathsf{M}(t_1, \ldots, t_n)$: By definition

$$(g \circ f)_* \mathsf{M}(t_1, \ldots, t_n) = \mathsf{M}((g \circ f)_* t_1, \ldots, (g \circ f)_* t_n)$$

which by induction hypothesis equals $\mathsf{M}(g_*(f_* t_1), \ldots, g_*(f_* t_n))$ and this is by definition of action equal to $g_*(f_*(\mathsf{M}(t_1, \ldots, t_n)))$.

Case $e = \mathsf{S}(e_1, \ldots, e_n)$: We compute

$$\begin{aligned}(g \circ f)_* \mathsf{S}(e_1, \ldots, e_n) &= \langle \mathsf{M}_1 \mapsto (g \circ f)_* e_1, \ldots, \mathsf{M}_n \mapsto (g \circ f)_* e_n \rangle_*((g \circ f)(\mathsf{S})) \\ &= \langle \mathsf{M}_1 \mapsto g_*(f_* e_1), \ldots, \mathsf{M}_n \mapsto g_*(f_* e_n) \rangle_*(g_*(f\mathsf{S})) \\ &= g_*(\langle \mathsf{M}_1 \mapsto f_* e_1, \ldots, \mathsf{M}_n \mapsto f_* e_n \rangle_*(f\mathsf{S})) \\ &= g_*(f_*(\mathsf{S}(e_1, \ldots, e_n))\end{aligned}$$

using the induction hypothesis and definition of composition of syntactic transformations at the second step and Lemma 8.3.3 at the third step. □

To prove the third equation let $\Sigma, \Omega, \chi \in \mathrm{Sig}$ and let $f\colon J\,\Sigma \to T\Omega$ and $g\colon J\Omega \to T\chi$ be syntactic transformations. We need to prove that

$$(g^\dagger \circ f)^\dagger = g^\dagger \circ f^\dagger.$$

For a syntactic class c, metavariable shape ϑ, variable shape γ and expression $e \in \mathsf{Expr}_\Sigma(\mathsf{c}, \vartheta; \gamma)$ we compute

$$\begin{aligned}(g^\dagger \circ f)^\dagger_{(\mathsf{c},\vartheta,\gamma)}\, e &= ((g^\dagger \circ f)_{(\mathsf{c},\vartheta,\gamma)})_* e \\ &= (g_{(\mathsf{c},\vartheta,\gamma)})_*((f_{(\mathsf{c},\vartheta,\gamma)})_* e) \\ &= ((g^\dagger \circ f^\dagger)_{(\mathsf{c},\vartheta,\gamma)})_* e\end{aligned}$$

using Lemma 8.3.4 in the second step.

Corollary 8.3.5 Composition is associative: For syntactic transformations

$$\Sigma_1 \xrightarrow{f} \Sigma_2 \xrightarrow{g} \Sigma_3 \xrightarrow{h} \Sigma_4$$

the following holds

$$h \circ (g \circ f) = (h \circ g) \circ f.$$

Proof. We compute on a symbol $\mathsf{S} \in \Sigma_1$

$$(h \circ (g \circ f))(\mathsf{S}) = h_*((g \circ f)\ \mathsf{S}) = h_*(g_*(f\ \mathsf{S}))$$

and we can use Lemma 8.3.4 to derive the desired result

$$h_*(g_*(f\ \mathsf{S})) = (h \circ f)_*(f\ \mathsf{S}) = ((h \circ g) \circ f)(\mathsf{S}). \qquad \square$$

The action of syntactic transformation $f : \Sigma \to \Omega$ on abstracted judgements is given by

$$\begin{aligned}
f_*(A\ \mathsf{type}) &= (f_*A\ \mathsf{type}),\\
f_*(t : A) &= (f_*t : f_*A),\\
f_*(A \equiv B\ \mathsf{by}\ \star) &= (f_*A \equiv f_*B\ \mathsf{by}\ \star),\\
f_*(s \equiv t : A\ \mathsf{by}\ \star) &= (f_*s \equiv f_*t : f_*A\ \mathsf{by}\ \star),\\
f_*(\{x{:}A\}\ \mathcal{J}) &= (\{x{:}f_*A\}\ f_*\mathcal{J}).
\end{aligned}$$

The action on an abstracted boundary is defined analogously.

On a metacontext $\Theta = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ the action of f is defined by

$$f_*\Theta = [\mathsf{M}_1{:}f_*\mathscr{B}_1, \ldots, \mathsf{M}_n{:}f_*\mathscr{B}_n].$$

The action on the metacontext Θ is well-defined, since the action of f on expressions preserve metavariable shape: for $i = 1, \ldots, n$ the boundary $\mathscr{B}_i$ is over the metavariable shape $[\mathsf{M}_1{:}\beta_1, \ldots, \mathsf{M}_{i-1}{:}\beta_{i-1}]$ and since the action $f_*\mathscr{B}_1$ preserves metavariable shape, the boundary of M_i is well formed in the new metacontext.

For a variable context $\Gamma = [x_1{:}A_1, \ldots, x_n{:}A_n]$ over a metacontext Θ we define

$$f_*\Gamma = [x_1{:}f_*A_1, \ldots, x_n{:}f_*A_n]$$

which is well-formed over the metacontext $f_*\Theta$. The syntactic transformation f acts on a hypothetical judgement $\Theta; \Gamma \vdash \mathcal{J}$ with

$$f_*\Theta; f_*\Gamma \vdash f_*\mathcal{J}$$

and analogously on a hypothetical boundary.

Corollary 8.3.6 Let $f : \Sigma_1 \to \Sigma_2$ be a syntactic transformation, $\Theta = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ and Ξ metacontexts over signature Σ_1 and $\Theta; \Gamma \vdash \mathcal{J}$ a judgement in the syntax over Σ_1. Suppose I is an instantiation of Θ over $\Xi; \Delta$. Then

$$f_*(I_*(\Theta; \Gamma \vdash \mathcal{J})) = I^f_*(f_*\Theta; f_*\Gamma \vdash f_*\mathcal{J})$$

where $I^f = \langle \mathsf{M}_1 \mapsto f_*(I(\mathsf{M}_1)), \ldots, \mathsf{M}_n \mapsto f_*(I(\mathsf{M}_n)) \rangle$ is an instantiation of $f_*\Theta$ over $f_*\Xi; f_*\Delta$.

Proof. The proof is a direct application of Lemma 8.3.3: We compute

$$f_*(I_*(\Theta; \Gamma \vdash \mathcal{J})) = f_*(\Xi; \Delta, I_*\Gamma \vdash I_*\mathcal{J}) = f_*\Xi; f_*\Delta, f_*(I_*\Gamma) \vdash f_*(I_*\mathcal{J}) \quad (8.2)$$

By Lemma 8.3.3 we have that $f_*(I_*\mathcal{J}) = I^f_*(f_*\mathcal{J})$ and by applying the same lemma component-wise we get that $f_*(I_*\Gamma) = I^f_*(f_*\Gamma)$. We can thus continue (8.2) and finish the proof by

$$= f_*\Xi; f_*\Delta, I^f_*(f_*\Gamma) \vdash I^f_*(f_*\mathcal{J}) = I^f_*(f_*\Theta; f_*\Gamma \vdash f_*\mathcal{J}). \qquad \square$$

9. Type-theoretic transformations

While syntactic transformations are a good notion of a transformation of type theories, they do not consider a very important aspect: derivability. In general, a syntactic transformation can map a derivable judgement to a non-derivable one, or even one that would not be considered sensible as seen in Example 8.2.4. Clearly, derivability structure needs to be taken into account to have a more reasonable definition of a transformation between type theories.

9.1. The definition and properties of type-theoretic transformations

A transformation of type theories is expected to preserve derivability: it maps derivable judgements to derivable judgements. There are again several ways to ensure that, depending on what we take as a transformation on the syntactic level and what kind of restrictions we impose on it. We propose two notions, the later being our go-to definition of a transformation between type theories.

Definition 9.1.1 A *simple transformation* from type theory $\mathcal{T}$ to $\mathcal{U}$ is a symbol renaming $f : \Sigma_{\mathcal{T}} \to \Sigma_{\mathcal{U}}$ of signatures such that for every specific rule $\Theta \Longrightarrow \jmath$ in $\mathcal{T}$ there is a derivation $\mathcal{D}$ of $f_*\Theta \Longrightarrow f_*\jmath$ in $\mathcal{U}$.

The notion of the simple map is similar to the simple maps defined in [22].

[22]: Bauer et al. (2020), *A general definition of dependent type theories*

Just like in the case of symbol renamings, simple transformations are quite restrictive, since we can only rename symbols in a derivable way. A much more flexible notion arises from the syntactic transformation.

Definition 9.1.2 A *type-theoretic transformation* $f : \mathcal{T} \to \mathcal{U}$ is a syntactic transformation $f : \Sigma_{\mathcal{T}} \to \Sigma_{\mathcal{U}}$ such that for each rule $\Theta \Longrightarrow \jmath$ in $\mathcal{T}$ we have a derivation $\mathcal{D}$ of $f_*\Theta \Longrightarrow f_*\jmath$ in $\mathcal{U}$.

The notion of type-theoretic transformation corresponds to a raw type theory map from [22].

In the definitions of simple transformations and type-theoretic transformations we could alternatively state the condition that $f_*\Theta \Longrightarrow f_*\jmath$ is *derivable* in $\mathcal{U}$. In principle if we know a judgement is derivable, we can always obtain a derivation by running a brute-force proof search. However, since our proofs are inductive and to avoid unnecessary uses of axiom of choice, we make the derivations a part of the specification of the type-theoretic transformation. This way the proofs remain constructive as we are always able to give explicit constructions, even when they depend on the derivations.

We will again face the dilemma of explicit derivation in Section 10.3.

A simple example of a type-theoretic transformation is the *identity transformation* $\mathrm{id}\colon \mathcal{T} \to \mathcal{T}$, which acts as the identity syntactic transformation on the underlying signature $\mathrm{id}\colon \Sigma_{\mathcal{T}} \to \Sigma_{\mathcal{T}}$. Indeed, if we take a rule $\Theta \Longrightarrow \mathcal{J}$ in the theory $\mathcal{T}$, by Lemma 8.3.1 $\mathrm{id}_*(\Theta; [] \vdash \mathcal{J}) = \Theta; [] \vdash \mathcal{J}$, which can be derived in one step using the same rule. More interesting examples are shown in Section 9.3.

The condition for type-theoretic transformations ensures the desired property that the transformations preserve derivability, as we can see from the next theorem.

Theorem 9.1.3 Let $f\colon \mathcal{T} \to \mathcal{U}$ be a type-theoretic transformation. For a derivable judgement $\Theta; \Gamma \vdash \mathcal{J}$ in $\mathcal{T}$, the action of f

$$f_*\Theta; f_*\Gamma \vdash f_*\mathcal{J}$$

is derivable in $\mathcal{U}$ and similarly for boundaries.

Proof. We proceed by induction on the derivation and consider cases for the last rule applied.

Cases TT-Abstr, TT-Ty-Refl, TT-Ty-Sym, TT-Ty-Tran, TT-Tm-Refl, TT-Tm-Sym, TT-Tm-Tran, TT-Conv-Tm, TT-Conv-Eq: Apply the induction hypothesis on the premises of the rule and use the same rule again.

Cases TT-Bdry-Abstr, TT-Bdry-EqTm, TT-Bdry-EqTy, TT-Bdry-Tm, TT-Bdry-Ty : Again apply the induction hypothesis on the premises of the rule and use the same rule to get the desired conclusion.

Case TT-Var: We get the desired result by applying the rule TT-var again, since the action of the transformation does not change the variable name and it acts on the variable context appropriately.

Cases TT-Meta, TT-Meta-Congr: Apply the induction hypothesis on the premises of the rule. Since the action of transformation does not change the metavariable name and acts on the metacontext appropriately, we can apply the rule TT-Meta again.

Case Specific rule $\Xi \Longrightarrow \mathcal{J}'$ *in the theory* $\mathcal{T}$: We have a derivable instantiation I of metavariables Ξ over $\Theta; \Gamma$ such that

$$I_*(\Xi; [] \vdash \mathcal{J}') = (\Theta; \Gamma \vdash \mathcal{J}).$$

Since f is a type-theoretic transformation, we have a derivation of $f_*\Xi; [] \vdash f_*\mathcal{J}'$ in the theory $\mathcal{U}$. By induction hypothesis the instantiation $I^f = f_*I_*$ is derivable in $\mathcal{U}$[1]. By Theorem 5.1.4 the judgement $I^f_*(f_*\Xi; [] \vdash f_*\mathcal{J}')$ is derivable, which is by Corollary 8.3.6 equal to the desired judgement $f_*\Theta; f_*\Gamma \vdash f_*\mathcal{J}$. □

1: Instantiation I^f maps the metavariables of Ξ to the premises of the (derived) rule $f_*\Xi; [] \vdash f_*\mathcal{J}'$. The premises are derivable by induction hypothesis, so the instantiation I^f is derivable as well.

Corollary 9.1.4 The composition of type theoretic transformations as syntactic transformations is also a type-theoretic transformation.

Proof. Let $f : \mathcal{T} \to \mathcal{U}$ and $g : \mathcal{U} \to \mathcal{V}$ be type-theoretic transformations. To show that the composition of underlying syntactic transformations $g \circ f$ is a type theoretic transformation, we need to check the following condition: for every specific rule $\Theta \Longrightarrow \mathcal{J}$ in $\mathcal{T}$ we have a derivation of

$$(g \circ f)_*\Theta; [] \vdash (g \circ f)_*\mathcal{J}$$

which is equal to

$$g_*(f_*\Theta); [] \vdash g_*(f_*\mathcal{J})$$

by Lemma 8.3.4. Since f is a type-theoretic transformation, we have a derivation of $f_*\Theta; []; f_*\mathcal{J}$. Because g is also a type-theoretic transformation, by Theorem 9.1.3 we have a derivation of $g_*(f_*\Theta); [] \vdash g_*(f_*\mathcal{J})$, which concludes the proof. □

Corollary 9.1.4 allows us to define the *composition of type-theoretic transformations* as the composition of the underlying syntactic transformations.

Similarly to equality of instantiations we can define equality of type-theoretic transformations.

Definition 9.1.5 Type theoretic transformations $f : \mathcal{T} \to \mathcal{U}$ and $g : \mathcal{T} \to \mathcal{U}$ are *judgementally equal* if for every specific object rule $R = \Theta \Longrightarrow \mathcal{b}\boxed{e}$ of $\mathcal{T}$ the judgement

$$f_*\Theta; [] \vdash_{\mathcal{U}} (f_*\mathcal{b})\boxed{f_*e \equiv g_*e}$$

is derivable.

Similarly to the congruence rules, the definition is left-leaning. But the right-leaning version is admissible, once Proposition 9.1.6 is established.

If f and g are judgementally equal type-theoretic transformations, they act in a judgementally equal way on every judgement.

Proposition 9.1.6 Let $f : \mathcal{T} \to \mathcal{U}$ and $g : \mathcal{T} \to \mathcal{U}$ be judgementally equal type-theoretic transformations. Let $\Theta; \Gamma \vdash_{\mathcal{T}} \mathcal{B}\boxed{e}$ be a strongly derivable object judgement in $\mathcal{T}$. Then

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} (f_*\mathcal{B})\boxed{f_*e \equiv g_*e}$$

is derivable in $\mathcal{U}$.

The proof is mutually recursive with other lemmas about judgementally equal transformations: Lemma 9.1.9, Lemma 9.1.8 and Lemma 9.1.7. The recursion is well-founded by a lexicographic order on the length of the metacontext and the size of the derivation: either we use the lemmas on a shorter metacontext, or on a metacontext of the same size, but smaller derivations (premises).

Proof. By induction on the derivation of $\Theta; \Gamma \vdash_{\mathcal{T}} \mathcal{B}\boxed{e}$. By inversion we consider the cases of how the derivation ends.

Case TT-Var: Because type-theoretic transformations act trivially on variables, $f_*\mathsf{a} = g_*\mathsf{a} = \mathsf{a}$, so the desired equation $f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} \mathsf{a} \equiv \mathsf{a} : (f_*\Gamma)(\mathsf{a})$ can be derived using TT-Tm-Refl.

Case TT-Abstr: The derivation ends with

$$\frac{\Theta;\Gamma \vdash_{\mathcal{T}} A\ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma| \qquad \Theta;\Gamma, \mathsf{a}{:}A \vdash_{\mathcal{T}} (\mathcal{B}[\mathsf{a}/x])\boxed{e[\mathsf{a}/x]}}{\Theta;\Gamma \vdash_{\mathcal{T}} \{x{:}A\}\ \mathcal{B}\boxed{e}}$$

Using Theorem 9.1.3 on the first premise gives us a derivation of

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*A\ \mathsf{type}.$$

By induction hypothesis on the last premise we get a derivation of

$$f_*\Theta; f_*\Gamma, \mathsf{a}{:}f_*A \vdash_{\mathcal{U}} (f_*(\mathcal{B}[\mathsf{a}/x]))\boxed{f_*(e[\mathsf{a}/x]) \equiv g_*(e[\mathsf{a}/x])}$$

which is by Lemma 8.3.2 equal to

$$f_*\Theta; f_*\Gamma, \mathsf{a}{:}f_*A \vdash_{\mathcal{U}} ((f_*\mathcal{B})[\mathsf{a}/x]))\boxed{(f_*e)[\mathsf{a}/x] \equiv (g_*e)[\mathsf{a}/x]}$$

We can now conclude the desired derivation by TT-Abstr.

Case TT-Meta: The derivation ends with

$$\frac{\begin{array}{l}\Theta(\mathsf{M}_k) = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathcal{C} \\ \Theta;\Gamma \vdash_{\mathcal{T}} t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m \\ \Theta;\Gamma \vdash_{\mathcal{T}} \mathcal{C}[\vec{t}/\vec{x}]\end{array}}{\Theta;\Gamma \vdash_{\mathcal{T}} (\mathcal{C}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_k(\vec{t})}}$$

By induction hypothesis for the first m premises we get derivations of

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*t_j \equiv g_*t_j : f_*(A_j[\vec{t}_{(j)}/\vec{x}_{(j)}])$$

for $j = 1, \ldots, m$ which are by Lemma 8.3.2 equal to

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*t_j \equiv g_*t_j : (f_*A_j)[f_*\vec{t}_{(j)}/\vec{x}_{(j)}].$$

Using Theorem 9.1.3 and Lemma 8.3.2 on the last premise gives us a derivation of

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{T}} (f_*\mathcal{C})[f_*\vec{t}/\vec{x}]$$

and we can conclude by TT-Meta-Congr-Ec.

Case Specific object rule: Suppose $\Theta;\Gamma \vdash_{\mathcal{T}} \mathcal{B}\boxed{e}$ is derived by the specific object rule $\Xi \Longrightarrow \mathcal{C}'\boxed{e'}$ with a derivable (in $\mathcal{T}$) instantiation

$$I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$$

for $\Xi = [\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n]$. By assumption f and g are judgementally equal, so we have a derivation of

$$f_*\Xi; [] \vdash_{\mathcal{U}} (f_*\mathcal{C}')\boxed{f_*e' \equiv g_*e'}. \tag{9.1}$$

Since I is derivable we have derivations of

$$\Theta;\Gamma \vdash_{\mathcal{T}} (I_{(i)*}\mathcal{B}_i)\boxed{e_i} \tag{9.2}$$

for $i = 1, \dots, n$ and by induction hypothesis

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} (f_*(I_{(i)*}\mathscr{B}_i))\boxed{f_*e_i \equiv g_*e_i} \tag{9.3}$$

are also derivable for the object premises. Because f and g preserve derivability by Theorem 9.1.3 the judgements

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} (f_*(I_{(i)*}\mathscr{B}_i))\boxed{f_*e_i} \quad \text{and} \quad g_*\Theta; g_*\Gamma \vdash_{\mathcal{U}} (g_*(I_{(i)*}\mathscr{B}_i))\boxed{g_*e_i}$$

are also derivable for $i = 1, \dots, n$ and they are by Lemma 8.3.3 equal to

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} ((I_f)_{(i)*}(f_*\mathscr{B}_i))\boxed{f_*e_i} \quad \text{and} \quad g_*\Theta; g_*\Gamma \vdash_{\mathcal{U}} ((I_g)_{(i)*}(g_*\mathscr{B}_i))\boxed{g_*e_i} \tag{9.4}$$

for instantiations

$$I_f = \langle \mathsf{M}_1 \mapsto f_*e_1, \dots, \mathsf{M}_n \mapsto f_*e_n \rangle \quad \text{and} \quad I_g = \langle \mathsf{M}_1 \mapsto g_*e_1, \dots, \mathsf{M}_n \mapsto g_*e_n \rangle$$

of $f_*\Xi$ over $f_*\Theta; f_*\Gamma$ and of $g_*\Xi$ over $g_*\Theta; g_*\Gamma$ respectively. This also means that I_f and I_g are derivable instantiations. Using Lemma 9.1.8 and Lemma 9.1.9 on the judgements from the right side of (9.4) we get derivations of

We use Lemma 9.1.8 and Lemma 9.1.9 is on smaller derivations than the one we started with.

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} ((I'_g)_{(i)*}(g_*\mathscr{B}_i))\boxed{g_*e_i}$$

for $I'_g = \langle \mathsf{M}_1 \mapsto g_*e_1, \dots, \mathsf{M}_n \mapsto g_*e_n \rangle$ an instantiation of[2] $f_*\Xi$ over $f_*\Theta; f_*\Gamma$. This means I'_g is a derivable instantiation. Derivability of (9.3) gives us that I_f and I'_g are judgementally equal instantiations. Instantiating (9.1) with I_f gives us by Theorem 5.1.4 a derivable judgement

2: We can view I'_g as an instantiation of $f_*\Xi$ because $\mathsf{ar}(f_*\mathscr{B}_i) = \mathsf{ar}(g_*\mathscr{B}_i)$.

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} ((I_f)_*(f_*\mathscr{b}'))\boxed{(I_f)_*(f_*e') \equiv (I_f)_*(g_*e')}$$

Which is by Lemma 8.3.3 equal to

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} (f_*\mathscr{B})\boxed{f_*e \equiv (I_f)_*(g_*e')}. \tag{9.5}$$

By Theorem 5.1.6 on (9.1) we get a derivation of

$$f_*\Xi; [] \vdash_{\mathcal{U}} (f_*\mathscr{b}')\boxed{g_*e'}$$

on which we use Theorem 5.1.5 with judgementally equal instantiations I_f and I'_g to obtain a derivation of

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} ((I_f)_*(f_*\mathscr{b}'))\boxed{(I_f)_*(g_*e') \equiv (I'_g)_*(g_*e')}$$

that is by Lemma 8.3.3 equal to

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} (f_*\mathscr{B})\boxed{(I_f)_*(g_*e') \equiv g_*e}. \tag{9.6}$$

The proof conludes with the appropriate transitivity rule depending on the form of the object boundary $\mathscr{B}$.

We conclude the proof by TT-Ty-Tran or TT-Tm-Tran on (9.5) and (9.6). □

The following lemmas see to the fact that every boundary, judgement, variable context or metacontext which has been acted on by a transformation, can be replaced by the action of a judgementally equal transformation.

Lemma 9.1.7 Let $f : \mathcal{T} \to \mathcal{U}$ and $g : \mathcal{T} \to \mathcal{U}$ be judgementally equal type-theoretic transformations and $\Theta; \Gamma \vdash_{\mathcal{T}} \mathcal{B}$ be a strongly derivable boundary in $\mathcal{T}$. If

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} (f_*\mathcal{B})\boxed{e}$$

is derivable in $\mathcal{U}$, then

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} (g_*\mathcal{B})\boxed{e}$$

is also derivable.

The proof follows closely the analogous statement for judgementally equal instantiations Lemma 5.3.5.

Proof. By structural induction on the derivation of $\Theta; \Gamma \vdash_{\mathcal{T}} \mathcal{B}$.

Case TT-Bdry-Ty: Trivial, because $f_*(\square\ \mathsf{type}) = g_*(\square\ \mathsf{type}) = \square\ \mathsf{type}$.

Case TT-Bdry-Tm: If the derivation ends with

$$\frac{\Theta; \Gamma \vdash_{\mathcal{T}} A\ \mathsf{type}}{\Theta; \Gamma \vdash_{\mathcal{T}} \square : A}$$

We use Proposition 9.1.6 on smaller derivation than the one we started with.

then by Proposition 9.1.6 applied to the premise we have that

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*A \equiv g_*A.$$

We can thus convert

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} e : (f_*A)$$

to

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} e : (g_*A).$$

Case TT-Bdry-EqTm: If the derivation ends with

$$\frac{\Theta; \Gamma \vdash_{\mathcal{T}} A\ \mathsf{type} \qquad \Theta; \Gamma \vdash_{\mathcal{T}} s : A \qquad \Theta; \Gamma \vdash_{\mathcal{T}} t : A}{\Theta; \Gamma \vdash_{\mathcal{T}} s \equiv t : A\ \mathsf{by}\ \square}$$

We again use Proposition 9.1.6 on smaller derivations than the one we started with.

then Proposition 9.1.6 applied to the premises gives us derivations of

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*A \equiv g_*A \tag{9.7}$$
$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*s \equiv g_*s : f_*A \tag{9.8}$$
$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*t \equiv g_*t : f_*A \tag{9.9}$$

By TT-Tm-Tran we can combine the derivable equation

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*s \equiv f_*t : f_*A$$

with (9.8) and (9.9) to get a derivable judgement

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} g_*s \equiv g_*t : f_*A$$

which we can convert using TT-Conv-Eq on (9.7) to get the desired derivation.

Case TT-Bdry-EqTy: Similar to the case TT-Bdry-EqTm.

Case TT-Bdry-Abstr: Suppose $e = \{x\}e'$ and the derivation ends with

$$\frac{\Theta; \Gamma \vdash_{\mathcal{T}} A \ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma| \qquad \Theta; \Gamma, \mathsf{a}{:}A \vdash_{\mathcal{T}} \mathscr{B}'[\mathsf{a}/x]}{\Theta; \Gamma \vdash_{\mathcal{T}} \{x{:}A\}\ \mathscr{B}'}$$

where we may assume $\mathsf{a} \notin |\Gamma|$ without loss of generality. Proposition 9.1.6 applied to the first premise derives

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*A \equiv g_*A. \tag{9.10}$$

By inverting the assumption $f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} \{x{:}f_*A\}\ (f_*\mathscr{B}')\boxed{e}$, and possibly renaming a free variable to a, we obtain

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} f_*A \ \mathsf{type} \quad \text{and} \quad f_*\Theta; f_*\Gamma, \mathsf{a}{:}f_*A \vdash_{\mathcal{U}} ((f_*\mathscr{B}')\boxed{e})[\mathsf{a}/x].$$

Then the induction hypothesis for the second premise yields

$$f_*\Theta; f_*\Gamma, \mathsf{a}{:}f_*A \vdash_{\mathcal{U}} ((g_*\mathscr{B}')\boxed{e})[\mathsf{a}/x],$$

which we may abstract to $f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} \{x{:}f_*A\}\ (g_*\mathscr{B}')\boxed{e}$ and apply TT-Conv-Abstr to convert it to the desired judgement

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} \{x{:}g_*A\}\ (g_*\mathscr{B}')\boxed{e}.$$

The premise $f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} g_*A \ \mathsf{type}$ is derived by Theorem 5.1.6 from (9.10). □

Lemma 9.1.8 Let $f : \mathcal{T} \to \mathcal{U}$ and $g : \mathcal{T} \to \mathcal{U}$ be judgementally equal type-theoretic transformations and $\Theta; \Gamma$ a derivable context in $\mathcal{T}$. If $f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} \mathcal{J}$ is derivable, then $f_*\Theta; g_*\Gamma \vdash_{\mathcal{U}} \mathcal{J}$ is also derivable.

Proof. The idea is to abstract away the context and use TT-Conv-Abstr to replace the types with judgementally equal ones. By induction on the derivation of the variable context Γ.

Case VCtx-Empty: Trivial, because $f_*[] = g_*[] = []$.

Case VCtx-Extend: The derivation ends with

$$\frac{\Theta \vdash_{\mathcal{T}} \Gamma' \ \mathsf{vctx} \qquad \Theta, \Gamma' \vdash_{\mathcal{T}} A \ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma'|}{\Theta \vdash_{\mathcal{T}} \langle \Gamma', \mathsf{a}{:}A \rangle \ \mathsf{vctx}}$$

Theorem 9.1.3 on the second premise leads to derivability of

$$f_*\Theta, f_*\Gamma' \vdash_{\mathcal{U}} f_*A$$

and using TT-Abstr with $f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} \mathcal{J}$ gives us a derivable judgement $f_*\Theta, f_*\Gamma' \vdash_{\mathcal{U}} \{x{:}f_*A\}\ \mathcal{J}$. By induction hypothesis

$$f_*\Theta, g_*\Gamma' \vdash_{\mathcal{U}} \{x{:}f_*A\}\ \mathcal{J}$$

is derivable. Proposition 9.1.6 on the second premise gives us a derivation of $f_*\Theta, f_*\Gamma' \vdash_{\mathcal{U}} f_*A \equiv g_*A$ for which the induction hypothesis

We use Proposition 9.1.6 on smaller derivation than the one we started with.

again gives us $f_*\Theta, g_*\Gamma' \vdash_{\mathcal{U}} f_*A \equiv g_*A$. We use TT-Conv-Abstr to obtain a derivation of

$$f_*\Theta, g_*\Gamma' \vdash_{\mathcal{U}} \{x{:}g_*A\}\ \mathcal{J}$$

and conclude by un-abstracting using inversion on the last judgement.

□

While the previous two lemmas were similar in the case of judgementally equal instantiations, the following lemma is specific to type-theoretic transformations, as it relates "judgementally equal metacontexts": metacontexts acted on by judgementally equal transformations.

Lemma 9.1.9 Let $f : \mathcal{T} \to \mathcal{U}$ and $g : \mathcal{T} \to \mathcal{U}$ be judgementally equal type-theoretic transformations and Θ a derivable metacontext in $\mathcal{T}$.

1. If $f_*\Theta; \vdash_{\mathcal{U}} \Gamma\ \mathsf{vctx}$ is derivable, then $g_*\Theta \vdash_{\mathcal{U}} \Gamma\ \mathsf{vctx}$ is derivable.
2. If $f_*\Theta; \Gamma \vdash_{\mathcal{U}} \mathcal{B}$ is strongly derivable, then $g_*\Theta; \Gamma \vdash_{\mathcal{U}} \mathcal{B}$ is strongly derivable.
3. If $f_*\Theta; \Gamma \vdash_{\mathcal{U}} \mathcal{J}$ is strongly derivable, then $g_*\Theta; \Gamma \vdash_{\mathcal{U}} \mathcal{J}$ is strongly derivable.

The proof of Lemma 9.1.9 is very technical, so let us describe the main idea: we get from the metacontext $f_*\Theta$ to $g_*\Theta$ by the instantiation that maps every metavariable to its generic application. Since transformations act trivially on metavariables, this instantiation is well-formed. To prove it is derivable we recursively use the fact that judgementally equal instantiations lead to judgemental equality.

Proof. All parts of the proof are mutually recursive.

Part (1): We proceed by induction on the derivation of Γ.

Case VCtx-Empty: Trivial, because the empty variable context is derivable in every derivable metacontext.

Case VCtx-Extend: The derivation ends with

$$\frac{f_*\Theta \vdash_{\mathcal{U}} \Gamma'\ \mathsf{vctx} \qquad f_*\Theta, \Gamma' \vdash_{\mathcal{U}} A\ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma'|}{f_*\Theta \vdash_{\mathcal{U}} \langle \Gamma', \mathsf{a}{:}A \rangle\ \mathsf{vctx}}$$

By induction hypothesis on the first premise we get a derivation of $g_*\Theta \vdash_{\mathcal{U}} \Gamma'\ \mathsf{vctx}$ and induction hypothesis on the second premise with Part (3) yields derivability of $g_*\Theta, \Gamma' \vdash_{\mathcal{U}} A\ \mathsf{type}$ so we can conclude with VCtx-Extend.

Part (2): We proceed by induction of length of Θ.

Case $\Theta = []$: Trivial, because $f_*[] = g_*[] = []$.

Case $\Theta = [\mathsf{M}_1{:}\mathscr{B}_1, \dots, \mathsf{M}_{n+1}{:}\mathscr{B}_{n+1}]$: Note that $\vdash_{\mathscr{U}} g_*\Theta$ mctx is derivable by Theorem 9.1.3. We can obtain $g_*\Theta; \Gamma \vdash_{\mathscr{U}} \mathscr{B}$ from $f_*\Theta; \Gamma \vdash_{\mathscr{U}} \mathscr{B}$ by acting with the instantiation

$$I = \langle \mathsf{M}_1 \mapsto \widehat{\mathsf{M}_1}, \dots, \mathsf{M}_{n+1} \mapsto \widehat{\mathsf{M}_{n+1}} \rangle$$

of the metacontext $f_*\Theta$ over the context $g_*\Theta; \Gamma$ which is derivable by Part (1). Once we show that I is a derivable instantiation we can conclude the proof by Theorem 5.1.4. To prove I is derivable we need to show that for $i = 1, \dots, n+1$

$$g_*\Theta; \Gamma \vdash_{\mathscr{U}} (I_{(i)*}(f_*\mathscr{B}_i))\boxed{\widehat{\mathsf{M}_i}}$$

which is equal to

$$g_*\Theta; \Gamma \vdash_{\mathscr{U}} (f_*\mathscr{B}_i)\boxed{\widehat{\mathsf{M}_i}}$$

is derivable. Since f preserves derivability and Θ is a derivable metacontext, for every $i = 1, \dots, n+1$

$$f_*([\mathsf{M}_1{:}\mathscr{B}_1, \dots, \mathsf{M}_i{:}\mathscr{B}_i]); [] \vdash_{\mathscr{U}} (f_*\mathscr{B}_i)\boxed{\widehat{\mathsf{M}_i}}$$

is strongly derivable. By induction hypothesis on Part (3) for $i = 1, \dots, n$ we derive

$$g_*([\mathsf{M}_1{:}\mathscr{B}_1, \dots, \mathsf{M}_i{:}\mathscr{B}_i]); [] \vdash_{\mathscr{U}} (f_*\mathscr{B}_i)\boxed{\widehat{\mathsf{M}_i}}$$

which we can weaken to

$$g_*\Theta; \Gamma \vdash_{\mathscr{U}} (f_*\mathscr{B}_i)\boxed{\widehat{\mathsf{M}_i}}.$$

For $i = n+1$ by inversion on derivation of $\vdash_{\mathscr{T}} \Theta$ mctx we obtain a derivation of

$$[\mathsf{M}_1{:}\mathscr{B}_1, \dots, \mathsf{M}_n{:}\mathscr{B}_n]; [] \vdash_{\mathscr{T}} \mathscr{B}_{n+1}. \tag{9.11}$$

Let $\mathscr{B}_{n+1} = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\, \mathscr{C}$. By inversion we obtain derivations of

$$[\mathsf{M}_1{:}\mathscr{B}_1, \dots, \mathsf{M}_n{:}\mathscr{B}_n]; [\mathsf{a}_1{:}A_1, \dots, \mathsf{a}_{j-1}{:}A_{j-1}[\vec{\mathsf{a}}_{(j-1)}/\vec{x}_{(j-1)}]] \vdash_{\mathscr{T}} A_j[\vec{\mathsf{a}}_{(j)}/\vec{x}_{(j)}]\ \mathsf{type}$$

for $j = 1, \dots, m$. We can apply Proposition 9.1.6 followed by the induction hypothesis on the shorter metacontext to obtain derivations of

We use Proposition 9.1.6 on a shorter metacontext.

$$g_*([\mathsf{M}_1{:}\mathscr{B}_1, \dots, \mathsf{M}_n{:}\mathscr{B}_n]); [\mathsf{a}_1{:}f_*A_1, \dots, \mathsf{a}_{j-1}{:}f_*A_{j-1}[\vec{\mathsf{a}}_{(j-1)}/\vec{x}_{(j-1)}]]$$
$$\vdash_{\mathscr{U}} f_*A_j[\vec{\mathsf{a}}_{(j)}/\vec{x}_{(j)}] \equiv g_*A_j[\vec{\mathsf{a}}_{(j)}/\vec{x}_{(j)}]$$

which we can weaken to

$$g_*\Theta; [\mathsf{a}_1{:}f_*A_1, \dots, \mathsf{a}_{j-1}{:}f_*A_{j-1}[\vec{\mathsf{a}}_{(j-1)}/\vec{x}_{(j-1)}]]$$
$$\vdash_{\mathscr{U}} f_*A_j[\vec{\mathsf{a}}_{(j)}/\vec{x}_{(j)}] \equiv g_*A_j[\vec{\mathsf{a}}_{(j)}/\vec{x}_{(j)}].$$

Using TT-Meta we can derive

$$g_*\Theta; [] \vdash_{\mathscr{U}} \{\vec{x}{:}g_*\vec{A}\}(g_*\mathscr{C})\boxed{\widehat{\mathsf{M}_{n+1}}}. \tag{9.12}$$

Iteratively using TT-Abstr and TT-Conv-Abstr on (9.12) we can now

derive

$$g_*\Theta; [] \vdash_{\mathcal{U}} \{\vec{x}{:}f_*\vec{A}\}(g_*\mathscr{b})\boxed{\widehat{\mathsf{M}}_{n+1}}.$$

By Proposition 4.3.7 the judgement

$$g_*\Theta; [\vec{\mathsf{a}}{:}f_*\vec{A}[\vec{\mathsf{a}}/\vec{x}]] \vdash_{\mathcal{U}} (g_*\mathscr{b}[\vec{\mathsf{a}}/\vec{x}])\boxed{\mathsf{M}_{n+1}(\vec{\mathsf{a}})} \tag{9.13}$$

is derivable. We now consider the cases for the boundary thesis $\mathscr{b}$.

We use vector notation $\vec{A}[\vec{\mathsf{a}}/\vec{x}]$ to mean we substitute simultaneously all the x_k with a_k for $k = 1, \ldots, m-1$ in all A_j for $j = 1, \ldots, m$.

Case $\mathscr{b} = (\square\ \mathsf{type})$: Since $f_*(\mathsf{M}_{n+1}(\vec{\mathsf{a}})\ \mathsf{type}) = g_*(\mathsf{M}_{n+1}(\vec{\mathsf{a}})\ \mathsf{type}) = \mathsf{M}_{n+1}(\vec{\mathsf{a}})\ \mathsf{type}$ we can obtain the desired derivation by Proposition 4.3.7.

Case $\mathscr{b} = (\square : B)$: Proposition 9.1.6 gives us

$$f_*([\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]); [\vec{\mathsf{a}}{:}f_*\vec{A}[\vec{\mathsf{a}}/\vec{x}]] \vdash_{\mathcal{U}} f_*B[\vec{\mathsf{a}}/\vec{x}] \equiv g_*B[\vec{\mathsf{a}}/\vec{x}]$$

which by induction hypothesis and weakening yields

$$g_*\Theta; [\vec{\mathsf{a}}{:}f_*\vec{A}[\vec{\mathsf{a}}/\vec{x}]] \vdash_{\mathcal{U}} f_*B[\vec{\mathsf{a}}/\vec{x}] \equiv g_*B[\vec{\mathsf{a}}/\vec{x}]$$

We can convert (9.13) along the symmetric version of this equality and conclude by Proposition 4.3.7.

Case $\mathscr{b} = (B \equiv C\ \mathsf{by}\ \square)$: By inversion

$$[\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]; [] \vdash_{\mathcal{T}} B\ \mathsf{type} \quad \text{and} \quad [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]; [] \vdash_{\mathcal{T}} C\ \mathsf{type}$$

are also derivable. Proposition 9.1.6 gives us derivations of

$$f_*([\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]); [\vec{\mathsf{a}}{:}f_*\vec{A}[\vec{\mathsf{a}}/\vec{x}]] \vdash_{\mathcal{U}} f_*B[\vec{\mathsf{a}}/\vec{x}] \equiv g_*B[\vec{\mathsf{a}}/\vec{x}]$$
$$f_*([\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]); [\vec{\mathsf{a}}{:}f_*\vec{A}[\vec{\mathsf{a}}/\vec{x}]] \vdash_{\mathcal{U}} f_*C[\vec{\mathsf{a}}/\vec{x}] \equiv g_*C[\vec{\mathsf{a}}/\vec{x}]$$

which by induction hypothesis and weakening yield

$$g_*\Theta; [\vec{\mathsf{a}}{:}f_*\vec{A}[\vec{\mathsf{a}}/\vec{x}]] \vdash_{\mathcal{U}} f_*B[\vec{\mathsf{a}}/\vec{x}] \equiv g_*B[\vec{\mathsf{a}}/\vec{x}]$$
$$g_*\Theta; [\vec{\mathsf{a}}{:}f_*\vec{A}[\vec{\mathsf{a}}/\vec{x}]] \vdash_{\mathcal{U}} f_*C[\vec{\mathsf{a}}/\vec{x}] \equiv g_*C[\vec{\mathsf{a}}/\vec{x}].$$

With the above equations and the use of TT-Tm-Sym and TT-Ty-Tran on (9.13) we obtain the desired judgement.

Case $\mathscr{b} = (s \equiv t : T\ \mathsf{by}\ \square)$: Proceed similarly to the case for type equality boundary.

Part (3): The proof proceeds similarly to Part (2). □

Corollary 9.1.10 Let $f : \mathcal{T} \to \mathcal{U}$ and $g : \mathcal{T} \to \mathcal{U}$ be judgementally equal type-theoretic transformations and $\Theta; \Gamma \vdash_{\mathcal{T}} \mathscr{B}$ be a strongly derivable boundary in $\mathcal{T}$. If

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{U}} (f_*\mathscr{B})\boxed{e}$$

is derivable in $\mathcal{U}$, then

$$g_*\Theta; g_*\Gamma \vdash_{\mathcal{U}} (g_*\mathscr{B})\boxed{e}$$

is also derivable.

Proof. Apply Lemma 9.1.7, followed by Lemma 9.1.8 and Lemma 9.1.9. □

9.2. The category of type theories

Having defined type-theoretic transformations and their composition, we can now organize type theories in a category.

Definition 9.2.1 The *category of type theories* has objects (raw) type theories and morphisms transformations of type theories.

Since composition of type theories is associative by Corollary 8.3.5 and the identity type-theoretic transformation behaves appropriately, this is a well-defined category.

We can now inspect the properties of our newly defined category. We state the two obvious ones and leave the rest for future work.

Proposition 9.2.2 The initial object in the category of type theories is the empty type theory.

The *empty type theory* is the type theory with the empty signature and no specific rules.

Proof. Let $\mathcal{T}$ be a type theory. The unique type-theoretic transformation from the empty type theory to $\mathcal{T}$ is the empty syntactic transformation from the empty signature [] to the expressions of $\mathcal{T}$. It is trivially a type-theoretic transformation, because there are no specific rules to map to derivations. □

Note that the empty type theory still has derivable judgements, but they always happen in a non-empty context. For example

$$[\mathsf{A}{:}(\square\ \mathsf{type}),\ \mathsf{a}{:}(\square : A),\ \mathsf{B}{:}(\{x{:}A\}\square\ \mathsf{type})];\ [] \vdash \mathsf{B(a)\ type}$$

is a derivable judgement of the empty type theory.

Proposition 9.2.3 The category of type theories has coproducts.

Proof. The idea is that the coproduct of type theories is obtained by a disjoint union of the signatures and a disjoint union of rules. The injection maps are the obvious injections of signatures and the (chosen) derivations of rules are the injections of the rules.

To dissect this idea in our formalism, let $\mathcal{T}$ and $\mathcal{U}$ be type theories. Let

$$\Sigma_{\mathcal{T}} = [T_1{:}\alpha_1, \ldots, T_n{:}\alpha_n] \qquad \text{and} \qquad \Sigma_{\mathcal{U}} = [U_1{:}\beta_1, \ldots, U_m{:}\beta_m]$$

be the signatures[3] of the given theories. We construct the signature of the coproduct $\mathcal{T} + \mathcal{U}$

$$\Sigma_{\mathcal{T}+\mathcal{U}} = [T_1^{\mathcal{T}}{:}\alpha_1, \ldots, T_n^{\mathcal{T}}{:}\alpha_n, U_1^{\mathcal{U}}{:}\beta_1, \ldots, U_m^{\mathcal{U}}{:}\beta_m]$$

3: The signatures do not necessarily contain different symbols. The indexed letters T and U in the signatures are meta-level variables for the symbols of the two type theories.

and injections the syntactic transformations induced by the symbol renamings

$$\begin{aligned} i_1 \colon \Sigma_{\mathcal{T}} &\to \Sigma_{\mathcal{T}+\mathcal{U}} \\ T_j &\mapsto T_j^{\mathcal{T}} \text{ for } j = 1, \dots, n \end{aligned}$$

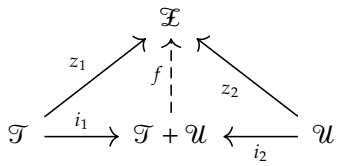
and

$$\begin{aligned} i_1 \colon \Sigma_{\mathcal{U}} &\to \Sigma_{\mathcal{T}+\mathcal{U}} \\ U_k &\mapsto U_k^{\mathcal{U}} \text{ for } k = 1, \dots, m. \end{aligned}$$

The specific rules of $\mathcal{T} + \mathcal{U}$ are

- $(i_1)_* R$ for R a specific rule in $\mathcal{T}$,
- $(i_2)_* R$ for R a specific rule in $\mathcal{U}$.

With this definition i_1 and i_2 are trivially type-theoretic transformations. We need to show that $\mathcal{T} + \mathcal{U}$ is indeed a coproduct in the category of type theories. Let $\mathcal{E}$ be a type theory with type-theoretic transformations z_1 and z_2 as in the following diagram.



For the diagram to commute, the unique type-theoretic transformation f maps symbols $T_j^{\mathcal{T}}$ to $z_1(T_j)$ for $j = 1, \dots, n$ and symbols $U_j^{\mathcal{U}}$ to $z_2(U_j)$ for $j = 1, \dots, m$. It is now easy to see that f is a type-theoretic transformation if it uses the derivations from transformations z for appropriate specific rules. □

Using the construction for coproducts we can build raw type theories by combining independent parts of the type theory (like having dependent products and dependent sums). However in practice we are usually interested in finitary or even standard type theories, so we would need to appropriately combine the well-ordering of the rules as well.

9.3. Examples of type-theoretic transformations

We show the scope and limitations of our definition of type-theoretic transformations by exhibiting some interesting examples and non-examples.

We start by looking at examples. Besides the usual symbol renamings, there are some concrete and some more theoretically flavored transformations that adhere the this structure.

Example 9.3.1 One of the most well-known transformations is the *propositions as types*, also known as the Curry-Howard correspondence [48, 49, 74, 149] that translates mathematical proofs of first-order logic to terms of type theory and thus gives them computational content. We exhibit the transformation from first-order logic (FOL) to the relevant fragment of the Martin-Löf type theory (MLTT) [97–100] with one universe of (small) types.

[149]: Wadler (2015), "Propositions as Types"
[48]: Curry (1934), "Functionality in Combinatory Logic"
[49]: Curry et al. (1958), *Combinatory logic. Vol. I*
[74]: Howard (1980), "The Formulae-as-Types Notion of Construction"

The full formulation of FOL and MLTT, as well as the type-theoretic transformation is in the Appendix Chapter A. The idea is to map the type o of propositions in FOL to the universe U of MLTT and the terms that represent propositions $\mathsf{p} : \mathsf{o}$ to the codes for types $\mathsf{a} : \mathsf{U}$. We also map the type i of individuals in FOL to the base type in MLTT.

The rest of the connectives are mapped as usual, for example the conjunctions conj are mapped to the simple products $\sigma(\mathsf{p}, \{x\}\mathsf{q})$ and universal quantifiers to the dependent products.

In Chapter A we also write justification that this is indeed a type-theoretic transformation, i.e. that derivability is preserved.

[100]: Martin-Löf (1998), "An intuitionistic theory of types"
[97]: Martin-Löf (1975), "An intuitionistic theory of types: predicative part"
[98]: Martin-Löf (1982), "Constructive mathematics and computer programming"
[99]: Martin-Löf (1984), *Intuitionistic type theory*

Similarly to propositions as types we could also use type-theoretic transformations to formulate propositions as bracketed or squash types [16, 44, 117, 142], which hide the computational content. The bracket types as proposed in [16] are an example of a type theory where an introduction rule for a term judgement has an equational premise. The paper also contains an example of a formula that is not provable in intuitionistic first-order logic, but its translation is derivable in the dependent type theory. We can generalise this construction to propositions as n-types translation (for n-truncations as described in the HoTT book [142]). However, for the general modalities the same construction might not be available, as they are not necessarily closed under Σ-types, which we need to make this a valid transformation.

[16]: Awodey et al. (2004), "Propositions as [Types]"
[117]: Pfenning (2001), "Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory"
[142]: The Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*
[44]: Constable et al. (1986), *Implementing mathematics with the Nuprl proof development system*

[142]: The Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

While in the above examples we had concrete transformations between instances of type theories, there are also some more general examples.

Example 9.3.2 Often a type theory is extended by adding a symbol to the signature and possibly some rules that involve the newly added symbol. We can always inject the theory into the extended one by mapping symbols to their generic applications and the rules to the generic derivations. This injection is trivially a type-theoretic transformation.

Often we don't want to just merely add new symbols, but we want to make a *conservative* extension, a super-theory, which proves no new theorems about the language of the original theory, but is in some way more convenient for proving theorems. We can make the notion of conservativity precise for type theories with the following definition.

Definition 9.3.3 A transformation of type theories $f : \mathcal{T} \to \mathcal{U}$ is *conservative* if for every strongly derivable rule-boundary $\Theta \Longrightarrow \mathcal{b}$ in $\mathcal{T}$ it holds that if $f_*\Theta \Longrightarrow f_*\mathcal{b}\boxed{e}$ is derivable in $\mathcal{U}$ then we have e' such that $\Theta \Longrightarrow \mathcal{b}\boxed{e'}$ is derivable in $\mathcal{T}$.

In the Definition 9.3.3 above we can think of the theory $\mathcal{T}$ as a conservative extension of $\mathcal{U}$, so we have a transformation from the extension to the original theory. To make sure the theory $\mathcal{T}$ indeed captures all the relevant derivability structure of $\mathcal{U}$ it should also be a *cover* in the following sense.

Definition 9.3.4 A type-theoretic transformation $f : \mathcal{T} \to \mathcal{U}$ is a *cover*, if it is surjective on strongly derivable judgements, i.e. for every strongly derivable judgement $\Theta;\Gamma \vdash_{\mathcal{U}} \mathcal{J}$ in $\mathcal{U}$ there is a judgement $\Theta';\Gamma' \vdash_{\mathcal{T}} \mathcal{J}'$ such that

$$f_*(\Theta';\Gamma' \vdash_{\mathcal{T}} \mathcal{J}') = \Theta;\Gamma \vdash_{\mathcal{U}} \mathcal{J}.$$

Example 9.3.5 A typical example of a conservative extension is the so called *definitional extension*. Let $\mathcal{T}$ be a type theory and let S be a symbol, that is not is the signature of $\mathcal{T}$. We construct the definitional extension $\mathcal{U}$ of $\mathcal{T}$ by adding $\mathsf{S}{:}(\mathsf{c}, \vartheta)$ to the signature and adding two specific rules

$$\frac{\vdash \mathcal{B}_i\boxed{\mathsf{M}_i} \quad \text{for } i = 1,\dots,n}{\vdash \mathcal{b}\boxed{\mathsf{S}(\widehat{\mathsf{M}_1},\dots,\widehat{\mathsf{M}_n})}} \qquad \frac{\vdash \mathcal{B}_i\boxed{\mathsf{M}_i} \quad \text{for } i = 1,\dots,n}{\vdash \mathcal{b}\boxed{\mathsf{S}(\widehat{\mathsf{M}_1},\dots,\widehat{\mathsf{M}_n}) \equiv e}}$$

for $\mathsf{M}_1,\dots,\mathsf{M}_n$ metavariables from the metavariable shape ϑ, boundaries $\mathcal{B}_1$, ..., $\mathcal{B}_n$ of appropriate arities according to ϑ and some valid expression e of the syntactic class c, such that we have a derivation $\mathcal{D}$ of $[\mathsf{M}_1{:}\mathcal{B}_1,\dots,\mathsf{M}_n{:}\mathcal{B}_n];[] \vdash \mathcal{b}\boxed{e}$ in $\mathcal{T}$. The conservative cover $f : \mathcal{U} \to \mathcal{T}$ is defined by

$$\begin{aligned} \mathsf{S} &\mapsto e \\ \mathsf{T} &\mapsto \mathsf{T}(\widehat{\mathsf{M}_1},\dots,\widehat{\mathsf{M}_m}) \end{aligned}$$

for T symbols from the signature of $\mathcal{T}$ (and the signature of $\mathcal{U}$) and metavariables $\mathsf{M}_1,\dots,\mathsf{M}_m$ from the metavariable shape of the symbol T. The derivations for the specific rules that are common to $\mathcal{T}$ and $\mathcal{U}$ are just generic derivations that apply that same rule. The derivation pertaining the new symbol rule for S is $\mathcal{D}$ and we map the new equality to an application of TT-TM-REFL on e. With this data f is type-theoretic transformation.

Furthermore, if we think of f as a transformation from $\mathcal{U}$ back to $\mathcal{U}$ it is judgementally equal to the identity transformation $\mathrm{id}_{\mathcal{U}}$. Indeed, f and $\mathrm{id}_{\mathcal{U}}$ act the same way on all object specific rules except for the symbol rule for S, where the defining equation for S makes sure we get the necessary judgemental equality.

It is easy to see that f is a cover, because it hits all judements of $\mathcal{T}$,

not just the strongly derivable ones. To prove it is conservative, let $\Theta \Longrightarrow \mathcal{b}'$ be a strongly derivable rule-boundary in $\mathcal{U}$ such that $f_*\Theta \Longrightarrow f_*\mathcal{b}'\boxed{e'}$ is derivable in $\mathcal{T}$. Since $\mathcal{T}$ is embedded in $\mathcal{U}$, the judgement $f_*\Theta \Longrightarrow f_*\mathcal{b}'\boxed{e'}$ is also derivable in $\mathcal{U}$. Since f is judgementally equal to $\mathrm{id}_{\mathcal{U}}$ by Corollary 9.1.10 we have that $\Theta;[] \vdash \mathcal{b}'\boxed{e'}$ is also derivable, which concludes the argument.

We have seen an example of a definitional extension in the Example 9.3.7, where we added a symbol ¬ for negation and its definitional equation to the FOL.

Example 9.3.6 Another example of a conservative cover is the *retrogression transformation* from the elaborated version $\mathcal{S}$ of a finitary type theory $\mathcal{T}$. The full definition of the elaboration and the retrogression transformation are in Chapter 10.

While the propositions as types transformation relates FOL and MLTT, the usual translations of classical logic to an intuitionistic setting cannot be expressed as type-theoretic transformations as defined in Definition 9.1.2.

Example 9.3.7 One of the most known translations between logics is the *double negation translation* that was developed by Kolmogorov [83], Gödel [62], Gentzen [56, 57, 138], Kuroda [86] and Krivine [84] (their versions differ slightly) taking each classical proposition into its double negation and thereby translating a classically valid formula into an intuitionistically valid one.

[83]: Kolmogorov (1925), "On the principles of excluded middle (Russian)"
[62]: Gödel (1933), "Zur intuitionistischen Arithmetik und Zahlentheorie"

[57]: Gentzen (1974), "Über das Verhältnis zwischen intuitionistischer und klassischer Arithmetik"
[138]: Szabo (1971), "The Collected Papers of Gerhard Gentzen"
[56]: Gentzen (1936), "Die Widerspruchsfreiheit der reinen Zahlentheorie"

It is not immediately obvious that the translation cannot be expresseed in our setting, since the translation is in essence adding the double negation to disjunctions, existential quantifiers and atomic formulas. In our setting, we can extend the definition of FOL from the Appendix Chapter A by a symbol ¬ for negation and its definitional rules:

$$\frac{\vdash \mathsf{p} : \mathsf{o}}{\vdash \neg(\mathsf{p}) : \mathsf{o}} \qquad \text{NOT-DEF}\ \frac{\vdash \mathsf{p} : \mathsf{o}}{\vdash \neg(\mathsf{p}) \equiv \mathsf{imp}(\mathsf{p}, \mathsf{false})}$$

[86]: Kuroda (1951), "Intuitionistische Untersuchungen der formalistischen Logik"

[84]: Krivine (1990), "Opérateurs de mise en mémoire et traduction de Gödel"

We could start the syntactic transformation as follows.

Symbol in FOL	Metavariable shape	Expression
o	[]	o
true	[p:(Tm, 0)]	true(p)
i	[]	i
conj	[p:(Tm, 0), q:(Tm, 0)]	conj(p, q)
disj	[p:(Tm, 0), q:(Tm, 0)]	¬(¬(disj(p, q)))
exists	[P:(Tm, 1)]	¬(¬(exists({x}P(x))))
Atomic	[M_1:(Tm, 0), . . . , M_n:(Tm, 0)]	¬(¬(Atomic($\vec{\mathsf{M}}$)))

The notation $\vec{\mathsf{M}}$ is just short for $\mathsf{M}_1, \ldots, \mathsf{M}_n$.

The last row represents how we map atomic formulas (predicates)

that are in our setting added as new symbols, so we have to double-negate them as well when extending the transformation. We can even transform the introduction rules conjI, existsI, disjI1 etc.

The issue lies in the elimination rules disjE and existsE. In the classical proofs that the double-negation translation translates classical FOL to intuitionistic one, the derivations using elimination of disjunction are translated by identifying all the specific uses of the law of excluded middle (LEM) and translating those instances separately. The translation is thus admissible and not derivable as our notion of type-theoretic transformation requires: the symbols and specific rules need to be mapped in a generic way, not depending on their specific instantiations.

Similar to the double negation translation is the *A-translation* [55], which fails to be a type-theoretic transformation for a similar reason.

[55]: Friedman (1978), "Classically and intuitionistically provably recursive functions"

The type-theoretic transformation as a proposed notion of a transformation between type theories preserves quite a bit of the syntactic structure: it preserves syntactic classes of expressions, arities of metavariables and judgement forms. Such restrictions inevitably exclude several useful translations between type theories.

One such translation is the *elimination of equality reflection* from intentional type theory, which was implemented by Winterhalter, Sozeau and Tabareau [152]. They translate derivations of ETT into proof terms of ITT (with uniqueness of identity proofs and function extensionality), which was first done on a semantic level (categorically) by Hofmann in 1995 [70, 72], later syntactically by Oury [112] and has now been implemented in Coq. The translation is also conservative. Because it works on derivations and does not preserve judgemental equality, it is not a type-theoretic transformation in the sense of Definition 9.1.2. A transformation of a similar nature (mapping derivations to judgements) is the *elaboration map* from Chapter 10, which is not a type-theoretic transformation for the same reason.

[152]: Winterhalter et al. (2019), "Eliminating Reflection from Type Theory"

[70]: Hofmann (1996), "Conservativity of Equality Reflection over Intensional Type Theory"
[72]: Hofmann (1997), *Extensional Constructs in Intensional Type Theory*

[112]: Oury (2005), "Extensionality in the Calculus of Constructions"

Preserving judgemental equality is not the only reason, a useful translation of type theories fails to qualify as a type-theoretic transformation. A counter-example is the so called *functional functional interpretation*, a reformulation of the Dialectica interpretation [13, 63] for dependently-typed calculus, developed by Pédrot [113, 114]. The transformation preserves judgemental equality, but it does not preserve variable shapes: variables do not get mapped to variables.

[63]: Gödel (1958), "Über eine bisher nicht erweiterung des finiten standpunktes"
[13]: Avigad et al. (1998), "Gödel's Functional Interpretation"

[113]: Pédrot (2014), "A functional functional interpretation"
[114]: Pédrot (2015), "A Materialist Dialectica. (Une Dialectica matérialiste)"

The computational counterpart of the double negation translation, the *continuation passing style translation* ([54, 126]), fails to be a type-theoretic transformation for the same reason: we translate a variable a in a CPS translation to something like $\lambda\kappa.\kappa(\mathsf{a})$, which is not how type-theoretic transformations act on variables.

[54]: Fischer (1993), "Lambda-Calculus Schemata"
[126]: Reynolds (1972), "Definitional Interpreters for Higher-order Programming Languages"

The proposed notion of a type-theoretic transformation is just one possible version of a transformation between type theories. A more general notion is needed to cover the above counter-examples as well. It is likely that such a general notion of a transformation would

be easier to express in a more general syntax, that deals with variables, metavariables and symbols in a more uniform way. Another important generalisation lies in having more than the four proscribed judgement forms, as there could be several others. Examples are the interval judgement in the Cubical Type Theory [25, 42, 91] and the two additional judgement forms for asserting propositions and implications of propositions in the logic-enriched intuitionistic type theory of Aczel and Gambino [4]. We leave the exploration of such general transformations for future endeavors.

[42]: Cohen et al. (2015), "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom"
[25]: Bezem et al. (2014), "A Model of Type Theory in Cubical Sets"
[91]: Licata et al. (2015), "A Cubical Approach to Synthetic Homotopy Theory"

[4]: Aczel et al. (2002), "Collection Principles in Dependent Type Theory"

10. An Elaboration theorem

When designing a type theory, especially for using it in a proof assistant, one often faces a dilemma of how verbose the syntax should be. Terms annotated with full typing information are easily amenable to algorithmic processing and have good meta-theoretic properties. However, the syntax can quickly become too verbose to handle[1], so more economic terms that omit typing information are much more usable in practice.

1: We invite the reader to write a script in the Andromeda 2 proof assistant to experience this issue firsthand.

One common solution to this problem is to design two type theories: a fully annotated type theory $\mathcal{S}$ that resides in the kernel of the proof assistant and an economic one $\mathcal{T}$ for the users input. The latter version is then translated to the former via an *elaborator* i.e., the missing information is somehow recovered. We can see this process in practice, for example with Agda's [5] or Coq's [45] inferred implicit arguments, termination checking [1] (where evidence of termination is added), or universe polymorphism [134] (where explicit universe levels are calculated and constraints checked).

[5]: (2021), *The Agda proof assistant*

[45]: (2021), *The Coq proof assistant, version 2021.02.2*

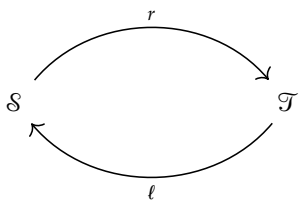
[1]: Abel (1998), *foetus - termination checker for simple functional programs*

[134]: Sozeau et al. (2014), "Universe Polymorphism in Coq"

Of course we want our economic version $\mathcal{T}$ to be conservative (Definition 9.3.3) over $\mathcal{S}$, namely that for every derivable type in $\mathcal{S}$, if we can provide a term of said type in $\mathcal{T}$, there is also a term of the original type.

In this chapter we use type-theoretic transformations to precisely define what elaboration is and prove the *elaboration theorem* (Theorem 10.2.1) which states that every finitary type theory can be elaborated. The second part of the chapter describes a basic relationship between the algorithmic content of an elaboration to $\mathcal{S}$ and type-checking of $\mathcal{T}$.

Using the vocabulary of finitary type theories, let us explain the intuition behind the elaboration theorem. We can summarize it in the following diagram.



We start with a finitary type theory $\mathcal{T}$, which represents the economic version. The fully-annotated type theory to which we elaborate is a *standard type theory* $\mathcal{S}$: indeed by definition of a standard type theory all specific object rules are symbol rules that faithfully record all the premises. We have a "forgetful" type-theoretic transformation $r\colon \mathcal{S} \to \mathcal{T}$, called the *retrogression transformation*, which erases the annotations, but is still conservative. The interesting part is in the

other direction, the so called *elaboration map* ℓ, which maps *derivations*[2] in the finitary type theory $\mathcal{T}$ to judgements in the standard type theory $\mathcal{S}$.

2: Note that the elaboration map ℓ is not a type-theoretic transformation as it does not work syntactically, but relies on the entire derivations.

10.1. Elaboration

10.1.1. Definition of elaboration

Recall that a transformation of type theories $f : \mathcal{T} \to \mathcal{U}$ is *conservative* if for every strongly derivable rule-boundary $\Theta \Longrightarrow \mathcal{b}$ in $\mathcal{T}$ it holds that if $f_*\Theta \Longrightarrow f_*\mathcal{b}\boxed{e}$ is derivable in $\mathcal{U}$ then we have e' such that $\Theta \Longrightarrow \mathcal{b}\boxed{e'}$ is derivable in $\mathcal{T}$.

For elaboration to behave as expected, the retrogression map needs to preserve enough of the derivability structure, namely it needs to be conservative (Definition 9.3.3) and a cover (Definition 9.3.4).

The other equally important aspect of preserving derivability is whether derivable judgements of a finitary type theory $\mathcal{T}$ have an elaboration in the standard type theory $\mathcal{S}$. Since the elaboration map depends on the derivations, we can hardly expect it to work on every derivable judgement, but rather on every *strongly derivable* judgement, because we also need to elaborate the context of the judgement.

Recall that a type-theoretic transformation $f : \mathcal{T} \to \mathcal{U}$ is a *cover*, if it is surjective on strongly derivable judgements.

With that in mind we provide the *elaboration map* ℓ that maps derivations of $\mathcal{T}$ to appropriate pieces of syntax of $\mathcal{S}$ and acts like a section of the retrogression transformation r. But before we can specify ℓ, we need to define what a good candidate for such a map is, what precisely we mean by being a form of a section of r.

Definition 10.1.1 For a type-theoretic transformation $r : \Sigma_{\mathcal{S}} \to \Sigma_{\mathcal{T}}$ form a standard type theory to a finitary type theory an *elaboration candidate* is an element of the fiber of r, specifically

- For a metacontext Θ in $\mathcal{T}$ an elaboration candidate is a metacontext Θ' over $\Sigma_{\mathcal{S}}$ such that $r_*(\Theta') = \Theta$.
- For metacontexts Θ and Θ' as above and for a variable context Γ over Θ in $\mathcal{T}$ an elaboration candidate for Γ is a variable context Γ' over Θ' over $\Sigma_{\mathcal{S}}$ such that $r_*(\Gamma') = \Gamma$.
- For contexts $\Theta;\Gamma$ and $\Theta';\Gamma'$ as above and for a boundary $\mathcal{B}$ over $\Theta;\Gamma$ in $\mathcal{T}$ an elaboration candidate for $\mathcal{B}$ is a boundary $\mathcal{B}'$ over $\Theta';\Gamma'$ over $\Sigma_{\mathcal{S}}$ such that $r_*(\mathcal{B}') = \mathcal{B}$.
- For contexts $\Theta;\Gamma$ and $\Theta';\Gamma'$ and boundaries $\mathcal{B}$ and $\mathcal{B}'$ as above, for a judgement $\mathcal{B}\boxed{e}$ over $\Theta;\Gamma$ in $\mathcal{T}$ an elaboration candidate for $\mathcal{B}\boxed{e}$ is a judgement $\mathcal{B}'\boxed{e'}$ over $\Theta';\Gamma'$ over $\Sigma_{\mathcal{S}}$ such that $r_*(e') = e$.

Note that since r is a syntactic transformation whose action preserves judgement forms[3], the form of an elaboration candidate $\mathcal{B}'$ (or $\mathcal{J}'$) is the same as the form of $\mathcal{B}$ (or $\mathcal{J}$).

3: A type judgement $\Theta;\Gamma \vdash A$ type is mapped to $r_*\Theta; r_*\Gamma \vdash r_*A$ type and similarly for other judgement and boundary forms.

An elaboration map for a type-theoretic transformation is a choice of elaboration candidates based on the derivations it takes as input. The choice is made in a way that preserves derivability: starting with a derivation of a judgement in $\mathcal{T}$, the elaboration map gives a derivable judgement in $\mathcal{S}$. We make this precise in the following definition.

Definition 10.1.2 Let $r : \mathcal{S} \to \mathcal{T}$ be a type-theoretic transformation form a standard type theory to a finitary type theory. If the following holds:

1. If $\mathscr{D}_\Theta$ is a derivation of $\vdash_{\mathcal{T}} \Theta$ mctx then there is an elaboration candidate $\ell_m(\mathscr{D}_\Theta)$ for Θ such that
$$\vdash_{\mathcal{S}} \ell_m(\mathscr{D}_\Theta) \text{ mctx}$$
is derivable in $\mathcal{S}$.
2. If $\mathscr{D}_\Gamma$ is a derivation of $\Theta \vdash_{\mathcal{T}} \Gamma$ vctx and Θ' is a derivable elaboration candidate for Θ then there is an elaboration candidate $\ell_v(\Theta', \mathscr{D}_\Gamma)$ for Γ such that
$$\Theta' \vdash_{\mathcal{S}} \ell_v(\Theta', \mathscr{D}_\Gamma) \text{ vctx}$$
is derivable in $\mathcal{S}$.
3. If $\mathscr{D}_{\mathscr{B}}$ is a derivation of $\Theta; \Gamma \vdash_{\mathcal{T}} \mathscr{B}$ and Θ' and Γ' are derivable elaboration candidates for Θ and Γ respectively, then there is an elaboration candidate $\ell_b(\Theta', \Gamma', \mathscr{D}_{\mathscr{B}})$ for $\mathscr{B}$ such that
$$\Theta'; \Gamma' \vdash_{\mathcal{S}} \ell_b(\Theta', \Gamma', \mathscr{D}_{\mathscr{B}})$$
is derivable in $\mathcal{S}$.
4. If $\mathscr{D}_{\mathscr{J}}$ is a derivation of $\Theta; \Gamma \vdash_{\mathcal{T}} \mathscr{B}\boxed{e}$ and Θ', Γ' and $\mathscr{B}'$ are derivable elaboration candidates for Θ, Γ and $\mathscr{B}$ respectively, then there is an elaboration candidate $\ell_j(\Theta', \Gamma', \mathscr{B}', \mathscr{D}_{\mathscr{J}})$ for $\mathscr{B}\boxed{e}$ such that
$$\Theta'; \Gamma' \vdash_{\mathcal{S}} \ell_j(\Theta', \Gamma', \mathscr{B}', \mathscr{D}_{\mathscr{J}})$$
is derivable in $\mathcal{S}$.

Then we write the partial maps ℓ_m, ℓ_v, ℓ_b and ℓ_j as ℓ, which we call the *elaboration map* of r.

The sentence "There is an elaboration candidate" is meant constructively.

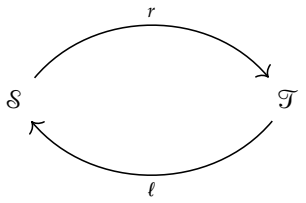
We think of the elaboration map ℓ as a section of the retrogression transformation r. However, since ℓ is not a type-theoretic transformation, but rather a partial map on the derivations of $\mathcal{T}$, it is not really a section. It takes more information, an entire derivation, to get a section-like behavior on the conclusion of the derivation.

Recall that a *section* of a type-theoretic transformation $f : \mathcal{T} \to \mathcal{U}$ is a type-theoretic transformation $s : \mathcal{U} \to \mathcal{T}$ such that for every expression e in $\mathcal{U}$ it holds that
$$(f \circ s)_* e = e.$$

We now have all the ingredients to give a formal definition of an elaboration.

Definition 10.1.3 An *elaboration* of a finitary type theory $\mathcal{T}$ is a standard type theory $\mathcal{S}$ with a conservative type-theoretic transformation $r : \mathcal{S} \to \mathcal{T}$ called the *retrogression transformation* and an elaboration map ℓ of r.

We can summarize the concept of an elaboration in the following diagram.



We keep in mind that the retrogression transformation is a type-theoretic transformation in the sense of Definition 9.1.2, but the elaboration map is not.

In the Definition 10.1.3 of an elaboration we did not need to specify that the retrogression map is a cover, because the elaboration map makes sure of it as seen in the following corollary.

Corollary 10.1.4 The retrogression map $r : \mathcal{S} \to \mathcal{T}$ is a cover.

Proof. Let $\Theta; \Gamma \vdash_{\mathcal{T}} \mathcal{B}\boxed{e}$ be a strongly derivable judgement. Then we have derivations

$$\begin{array}{lll} \mathscr{D}_\Theta & \text{of} & \vdash_{\mathcal{T}} \Theta\ \mathsf{mctx} \\ \mathscr{D}_\Gamma & \text{of} & \Theta \vdash_{\mathcal{T}} \Gamma\ \mathsf{vctx} \\ \mathscr{D}_{\mathcal{B}} & \text{of} & \Theta; \Gamma \vdash_{\mathcal{T}} \mathcal{B} \\ \mathscr{D}_{\mathcal{J}} & \text{of} & \Theta; \Gamma \vdash_{\mathcal{T}} \mathcal{B}\boxed{e} \end{array}$$

where $\mathscr{D}_{\mathcal{B}}$ by Theorem 5.1.6. By Definition 10.1.2 of the elaboration map we get the derivations of

$$\begin{aligned} \vdash_{\mathcal{S}} \ell(\mathscr{D}_\Theta)\ &\mathsf{mctx} \\ \ell(\mathscr{D}_\Theta) \vdash_{\mathcal{S}} \ell(\ell(\mathscr{D}_\Theta), \mathscr{D}_\Gamma)\ &\mathsf{vctx} \\ \ell(\mathscr{D}_\Theta); \ell(\ell(\mathscr{D}_\Theta), \mathscr{D}_\Gamma) \vdash_{\mathcal{S}} \ell(\ell(\mathscr{D}_\Theta), &\ell(\ell(\mathscr{D}_\Theta), \mathscr{D}_\Gamma), \mathscr{D}_{\mathcal{B}}) \\ \ell(\mathscr{D}_\Theta); \ell(\ell(\mathscr{D}_\Theta), \mathscr{D}_\Gamma) \vdash_{\mathcal{S}} \ell(\ell(\mathscr{D}_\Theta), &\ell(\ell(\mathscr{D}_\Theta), \mathscr{D}_\Gamma), \ell(\ell(\mathscr{D}_\Theta), \ell(\ell(\mathscr{D}_\Theta), \mathscr{D}_\Gamma), \mathscr{D}_{\mathcal{B}}), \mathscr{D}_{\mathcal{J}}) \end{aligned}$$

which are elaboration candidates for $\Theta, \Gamma, \mathcal{B}$ and $\mathcal{B}\boxed{e}$ respectively. □

Another important observation is that conservativity is enough of a restriction on the retrogression transformation r, that we do not have much choice when defining the elaboration map ℓ because it is unique up to judgemental equality on strongly derivable judgements. To prove that, we need the folowing lemma about conservative transformations.

Lemma 10.1.5 Let $f : \mathcal{U} \to \mathcal{V}$ be a conservative type-theoretic transformation between type theories $\mathcal{U}$ and $\mathcal{V}$. Suppose

$$\Theta; \Gamma \vdash_{\mathcal{V}} \mathcal{b}\boxed{e}$$

is a strongly derivable object judgement in $\mathcal{V}$ and

$$\Theta'; \Gamma' \vdash_{\mathcal{U}} \mathcal{b}'\boxed{e'} \qquad \text{and} \qquad \Theta'; \Gamma' \vdash_{\mathcal{U}} \mathcal{b}''\boxed{e''}$$

Both $\mathcal{b}'\boxed{e'}$ and $\mathcal{b}''\boxed{e''}$ are in the same context $\Theta'; \Gamma'$.

are strongly derivable object judgements in $\mathcal{U}$ such that

$$f_*(\Theta';\Gamma' \vdash_{\mathcal{U}} \mathcal{b}'\boxed{e'}) = f_*(\Theta';\Gamma' \vdash_{\mathcal{U}} \mathcal{b}''\boxed{e''}) = \Theta;\Gamma \vdash_{\mathcal{V}} \mathcal{b}\boxed{e}.$$

Then the equation

$$\Theta';\Gamma' \vdash_{\mathcal{U}} \mathcal{b}'\boxed{e' \equiv e''} \text{ by } \star$$

is strongly derivable in $\mathcal{U}$.

Proof. By assumption contexts $\Theta;\Gamma$ and $\Theta';\Gamma'$ are derivable. We promote the variable context Γ to a metavariable context using Proposition 4.3.6 to get a strongly derivable judgement

$$\Xi;[] \vdash_{\mathcal{V}} \mathcal{b}\boxed{e}$$

for $\Xi = (\Theta, \Gamma)$. Similarly we get that

$$\Xi';[] \vdash_{\mathcal{U}} \mathcal{b}'\boxed{e'} \qquad \text{and} \qquad \Xi';[] \vdash_{\mathcal{U}} \mathcal{b}''\boxed{e''}$$

are also derivable for $\Xi' = (\Theta', \Gamma')$. Since $\mathcal{b}$ is an object boundary we consider the following two cases:

- if $\mathcal{b} = \square$ type: the judgement

  $$\Xi;[] \vdash_{\mathcal{V}} e \equiv e \text{ by } \star$$

  is derivable by the rule TT-Ty-Refl. Conservativity of f with strongly derivable boundary

  $$\Xi';[] \vdash_{\mathcal{U}} e' \equiv e'' \text{ by } \square$$

  gives us the derivability of the corresponding judgement. Proposition 4.3.6 gives us the desired judgement

  $$\Theta';\Gamma' \vdash_{\mathcal{U}} e' \equiv e'' \text{ by } \star.$$

- if $\mathcal{b} = \square : A$: let $\mathcal{b}' = \square : A'$ and $\mathcal{b}'' = \square : A''$. The judgement

  $$\Xi';[] \vdash_{\mathcal{V}} A' \equiv A'' \text{ by } \star$$

  is derivable by the rule TT-Ty-Refl on A and conservativity of f. We can therefore convert e'' to the type A'. We again use conservativity of f, this time with the strongly derivable boundary

  $$\Xi';[] \vdash_{\mathcal{U}} e' \equiv e'' : A' \text{ by } \square$$

  to get the derivability of the corresponding judgement. Proposition 4.3.6 gives us the desired judgement

  $$\Theta';\Gamma' \vdash_{\mathcal{U}} e' \equiv e'' : A' \text{ by } \star. \qquad \square$$

Corollary 10.1.6 The elaboration map ℓ for a retrogression transformation $r : \mathcal{S} \to \mathcal{T}$ is unique up to judgemental equality: Suppose

$\mathscr{D}_{\mathcal{J}}$ is a derivation of

$$\Theta;\Gamma \vdash_{\mathcal{T}} \mathscr{b}\boxed{e}$$

where $\mathscr{b}$ is an object boundary, $\Theta';\Gamma'$ is an elaboration candidate for context $\Theta;\Gamma$ and $\mathscr{b}'$ is an elaboration candidate for $\mathscr{b}$. If

$$\Theta';\Gamma' \vdash_{\mathcal{S}} \mathscr{b}'\boxed{e'}$$

is strongly derivable and $r_*(e') = e$, then for

$$\mathscr{b}'\boxed{e''} = \ell(\Theta', \Gamma', \mathscr{b}', \mathscr{D}_{\mathcal{J}})$$

the equation

$$\Theta';\Gamma' \vdash_{\mathcal{S}} \mathscr{b}'\boxed{e' \equiv e''} \text{ by } \star.$$

is also derivable.

Note that strong derivability of $\Theta';\Gamma' \vdash_{\mathcal{S}} \mathscr{b}'\boxed{e'}$ implies that the context $\Theta;\Gamma$ is also derivable, because r is a type-theoretic transformation that preserves derivability.

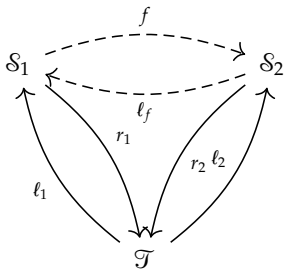
Proof. The proof is a direct application of Lemma 10.1.5. □

10.1.2. The universal property of elaboration

For a finitary type theory $\mathcal{T}$ every elaboration as defined in Definition 10.1.3 is equivalent in the sense of the following universal property.

Theorem 10.1.7 (The universal property of elaboration) Let $\mathcal{T}$ be a finitary type theory and $(\mathcal{S}_1, r_1, \ell_1)$ and $(\mathcal{S}_2, r_2, \ell_2)$ elaborations of $\mathcal{T}$. Then there exists a conservative type-theoretic transformation $f\colon \mathcal{S}_1 \to \mathcal{S}_2$ with an elaboration map $\ell_f\colon \mathcal{S}_2 \to \mathcal{S}_1$ such that $r_2 \circ f = r_1$ and f is unique up to judgemental equality.

Proof. The situation is summarized by the following diagram:



We need to construct a type theoretic transformation f such that the diagram commutes. We know that $\mathcal{S}_1$ and $\mathcal{S}_2$ are standard type theories, so they have just symbol rules and equality rules as their specific rules. Let $\mathcal{S}_1 = (R_i)_{i\in \mathrm{I}}$ and I is ordered with a well-founded order $\sqsubset$. We build the transformation f inductively on the order $\sqsubset$.

Suppose $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ and $R_i = \Xi \Longrightarrow \mathscr{b}\boxed{e}$ is a specific rule of $\mathcal{S}_1$ and we have already defined the type-theoretic transformation $f : \mathcal{S}_1^i \to \mathcal{S}_2$, where $\mathcal{S}_1^i$ is the fragment of $\mathcal{S}_1$ defined by the index

set $\downarrow i = \{j \mid j \sqsubset i\}$ and with the signature containing only the symbols that have a symbol rule in the fragment. We want to extend the definition of f.

Since $\mathcal{S}_1$ is a finitary type theory $\vdash_{\mathcal{S}_1^i} \Xi\ \mathsf{mctx}$ and $\Xi; [] \vdash_{\mathcal{S}_1^i} \mathcal{b}$ are derivable in $\mathcal{S}_1^i$, so $\vdash_{\mathcal{S}_2} f_*\Xi\ \mathsf{mctx}$ and $f_*\Xi; [] \vdash_{\mathcal{S}_2} f_*\mathcal{b}$ are also derivable and

$$(r_2)_*(f_*\Xi; [] \vdash_{\mathcal{S}_2} f_*\mathcal{b}) = (r_1)_*\Xi; [] \vdash_{\mathcal{T}} (r_1)_*\mathcal{b}.$$

Because r_1 is a type-theoretic transformation it maps R_i to a derivation $\mathcal{D}$ in $\mathcal{T}$. We consider the cases for an object rule and an equality rule:

- If $e = \mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})$, let e_{S} be the expression determined by

  $$(f_*\mathcal{b})\boxed{e_{\mathsf{S}}} = \ell_2(f_*\Xi, [], f_*\mathcal{b}, \mathcal{D}).$$

  We set

  $$f(\mathsf{S}) = e_{\mathsf{S}}.$$

  Because elaboration map ℓ_2 preserves derivability of strongly derivable judgements we can obtain a derivation $\mathcal{D}'$ of

  $$f_*\Xi; [] \vdash_{\mathcal{S}_2} (f_*\mathcal{b})\boxed{e_{\mathsf{S}}}$$

  so we set f to map the rule R_i to the derivation $\mathcal{D}'$.
- If $e = \star$, then again because elaboration map ℓ_2 preserves derivability of strongly derivable judgements we can obtain a derivation $\mathcal{D}'$ of

  $$f_*\Xi; [] \vdash_{\mathcal{S}_2} (f_*\mathcal{b})\boxed{\star}$$

  and we set f to map the rule R_i to the derivation $\mathcal{D}'$.

With this data the extended f is indeed a type-theoretic transformation and also

$$(r_2)_*(f(\mathsf{S})) = (r_2)_*(e_{\mathsf{S}}) = (r_1)_*(\mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})) = r_1(\mathsf{S}).$$

At the last step we use the fact that action of a transformation on a generically applied metavaraible does nothig.

By induction we get a type-theoretic transformation $f : \mathcal{S}_1 \to \mathcal{S}_2$. To show it is conservative suppose $\Theta; \Gamma \vdash_{\mathcal{S}_1} \mathcal{b}$ is a strongly derivable boundary in $\mathcal{S}_1$ such that

$$f_*\Theta; f_*\Gamma \vdash_{\mathcal{S}_2} (f_*\mathcal{b})\boxed{e}$$

is derivable in $\mathcal{S}_2$. Then

$$(r_2)_*(f_*\Theta); (r_2)_*(f_*\Gamma) \vdash_{\mathcal{T}} (r_2)_*((f_*\mathcal{b})\boxed{e})$$

is strongly derivable in $\mathcal{T}$ with some derivation $\mathcal{D}$. Since elaboration map ℓ_1 preserves derivability the judgement

$$\Theta; \Gamma \vdash_{\mathcal{S}_1} \mathcal{b}\boxed{\ell_1(\Theta; \Gamma, \mathcal{b}, \mathcal{D})}$$

is derivable thus proving conservativity.

The elaboration map ℓ_f maps a derivation $\mathcal{D}$ in $\mathcal{S}_2$ to $\ell_1((r_2)_*\mathcal{D})$. To be more precise, for a strogny derivable judgement $\Theta; \Gamma \vdash_{\mathcal{S}_2} \mathcal{B}\boxed{e}$ in $\mathcal{S}_2$

and Θ', Γ' and $\mathscr{B}'$ such that

$$f_*\Theta' = \Theta \qquad f_*\Gamma' = \Gamma \qquad f_*\mathscr{B}' = \mathscr{B}$$

we define

$$\ell_f(\Theta', \Gamma', \mathscr{B}', \mathscr{D}) = \ell_1(\Theta', \Gamma', \mathscr{B}', (r_2)_*\mathscr{D})$$

and similarly for contexts and boundaries. The elaboration map ℓ_f preserves derivability because both r_2 and ℓ_1 do so.

To prove that f is unique up to judgemental equality, suppose there is another type-theoretic tranformation $g\colon \mathcal{S}_1 \longrightarrow \mathcal{S}_2$ with an elaboration map ℓ_g, such that $r_2 \circ g = r_1$. Since $\mathcal{S}_1$ is a standard type theory, object rules of $\mathcal{S}_1$ are symbol rules and we prove that f and g are judgementally equal (Definition 9.1.5) by induction on the ordering of the rules.

Suppose $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ and $R_i = \Xi \Longrightarrow \mathscr{b}\boxed{\mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})}$ is a symbol rule of $\mathcal{S}_1$ and we have already established that f and g, restricted to the fragment $\mathcal{S}_1^i$ are judgementally equal, where $\mathcal{S}_1^i$ is the fragment of $\mathcal{S}_1$ defined by the index set $\downarrow i = \{j \mid j \sqsubset i\}$ and with the signature containing only the symbols that have a symbol rule in the fragment. We want to derive

$$f_*\Xi; [] \vdash_{\mathcal{S}_2} (f_*\mathscr{b})\boxed{f(\mathsf{S}) \equiv g(\mathsf{S})}$$

Since $\Xi; [] \vdash_{\mathcal{S}_1} \mathscr{b}$ is derivable in $\mathcal{S}_1$ and on this fragment f and g are judgementally equal, by Lemma 9.1.9 the instantiation

$$I = \langle \mathsf{M}_1 \mapsto \widehat{\mathsf{M}_1}, \ldots, \mathsf{M}_n \mapsto \widehat{\mathsf{M}_n} \rangle$$

of $g_*\Xi$ over $f_*\Xi; []$ is derivable so we can act with I on the derivable judgement $g_*\Xi; [] \vdash_{\mathcal{S}_2} (g_*\mathscr{b})\boxed{g(\mathsf{S})}$ to obtain a derivation of $f_*\Xi; [] \vdash_{\mathcal{S}_2} (g_*\mathscr{b})\boxed{g(\mathsf{S})}$. We conclude the proof by using Lemma 10.1.5 with the derivable judgements $f_*\Xi; [] \vdash_{\mathcal{S}_2} (f_*\mathscr{b})\boxed{f(\mathsf{S})}$ and $f_*\Xi; [] \vdash_{\mathcal{S}_2} (g_*\mathscr{b})\boxed{g(\mathsf{S})}$. □

The roles of $\mathcal{S}_1$ and $\mathcal{S}_2$ in Theorem 10.1.7 are symmetric, the standard type theories $\mathcal{S}_1$ and $\mathcal{S}_2$ are elaborations of each other. Also note that by Corollary 10.1.4 the type-theoretic transformation f is a cover.

10.2. The elaboration theorem

The universal property of elaboration tells us that all elaborations of a finitary type theory are equivalent, if they exist. We can now give the formal statement of the elaboration theorem, taking us a step further as it ensures the existence of elaborations.

Theorem 10.2.1 (The Elaboration Theorem) Every finitary type theory has an elaboration.

The rest of the section is dedicated to proving the elaboration theorem by constructing an elaboration for finitary type theories.

We fix a finitary type theory $\mathcal{T}$ and let I be the indexing set for the specific rules of $\mathcal{T}$ ordered by the well-found order $\sqsubset$. We need to construct a standard type theory $\mathcal{S}$ and a retrogression transformation $r : \mathcal{S} \longrightarrow \mathcal{T}$.

10.2.1. Syntactic part of elaboration $\mathcal{S}$

First, we make a signature $\Sigma_{\mathcal{S}}$ induced by the object rules of $\mathcal{T}$. For every specific object rule $R_i = \Theta \Longrightarrow \mathcal{C}\boxed{e}$ indexed by i in the theory $\mathcal{T}$ we introduce a symbol

$$\mathsf{S}_{\left(i,\Theta\Longrightarrow\mathcal{C}\boxed{e}\right)}$$

with arity $(\mathsf{cl}(\mathcal{C}), \mathsf{ar}(\Theta))$.

We can now define the syntactic part of the retrogression transformation

$$r : \Sigma_{\mathcal{S}} \longrightarrow \Sigma_{\mathcal{T}}$$

as follows: if $R_i = \Theta \Longrightarrow \mathcal{C}\boxed{e}$ is an object rule of $\mathcal{T}$, then

$$r(\mathsf{S}_{\left(i,\Theta\Longrightarrow\mathcal{C}\boxed{e}\right)}) = e.$$

This gives a syntactic transformation. To check that r is indeed a type-theoretic transformation we first need to give rules for the type theory $\mathcal{S}$. For that we inductively define the syntactic part of the elaboration map ℓ for r with the help of the following auxiliary notions.

Definition 10.2.2 For the retrogression transformation $r : \Sigma_{\mathcal{S}} \longrightarrow \Sigma_{\mathcal{T}}$ as above and for a specific object rule $R_i = \Theta \Longrightarrow \mathcal{C}\boxed{e}$ in $\mathcal{T}$ with

$$\Theta = [\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n]$$

an *elaboration candidate* for R_i is a well-formed judgement

$$[\mathsf{M}_1{:}\mathcal{B}'_1, \ldots, \mathsf{M}_n{:}\mathcal{B}'_n]; [] \vdash \mathcal{C}'\boxed{\mathsf{S}_{\left(i,\Theta\Longrightarrow\mathcal{C}\boxed{e}\right)}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})}$$

such that

$$r_*(\mathcal{B}'_j) = \mathcal{B}_j \text{ for } j = 1, \ldots, n$$
$$r_*(\mathcal{C}') = \mathcal{C}$$

We call a fragment $\mathcal{T}^{\mathsf{fr}}$ of the theory $\mathcal{T}$ an *elaborative fragment* if every specific object rule of $\mathcal{T}^{\mathsf{fr}}$ has a chosen elaboration candidate.

We do not need to define elaboration candidates for equality rules, because the head of an equality judgement is just $\star$ and since r preserves judgement forms we know what the elaboration candidate is.

The next lemma constructs the syntactic part of the elaboration map for r.

Lemma 10.2.3 The following statements hold.

1. Let $i \in \mathrm{I}$ such that the fragment $\mathcal{T}^i$ of $\mathcal{T} = (R_j)_{j\in\mathrm{I}}$ induced by the index set $\downarrow i = \{j \in \mathrm{I} \mid j \sqsubset i\}$ is an elaborative fragment of $\mathcal{T}$. Then there is an elaboration candidate R_i' for R_i.
2. Let $\mathcal{T}^{\mathsf{fr}}$ be an elaborative fragment of $\mathcal{T}$.
   a) If $\mathscr{D}_\Theta$ is a derivation of $\vdash_{\mathcal{T}^{\mathsf{fr}}} \Theta\ \mathsf{mctx}$ then there is an elaboration candidate $\ell_m(\mathscr{D}_\Theta)$ for Θ.
   b) If $\mathscr{D}_\Gamma$ is a derivation of $\Theta \vdash_{\mathcal{T}^{\mathsf{fr}}} \Gamma\ \mathsf{vctx}$ and Θ' is an elaboration candidate for Θ then there is an elaboration candidate $\ell_v(\Theta', \mathscr{D}_\Gamma)$ for Γ.
   c) If $\mathscr{D}_{\mathscr{B}}$ is a derivation of $\Theta;\Gamma \vdash_{\mathcal{T}^{\mathsf{fr}}} \mathscr{B}$ and Θ' and Γ' are elaboration candidates for Θ and Γ respectively, then there is an elaboration candidate $\ell_b(\Theta', \Gamma', \mathscr{D}_{\mathscr{B}})$ for $\mathscr{B}$.
   d) If $\mathscr{D}_{\mathcal{J}}$ is a derivation of $\Theta;\Gamma \vdash_{\mathcal{T}^{\mathsf{fr}}} \mathscr{B}\boxed{e}$ and Θ', Γ', $\mathscr{B}'$ are elaboration candidates for $\Theta, \Gamma, \mathscr{B}$ respectively, then there is an elaboration candidate $\ell_j(\Theta', \Gamma', \mathscr{B}', \mathscr{D}_{\mathcal{J}})$ for $\mathcal{J}$.
   e) If $\mathscr{D}_{\mathcal{J}}$ is a derivation of $\Theta;\Gamma \vdash_{\mathcal{T}^{\mathsf{fr}}} \mathcal{J}$ and Θ' and Γ' are elaboration candidates for Θ and Γ respectively, then there is an elaboration candidate $\ell_{jb}(\Theta', \Gamma', \mathscr{D}_{\mathcal{J}})$ for $\mathcal{J}$.

The partial map ℓ_{jb} is not a part of Definition 10.1.2. However, we include it to help with induction, as sometimes the expected boudary is not provided. Specifically, when converting a term using TT-Conv-Tm along some equation $A \equiv B$, there is no provided elaboration candidate for the type A , but we can still compute it from the derivation. The index in ℓ_{jb} stands for "judgement and boundary" as it provides both elaboration candidates.

Proof. We construct ℓ by mutual recursion on all five parts of the lemma.

Part (1): Let $R_i = (\Theta \Longrightarrow \mathscr{b}\boxed{e})$ be a specific rule in the theory $\mathcal{T}$ and $\mathcal{T}^i$ an elaborative fragment of $\mathcal{T}$ induced by the index set

$$\downarrow i = \{j \in \mathrm{I} \mid j \sqsubset i\}.$$

Since $\mathcal{T}$ is finitary, the rule R_i is finitary with respect to $\mathcal{T}^i$, so we have derivations

$$\frac{\mathscr{D}_1}{\vdash_{\mathcal{T}^i} \Theta\ \mathsf{mctx}} \qquad \frac{\mathscr{D}_2}{\Theta;[] \vdash_{\mathcal{T}^i} \mathscr{b}}$$

By induction hypothesis with the fragment $\mathcal{T}^i$ we have elaboration candidates

$$\Theta' = \ell_m\left(\frac{\mathscr{D}_1}{\vdash_{\mathcal{T}^i} \Theta\ \mathsf{mctx}}\right) \qquad \mathscr{b}' = \ell_b\left(\Theta', [], \frac{\mathscr{D}_2}{\Theta;[] \vdash_{\mathcal{T}^i} \mathscr{b}}\right)$$

We consider the following two cases:

- If R_i is an equality rule, we define

  $$R_i' = (\Theta' \Longrightarrow \mathscr{b}'\boxed{\star})$$

  which is a suitable elaboration candidate by the following equation:

  $$r_*(R_i') = r_*(\Theta' \Longrightarrow \mathscr{b}'\boxed{\star}) = (r_*\Theta' \Longrightarrow (r_*\mathscr{b}')\boxed{r_*\star}) = R_i.$$

- If R_i is an object specific rule, we define

$$R_i' = (\Theta' \Longrightarrow \mathscr{b}' \boxed{\mathsf{S}_{\left(i, \Theta \Longrightarrow \mathscr{b} \boxed{e}\right)}})$$

  which is a suitable elaboration candidate for the rule R_i because $r_*(\mathsf{S}_{\left(i, \Theta \Longrightarrow \mathscr{b} \boxed{e}\right)}) = e$.

Part (2): Suppose $\mathcal{T}^{\mathsf{fr}}$ is an elaborative fragment for $\mathcal{T}$. All the following judgements and derivations of the proof are made in this fragment, unless specified otherwise. For clarity we leave out the annotations $\vdash_{\mathcal{T}^{\mathsf{fr}}}$ on the judgements.

Part (2a): Suppose we have a derivation $\mathscr{D}_\Theta$ of a metacontext Θ. We consider cases depending on this derivation.

Case MCTX-EMPTY: We define $\ell_m(\mathscr{D}_\Theta) = []$, for which the equation

$$r_*(\ell_m(\mathscr{D}_\Theta)) = []$$

holds trivially.

Case MCTX-EXTEND: The derivation ends in

$$\frac{\dfrac{\mathscr{D}_1}{\vdash \Xi\ \mathsf{mctx}} \qquad \dfrac{\mathscr{D}_2}{\Xi; [] \vdash \mathscr{B}} \qquad \mathsf{M} \notin |\Xi|}{\vdash \langle \Xi, \mathsf{M}{:}\mathscr{B} \rangle\ \mathsf{mctx}}$$

By induction hypothesis on the first premise we have an elaboration candidate

$$\Xi' = \ell_m\left(\frac{\mathscr{D}_1}{\vdash \Xi\ \mathsf{mctx}}\right).$$

Induction hypothesis on the second premise for elaboration candidates Ξ' and $[]$ gives us an elaboration candidate for $\mathscr{B}$:

$$\mathscr{B}' = \ell_b\left(\Xi', [], \left(\frac{\mathscr{D}_2}{\Xi; [] \vdash \mathscr{B}}\right)\right).$$

We define

$$\ell_m(\mathscr{D}_\Theta) = \langle \Xi', \mathsf{M}{:}\mathscr{B}' \rangle$$

and we check the equation

$$r_*(\ell_m(\mathscr{D}_\Theta)) = r_*(\langle \Xi', \mathsf{M}{:}\mathscr{B}' \rangle) = \langle r_*\Xi', \mathsf{M}{:}r_*\mathscr{B}' \rangle = \langle \Xi, \mathsf{M}{:}\mathscr{B} \rangle.$$

which shows that $\ell_m(\mathscr{D}_\Theta)$ is indeed an elaboration candidate for Θ.

Part (2b): Suppose we have a derivation $\mathscr{D}$ of a variable context over a metacontext Θ and an elaboration candidate Θ' for Θ. We again consider cases for the derivation of the variable context.

Case VCTX-EMPTY:

We define $\ell_v(\Theta', \mathscr{D}_{[]}) = []$, for which the equation

$$r_*(\ell_v(\Theta', \mathscr{D}_{[]})) = []$$

holds trivially.

Case VCtx-Extend: The derivation ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta \vdash \Gamma\ \mathsf{vctx}} \qquad \dfrac{\mathscr{D}_2}{\Theta, \Gamma \vdash A\ \mathsf{type}} \qquad \mathsf{a} \notin |\Gamma|}{\Theta \vdash \langle \Gamma, \mathsf{a}{:}A \rangle\ \mathsf{vctx}}$$

By induction hypothesis on the first premise we have an elaboration candidate for Γ

$$\Gamma' = \ell_v \left(\Theta', \left(\dfrac{\mathscr{D}_1}{\Theta \vdash \Gamma\ \mathsf{vctx}} \right) \right).$$

Induction on the second premise with Θ' and Γ' gives us an elaboration candidate for $\Theta; \Gamma \vdash A\ \mathsf{type}$

$$A'\ \mathsf{type} = \ell_j \left(\Theta', \Gamma', \square\ \mathsf{type}, \dfrac{\mathscr{D}_2}{\Theta, \Gamma \vdash A\ \mathsf{type}} \right)$$

We define

$$\ell_v(\Theta', \mathscr{D}_\Gamma) = \langle \Gamma', \mathsf{a}{:}A' \rangle$$

and we check the equation

$$r_*(\ell(\mathscr{D}_\Gamma)) = r_*(\langle \Gamma', \mathsf{a}{:}A' \rangle) = \langle r_*\Gamma', \mathsf{a}{:}r_*A' \rangle = \langle \Gamma, \mathsf{a}{:}A \rangle.$$

verifying that $\ell_v(\Theta', \mathscr{D})$ is indeed an elaboration candidate.

Part (2c): Suppose we have a derivation $\mathscr{D}_{\mathscr{B}}$ of a boundary $\Theta; \Gamma \vdash \mathscr{B}$ and an elaboration candidate $\Theta'; \Gamma'$ for $\Theta; \Gamma$. We consider several cases depending on how the derivation $\mathscr{D}_{\mathscr{B}}$ ends.

Cases TT-Bdry-Ty, TT-Bdry-Tm, TT-Bdry-EqTy *or* TT-Bdry-EqTm: It is a direct use of the induction hypothesis. Let us take a look at TT-Bdry-EqTm as an example. The derivation $\mathscr{D}_{\mathscr{B}}$ ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash A\ \mathsf{type}} \qquad \dfrac{\mathscr{D}_2}{\Theta; \Gamma \vdash s : A} \qquad \dfrac{\mathscr{D}_3}{\Theta; \Gamma \vdash t : A}}{\Theta; \Gamma \vdash s \equiv t : A\ \mathsf{by}\ \square}$$

By induction hypothesis with $\Theta'; \Gamma'$ we get elaboration candidates

$$\ell_j \left(\Theta', \Gamma', \square\ \mathsf{type} \dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash A\ \mathsf{type}} \right) = A'\ \mathsf{type}$$

$$\ell_j \left(\Theta', \Gamma', \square : A', \dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash s : A} \right) = s' : A'$$

$$\ell_j \left(\Theta', \Gamma', \square : A', \dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash t : A} \right) = t' : A'$$

We define

$$\ell_b(\Theta', \Gamma', \mathscr{D}_{\mathscr{B}}) = s' \equiv t' : A'\ \mathsf{by}\ \square$$

which is a boundary elaboration candidate for $\mathscr{B}$.

Case TT-Bdry-Abstr: The derivation $\mathscr{D}_{\mathscr{B}}$ ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash A\ \mathsf{type}} \qquad \mathsf{a} \notin |\Gamma| \qquad \dfrac{\mathscr{D}_2}{\Theta;\Gamma, \mathsf{a}{:}A \vdash \mathscr{B}'[\mathsf{a}/x]}}{\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}'}$$

By induction hypothesis on the first premise with $\Theta';\Gamma'$ we get an elaboration candidate

$$A'\ \mathsf{type} = \ell_j\left(\Theta', \Gamma', \square\ \mathsf{type}, \dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash A\ \mathsf{type}}\right).$$

Since a is not in Γ, it also holds that $\mathsf{a} \notin \Gamma'$. Induction hypothesis on the third premise with Θ and $\langle \Gamma', \mathsf{a}{:}A' \rangle$ we get an elaboration candidate for $\mathscr{B}'[\mathsf{a}/x]$

$$\mathscr{B}'' = \ell_b\left(\Theta', \langle \Gamma', \mathsf{a}{:}A' \rangle, \dfrac{\mathscr{D}_2}{\Theta;\Gamma, \mathsf{a}{:}A \vdash \mathscr{B}'[\mathsf{a}/x]}\right).$$

We define

$$\ell_b\,(\Theta', \Gamma', \mathscr{D}) = \{x{:}A'\}\mathscr{B}''[x/\mathsf{a}]$$

and check the equation

$$r_*(\{x{:}A'\}\mathscr{B}''[x/\mathsf{a}]) = \{x{:}r_*(A')\}r_*(\mathscr{B}''[x/\mathsf{a}]) = \{x{:}A\}\mathscr{B}'.$$

Part (2d): Suppose we have a derivation $\mathscr{D}_{\mathscr{J}}$ of a judgement $\Theta;\Gamma \vdash \mathscr{B}\boxed{e}$ and elaboration candidates Θ', Γ', $\mathscr{B}'$ for Θ, Γ, $\mathscr{B}$ respectively. We consider several cases depending on how the derivation $\mathscr{D}_{\mathscr{J}}$ ends.

Case TT-Abstr: The construction is similar to the case TT-Bdry-Abstr. The derivation $\mathscr{D}_{\mathscr{J}}$ ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash A\ \mathsf{type}} \qquad \mathsf{a} \notin |\Gamma| \qquad \dfrac{\mathscr{D}_2}{\Theta;\Gamma, \mathsf{a}{:}A \vdash (\mathscr{B}''[\mathsf{a}/x])\boxed{e'[\mathsf{a}/x]}}}{\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}''\boxed{e'}}$$

The boundary $\mathscr{B}'$ is then an elaboration candidate for $\{x{:}A\}\ \mathscr{B}''$ and because r preserves forms of boundaries $\mathscr{B}' = \{x{:}A'\}\ \mathscr{B}'''$, where $\mathscr{B}'''$ is an elaboration candidate for $\mathscr{C}''[\mathsf{a}/x]$ in context $\Theta';\Gamma', \mathsf{a}{:}A'$. By induction hypothesis on the last premise we obtain

$$\mathscr{B}'''\boxed{e''} = \ell_j\left(\Theta', \langle \Gamma', \mathsf{a}{:}A' \rangle, \mathscr{B}''', \dfrac{\mathscr{D}_2}{\Theta;\Gamma, \mathsf{a}{:}A \vdash (\mathscr{B}''[\mathsf{a}/x])\boxed{e'[\mathsf{a}/x]}}\right)$$

and we set

$$\ell_j(\Theta', \Gamma', \mathscr{B}', \mathscr{D}_{\mathscr{J}}) = \mathscr{B}'\boxed{\{x\}(e''[x/\mathsf{a}])}.$$

This is a valid elaboration candidate for $\mathscr{B}\boxed{e}$ because

$$r_*(\{x\}e'[x/\mathsf{a}]) = \{x\}r_*(e'[x/\mathsf{a}]) = \{x\}(e[\mathsf{a}/x])[x/\mathsf{a}] = \{x\}e.$$

Case TT-VAR: The elaboration candidate is $\ell_j(\Theta', \Gamma', \mathscr{B}', \mathscr{D}_{\mathcal{J}}) = \mathsf{a} : \Gamma'(a)$. Since the names of the variable in a variable context are preserved with the syntactic transformation r and Γ' is an elaboration candidate for Γ, this is a well-formed judgement in the context Γ'.

Cases TT-TY-REFL, TT-TM-REFL, TT-TY-SYM, TT-TM-SYM, TT-TY-TRAN, TT-TM-TRAN, TT-CONV-EQ, TT-META-CONGR: We define the desired elaboration candidate

$$\ell_j(\Theta', \Gamma', \mathscr{B}', \mathscr{D}_{\mathcal{J}}) = \mathscr{B}'\boxed{\star}.$$

Cases TT-CONV-TM: The derivation $\mathscr{D}_{\mathcal{J}}$ ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash t : A} \qquad \dfrac{\mathscr{D}_2}{\Theta;\Gamma \vdash A \equiv B}}{\Theta;\Gamma \vdash t : B}$$

By induction hypothesis with Part (2e) on the first premise we obtain the elaboration candidate for $t : A$

$$t' : A' = \ell_{jb}\left(\Theta', \Gamma', \dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash t : A}\right)$$

and we define

$$\ell_j(\Theta', \Gamma', \mathscr{B}', \mathscr{D}_{\mathcal{J}}) = \mathscr{B}'\boxed{t'}$$

which is the desired elaboration candidate.

Cases TT-META: The derivation $\mathscr{D}_{\mathcal{J}}$ ends with

$$\dfrac{\begin{array}{c}\Theta(\mathsf{M}_k) = \{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\,\mathscr{b} \\ \dfrac{\mathscr{D}_j}{\Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]} \qquad \text{for } j = 1,\ldots,m \\ \dfrac{\mathscr{D}'}{\Theta;\Gamma \vdash \mathscr{b}[\vec{t}/\vec{x}]}\end{array}}{\Theta;\Gamma \vdash (\mathscr{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_k(\vec{t})}}$$

Since M_k is also in the domain of Θ', let $\Theta'(\mathsf{M}_k) = \{x_1{:}A'_1\}\cdots\{x_m{:}A'_m\}\mathscr{b}'$. By induction hypothesis we get

$$t'_j : A'_j[\vec{t}'_{(j)}/\vec{x}_{(j)}] = \ell_j\left(\Theta', \Gamma', \square : A'_j[\vec{t}'_{(j)}/\vec{x}_{(j)}], \dfrac{\mathscr{D}_{j1}}{\Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]}\right) \quad \text{for } j = 1,\ldots,m$$

We define

$$\ell_j(\Theta', \Gamma', \mathscr{B}', \mathscr{D}_{\mathcal{J}}) = (\mathscr{B}')\boxed{\mathsf{M}_k(\vec{t}')}$$

which is a good elaboration candidate.

Case Specific rule: Here we have to pay attention to the fragment in which the judgement happens: the derivation $\mathscr{D}_{\mathcal{J}}$ is in an elaborative fragment $\mathcal{T}^{\mathsf{fr}}$ of $\mathcal{T}$. Suppose the derivation $\mathscr{D}_{\mathcal{J}}$ ends with a specific

rule $R = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n] \Longrightarrow \mathscr{J}$ with a derivable (in $\mathscr{T}^{\mathsf{fr}}$) instantiation $I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$ over $\Theta;\Gamma$. By (1) we have an elaboration candidate $R' = [\mathsf{M}_1{:}\mathscr{B}'_1, \ldots, \mathsf{M}_n{:}\mathscr{B}'_n] \Longrightarrow \mathscr{C}'\boxed{e'}$ for R. Since I is derivable, we have derivations

$$\frac{\mathscr{D}_i}{\Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i}} \qquad \text{for } i = 1, \ldots, n.$$

By induction hypothesis we have elaboration candidates over $\Theta';\Gamma'$ for those derivable judgements:

$$I'_{(i)*}\mathscr{B}'_i\boxed{e'_i} = \ell_j\left(\Theta', \Gamma', I'_{(i)*}\mathscr{B}'_i, \frac{\mathscr{D}_i}{\Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i}}\right)$$

for $i = 1, \ldots, n$ where

$$I' = \langle \mathsf{M}_1 \mapsto e'_1, \ldots, \mathsf{M}_n \mapsto e'_n \rangle$$

and we define the elaboration candidate for $\mathscr{D}_{\mathscr{J}}$ to be

$$\ell_j(\Theta, \Gamma, \mathscr{B}', \mathscr{D}_{\mathscr{J}}) = \mathscr{B}'\boxed{I'_*e'}.$$

Since the equations $r_*(R') = R$ and $r_*(I') = I$ hold, we get the desired equation by Lemma 8.3.3.

Part (2e): Suppose we have a derivation $\mathscr{D}_{\mathscr{J}}$ of a judgement $\Theta;\Gamma \vdash \mathscr{J}$ and an elaboration candidate $\Theta';\Gamma'$ for $\Theta;\Gamma$. We consider several cases depending on how the derivation $\mathscr{D}_{\mathscr{J}}$ ends.

Case TT-Abstr: The construction is similar to the case TT-Bdry-Abstr.

Case TT-Var: The elaboration candidate is $\ell_{jb}(\Theta', \Gamma', \mathscr{D}_{\mathscr{J}}) = \mathsf{a} : \Gamma'(a)$. Since the names of the variable in a variable context are preserved with the syntactic transformation r and Γ' is an elaboration candidate for Γ, this is a well-formed judgement in the context Γ'.

Cases TT-Ty-Refl, TT-Tm-Refl, TT-Ty-Sym, TT-Tm-Sym: It is a direct application of the induction hypothesis. We take a look at the TT-Tm-Refl case. The derivation $\mathscr{D}_{\mathscr{J}}$ ends with

$$\frac{\dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash t : A}}{\Theta;\Gamma \vdash t \equiv t : A}$$

By induction hypothesis with $\Theta';\Gamma'$ we get an elaboration candidate

$$t' : A' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_1}{\Theta;\Gamma \vdash t : A}\right)$$

and we define the desired elaboration candidate

$$\ell_{jb}(\Theta', \Gamma', \mathscr{D}_{\mathscr{J}}) = t' \equiv t' : A'.$$

Cases TT-Ty-Tran *and* TT-Tm-Tran: We just take a look at the term equality case. The type equality case is similar. The derivation $\mathscr{D}_{\mathcal{J}}$ ends with

$$\frac{\dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash s \equiv t : A} \qquad \dfrac{\mathscr{D}_2}{\Theta;\Gamma \vdash t \equiv u : A}}{\Theta;\Gamma \vdash s \equiv u : A}$$

By induction hypothesis with $\Theta';\Gamma'$ we get

$$s' \equiv t' : A' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_1}{\Theta;\Gamma \vdash s \equiv t : A}\right)$$
$$t'' \equiv u' : A'' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_2}{\Theta;\Gamma \vdash t \equiv u : A}\right)$$

We again have more than one option to choose the type of the equality, but all options turn out to be judgementally equal by Corollary 10.1.6. Furthermore in the definition of the elaboration candidate we do not need to use the intermediate term t' (or t'').

and we define

$$\ell_{jb}(\Theta', \Gamma', \mathscr{D}_{\mathcal{J}}) = s' \equiv u' : A'$$

which is indeed an appropriate elaboration candidate.

Cases TT-Conv-Tm, TT-Conv-Eq: Both cases proceed in a similar fashion, we consider the case for TT-Conv-Eq. The derivation $\mathscr{D}_{\mathcal{J}}$ ends with

$$\frac{\dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash s \equiv t : A} \qquad \dfrac{\mathscr{D}_2}{\Theta;\Gamma \vdash A \equiv B}}{\Theta;\Gamma \vdash s \equiv t : B}$$

By induction hypothesis we obtain elaboration candidates

$$s' \equiv t' : A' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_1}{\Theta;\Gamma \vdash s \equiv t : A}\right)$$
$$A'' \equiv B' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_2}{\Theta;\Gamma \vdash A \equiv B}\right)$$

and we define

$$\ell_{jb}(\Theta', \Gamma', \mathscr{D}_{\mathcal{J}}) = s' \equiv t' : B'$$

which is the desired elaboration candidate.

Cases TT-Meta, TT-Meta-Congr: Both cases proceed in the same way.

We take a look at TT-Meta-Congr. The derivation $\mathscr{D}_\mathcal{J}$ ends with

$$\dfrac{\begin{array}{ll}\Theta(\mathsf{M}_k) = \{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\,\mathscr{b} & \\ \dfrac{\mathscr{D}_{j1}}{\Theta;\Gamma \vdash s_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}]} & \text{for } j = 1,\ldots,m \\ \dfrac{\mathscr{D}_{j2}}{\Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]} & \text{for } j = 1,\ldots,m \\ \dfrac{\mathscr{D}_{j3}}{\Theta;\Gamma \vdash s_j \equiv t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}]} & \text{for } j = 1,\ldots,m \\ \dfrac{\mathscr{D}_4}{\Theta;\Gamma \vdash C[\vec{s}/\vec{x}] \equiv C[\vec{t}/\vec{x}]} & \text{if } \mathscr{b} = (\square : C)\end{array}}{\Theta;\Gamma \vdash (\mathscr{b}[\vec{s}/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}) \equiv \mathsf{M}_k(\vec{t})}}$$

Since M_k is also in the domain of Θ', let $\Theta'(\mathsf{M}_k) = \{x_1{:}A'_1\}\cdots\{x_m{:}A'_m\}\mathscr{b}'$. By induction hypothesis we get

$$s'_j : A^1_j = \ell_{jb}\left(\Theta', \Gamma', \dfrac{\mathscr{D}_{j1}}{\Theta;\Gamma \vdash s_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}]}\right) \quad \text{for } j = 1,\ldots,m$$

$$t'_j : A^2_j = \ell_{jb}\left(\Theta', \Gamma', \dfrac{\mathscr{D}_{j2}}{\Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]}\right) \quad \text{for } j = 1,\ldots,m$$

$$s''_j \equiv t''_j : A^3_j = \ell_{jb}\left(\Theta', \Gamma', \dfrac{\mathscr{D}_{j3}}{\Theta;\Gamma \vdash s_j \equiv t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}]}\right) \quad \text{for } j = 1,\ldots,m$$

We define

$$\ell_{jb}(\Theta', \Gamma', \mathscr{D}_\mathcal{J}) = (\mathscr{b}'[\vec{s}'/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}') \equiv \mathsf{M}_k(\vec{t}')}$$

We only use $\vec{s}'$ and $\vec{t}'$ in the definition and not $\vec{s}''$ or $\vec{t}''$ which turn out to be judgementally equal to $\vec{s}'$ and $\vec{t}'$ respectively.

and we check that this is a good elaboration candidate:

$$\begin{aligned} & r_*((\mathscr{b}'[\vec{s}'/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}') \equiv \mathsf{M}_k(\vec{t}')}) \\ = {} & (r_*\mathscr{b}'[(r_*\vec{s}')/\vec{x}])\boxed{\mathsf{M}_k(r_*\vec{s}') \equiv \mathsf{M}_k(r_*\vec{t}')} \\ = {} & (\mathscr{b}[\vec{s}'/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}) \equiv \mathsf{M}_k(\vec{t})}. \end{aligned}$$

Case Specific rule: Here we have to pay attention to the fragment in which the judgement happens: the derivation $\mathscr{D}_\mathcal{J}$ is in an elaborative fragment $\mathscr{T}^{\mathsf{fr}}$ of $\mathscr{T}$. Suppose the derivation $\mathscr{D}_\mathcal{J}$ ends with a specific rule $R = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n] \Longrightarrow \mathcal{J}$ with a derivable (in $\mathscr{T}^{\mathsf{fr}}$) instantiation $I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$ over $\Theta;\Gamma$. By (1) we have an elaboration candidate R' for R. Since I is derivable, we have derivations

$$\dfrac{\mathscr{D}_i}{\Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i}} \qquad \text{for } i = 1,\ldots,n.$$

By induction hypothesis we have elaboration candidates over $\Theta';\Gamma'$

for those derivable judgements:

$$\mathscr{B}'_i\boxed{e'_i} = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_i}{\Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i}}\right)$$

for $i = 1, \ldots, n$. We define an instantiation

$$I' = \langle \mathsf{M}_1 \mapsto e'_1, \ldots, \mathsf{M}_n \mapsto e'_n \rangle$$

and we define the elaboration candidate for $\mathscr{D}$ to be

$$\ell_{jb}(\Theta, \Gamma, \mathscr{D}_{\mathcal{J}}) = I'_* R'.$$

Since the equations $r_*(R') = R$ and $r_*(I') = I$ hold, we get the desired equation by Lemma 8.3.3. □

From now on we write partial maps ℓ_m, ℓ_v, ℓ_b and ℓ_j from the proof of Lemma 10.2.3 as ℓ and call it the elaboration map, even though we still need to prove it preserves derivability.

Note that ℓ_{jb} is not a part of elaboration map, but an auxiliary map to prove preservation of derivability.

10.2.2. Specific rules of elaboration $\mathcal{S}$

When defining the rules of the standard type theory $\mathcal{S}$ we need to make sure the type theory is finitary, so we need a well-founded order on the rules. To be able to construct such an order we build the theory $\mathcal{S}$ inductively on the order of the rules in $\mathcal{T} = (R_i)_{i \in \mathrm{I}}$. For every fragment $\mathcal{T}^i$ we define an elaboration $\mathcal{S}^i$, where its signature contains only the symbols for the specific object rules in $\mathcal{T}^i$. The syntactic part of the retrogression transformation (and elaboration map) for $\mathcal{S}^i$ is just a restriction of the defined r (and ℓ) for $\mathcal{S}$. However, we still write r and ℓ when the restriction can be inferred.

Now suppose $i \in \mathrm{I}$ and for the fragment $\mathcal{T}^i$ induced by the index set

$$\downarrow i = \{j \in \mathrm{I} \mid j \sqsubset i\}$$

we have already constructed an elaboration $\mathcal{S}^i$. Let $R_i = \Theta \Longrightarrow \mathcal{b}\boxed{e}$ be the specific rule in $\mathcal{T}$ and let $\mathcal{T}^{i+}$ be the fragment induced by

$$\{j \in \mathrm{I} \mid j \sqsubseteq i\}.$$

Since $\mathcal{T}$ is finitary, the fragment $\mathcal{T}^i$ derives

$$\vdash_{\mathcal{T}^i} \Theta\ \mathsf{mctx} \qquad \text{and} \qquad \Theta; [] \vdash_{\mathcal{T}^i} \mathcal{b}$$

with some derivations $\mathscr{D}_\Theta$ and $\mathscr{D}_{\mathcal{b}}$.

We define $\mathcal{S}^i_0$ in the following way:

- If R_i is a specific object rule in $\mathcal{T}$, then we add $\mathsf{S}_{\left(i, \Theta \Longrightarrow \mathcal{b}\boxed{e}\right)}$ to the signature of $\mathcal{S}^i_0$ and extend the syntactic part of the retrogression transformation. We pose the symbol rule

$$R'_i = \left(\ell_m(\mathscr{D}_\Theta) \Longrightarrow \ell_b(\ell_m(\mathscr{D}_\Theta), [], \mathscr{D}_{\mathcal{b}})\boxed{\mathsf{S}_{\left(i, \Theta \Longrightarrow \mathcal{b}\boxed{e}\right)}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})}\right)$$

for $\mathsf{M}_1, \ldots, \mathsf{M}_n$ metavariables from Θ.

- If R_i is a specific equality rule in $\mathcal{T}$, then we pose the specific equality rule

$$R_i' = (\ell_m(\mathcal{D}_\Theta) \Longrightarrow \ell_b(\ell_m(\mathcal{D}_\Theta), [], \mathcal{D}_b)\boxed{\star}).$$

To order the rules in $\mathcal{S}_0^i$, let $<_i$ be the order the index set J^i for $\mathcal{S}^i$. We extend the order $<_i$ by adding the new rule on top, i.e. adding $(i, 0, R_i')$ to the index set on top and mapping it to the new rule. With this ordering $\mathcal{S}_0^i$ is finitary, because $\mathcal{S}^i$ is finitary and the metacontext and the boundary of the new rule are derivable in $\mathcal{S}^i$ because ℓ is an elaboration map that preserves derivability. The theory $\mathcal{S}_0^i$ is also standard, because $\mathcal{S}^i$ is standard and if we added a symbol we posed precisely one symbol rule for it.

While $\mathcal{S}_0^i$ is standard, it is not yet an elaboration, because the retrogression transformation r as defined is not necessarily conservative. The crux of the problem is that there may be additional equalities that hold in $\mathcal{T}^{i+}$ but their counterparts in $\mathcal{S}_0^i$ cannot be derived as we can see from the following example.

Example 10.2.4 Suppose we have a finitary type theory like in Example 4.4.4 with specific rules

$$[] \Longrightarrow \mathsf{N}\ \mathsf{type}, \qquad [] \Longrightarrow \mathsf{O} : \mathsf{N}, \qquad \mathsf{n}{:}(\square : \mathsf{N}) \Longrightarrow \mathsf{S}(\mathsf{S}(\mathsf{n})) : \mathsf{N}$$

to which we add another specific rule

$$\mathsf{n}{:}(\square : \mathsf{N}) \Longrightarrow \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{n})))) : \mathsf{N}.$$

Suppose also we have an elaboration $\mathcal{S}^3$ for the fragment with the first three rules and for $\mathcal{S}_0^3$ we posit the symbol rule

$$\mathsf{n}{:}(\square : \mathsf{S}_\mathsf{N}) \Longrightarrow \mathsf{S}_{\mathsf{SSSS}}(\mathsf{n}) : \mathsf{S}_\mathsf{N}$$

where S_N is short for $\mathsf{S}_{(1,[]\Longrightarrow \mathsf{N}\ \mathsf{type})}$ and similarly for other symbols. In $\mathcal{S}_0^3$ the boundary

$$[\mathsf{n}{:}(\square : \mathsf{S}_\mathsf{N})]; [] \vdash_{\mathcal{S}_0^3} \mathsf{S}_{\mathsf{SS}}(\mathsf{S}_{\mathsf{SS}}(\mathsf{n})) \equiv \mathsf{S}_{\mathsf{SSSS}}(\mathsf{n}) : \mathsf{S}_\mathsf{N}\ \mathsf{by}\ \square \tag{10.1}$$

is derivable. We map it with the retrogression transformation and get the derivable judgement

$$[\mathsf{n}{:}(\square : \mathsf{N})]; [] \vdash \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{n}))))) \equiv \mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{S}(\mathsf{n}))))) : \mathsf{N}\ \mathsf{by}\ \star$$

which is derived by TT-Tm-Refl. However the judgement arising from the boundary (10.1) is not derivable in $\mathcal{S}_0^3$ with the given rules, so the retrogression transformation is not conservative.

The theory $\mathcal{S}^3$ is obtained by the construction of the elaboration theorem, so we have a symbol in the signature for every specific object rule of the theory we started with.

To ensure r is conservative we add equalities as specific rules. We start with $\mathcal{S}_0^i$ (with order $\prec_0^i$ on index set J_0^i) and inductively generate $\mathcal{S}_{k+1}^i$ with ordering $\prec_{k+1}^i$ on index set J_{k+1}^i in the following way: the specific rules of $\mathcal{S}_{k+1}^i$ are those of $\mathcal{S}_k^i$ and for every strongly derivable

equational rule-boundary $\Theta \Longrightarrow \mathcal{b}$ in $\mathcal{S}^i_k$ if

$$r_*(\Theta); [] \vdash_{\mathcal{T}^{i+}} r_*(\mathcal{b}\boxed{\star})$$

is derivable in the fragment $\mathcal{T}^{i+}$ then

$$\Theta \Longrightarrow \mathcal{b}\boxed{\star}$$

is a specific equality rule in $\mathcal{S}^i_{k+1}$. There are countably many new specific rules. We index these equality rules by the triples $(i, k, \Theta \Longrightarrow \mathcal{b}\boxed{\star})$, order them in a flat order and place them on top of $(\mathrm{J}^i_k, \prec^i_k)$ to get the well-founded ordering $(\mathrm{J}^i_{k+1}, \prec^i_{k+1})$. The theory $\mathcal{S}^i_{k+1}$ is standard by definition.

We can now define the type theory $\mathcal{S}^{i+}$, which has the signature equal to the signature of $\mathcal{S}^i_0$ and the specific rules the union of the specific rules of $\mathcal{S}^i_k$ for $k \in \mathbb{N}$. Specifically, the rules are indexed by

$$\mathrm{J}^{i+1} = \cup_{k\in\mathbb{N}} \mathrm{J}^i_k$$

and the order $\prec^{i+1}$ is induced by the union of the well-founded orders $\prec^i_k$ for $k \in \mathbb{N}$. This is a well-founded order by Lemma 4.4.2.

The type theory $\mathcal{S}^{i+}$ is finitary: let $R = \Theta \Longrightarrow \mathcal{b}\boxed{e}$ be a specific rule of $\mathcal{S}^{i+}$. Then there exist $k \in \mathbb{N}$ such that R is a specific rule in $\mathcal{S}^i_k$. Since $\mathcal{S}^i_k$ is finitary, the judgements $\vdash \Theta$ mctx and $\Theta; [] \vdash \mathcal{b}$ are derivable in the fragment of smaller rules according to the well-founded order. Because these derivations are embedded in the appropriate fragment of $\mathcal{S}^{i+}$, the rule R is finitary. It is also easy to see that $\mathcal{S}^{i+}$ is standard, because all the specific object rules are symbol rules, which are already present in $\mathcal{S}^i_0$.

Once $\mathcal{S}^{i+}$ is proven to be an elaboration, we can construct the elaboration $\mathcal{S}$ as follows:

- The signature of $\mathcal{S}$ is already given in Subsection 'Syntactic part of elaboration $\mathcal{S}$', as well as the (syntactic part of) retrogression transformation and elaboration map.
- The index set for the rules is the dependent sum $\mathrm{I}_{\mathcal{S}} = \Sigma_{(i:\mathbb{I})}\mathrm{J}^i$.
- The ordering $<$ on the index set $\mathrm{I}_{\mathcal{S}}$ is the lexicographic order: $(i, j) < (i', j')$ if, and only if $i \sqsubset i'$ or $(i = i'$ and $j \prec^i j')$.
- Rules are mapped accordingly: $(i, j) \in \mathrm{I}_{\mathcal{S}}$ is mapped to the rule in $\mathcal{S}^i$ indexed by j.

With this definition the ordering of the rules of $\mathcal{S}$ is well-founded, as the lexicographic order on a dependent sum of well-founded orders is well founded, see [140]. The theory $\mathcal{S}$ is finitary: a rule indexed by (i, j) is already finitary in the fragment $\mathcal{S}^i$. The theory $\mathcal{S}$ is also standard by construction. The retrogression transformation r is indeed a type-theoretic transformation: for a specific rule at index (i, j) we have a derivation provided by the retrogression map for $\mathcal{S}^i$. Conservativity of r follows from a similar argument[4], and so does the fact that ℓ preserves derivability.

[140]: team (2021), *Agda standard library*

4: Derivations in $\mathcal{T}$ are finite, so they happen in a fragment of $\mathcal{T}$ that is elaborated with a relevant fragment $\mathcal{S}^i$.

It remains to prove that $\mathcal{S}^{i+}$ is an elaboration with the retrogression transformation r and elaboration map ℓ. Specifically, we need to prove

that r is a conservative (Corollary 10.2.8) type-theoretic transformation (Corollary 10.2.7) and that ℓ preserves derivability (Lemma 10.2.6).

To handle equalities in $\mathcal{S}^{i+}$, we prove the following lemma.

Lemma 10.2.5 If $\Xi; \Delta \vdash_{\mathcal{S}^{i+}} \mathcal{b}$ is a strongly derivable equality boundary in $\mathcal{S}^{i+}$ such that

$$r_*(\Xi); r_*(\Delta) \vdash_{\mathcal{T}^{i+}} (r_*(\mathcal{b}))\boxed{\star}$$

is derivable, then

$$\Xi; \Delta \vdash_{\mathcal{S}^{i+}} \mathcal{b}\boxed{\star}$$

is (strongly) derivable in $\mathcal{S}^{i+}$.

Proof. Using Proposition 4.3.6 we obtain a derivation of

$$(\Xi, \Delta); [] \vdash_{\mathcal{S}^{i+}} \mathcal{b}. \tag{10.2}$$

Because the derivation is finite, there exists $k \in \mathbb{N}$ such that (10.2) is derivable in $\mathcal{S}^i_k$. Since the judgement

$$(r_*(\Xi), r_*(\Delta)); [] \vdash_{\mathcal{T}^{i+}} r_*(\mathcal{b}\boxed{\star})$$

is just a promoted version of the derivable judgement

$$r_*(\Xi); r_*(\Delta) \vdash_{\mathcal{T}^{i+}} r_*(\mathcal{b}\boxed{\star}),$$

by definition of $\mathcal{S}^i_{k+1}$ we get that the promoted judgement

$$(\Xi, \Delta); [] \vdash_{\mathcal{S}^i_{k+1}} \mathcal{b}\boxed{\star}$$

is derivable in $\mathcal{S}^i_{k+1}$. Using Proposition 4.3.6 we get derivability of

$$\Xi; \Delta \vdash_{\mathcal{S}^i_{k+1}} \mathcal{b}\boxed{\star}$$

and because the theory $\mathcal{S}^i_{k+1}$ is embedded in $\mathcal{S}^{i+}$ we get the desired derivability of

$$\Xi; \Delta \vdash_{\mathcal{S}^{i+}} \mathcal{b}\boxed{\star}.$$

□

Lemma 10.2.6 Elaboration preserves derivability:

1. If $\mathcal{D}_\Theta$ is a derivation of $\vdash_{\mathcal{T}^{i+}} \Theta$ mctx, then

   $$\vdash_{\mathcal{S}^{i+}} \ell(\mathcal{D}_\Theta)\ \text{mctx}$$

   is derivable in $\mathcal{S}^{i+}$.
2. If Θ' is a derivable elaboration candidate for Θ and $\mathcal{D}_\Gamma$ is a derivation of $\Theta \vdash_{\mathcal{T}^{i+}} \Gamma$ vctx, then

   $$\Theta' \vdash_{\mathcal{S}^{i+}} \ell(\Theta', \mathcal{D}_\Gamma)\ \text{vctx}$$

   is derivable in $\mathcal{S}^{i+}$.
3. If Θ', Γ' are derivable elaboration candidates for Θ, Γ and $\mathcal{D}_{\mathcal{B}}$

is a derivation of $\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} \mathscr{B}$, then

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \ell(\Theta',\Gamma',\mathscr{D}_{\mathscr{B}})$$

is derivable in $\mathcal{S}^{i+}$.

4. If Θ', Γ', $\mathscr{B}'$ are derivable elaboration candidates for Θ, Γ, $\mathscr{B}$ respectively and $\mathscr{D}_{\mathscr{J}}$ is a derivation of $\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} \mathscr{B}[\mathscr{e}]$, then

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \ell(\Theta',\Gamma',\mathscr{B}',\mathscr{D}_{\mathscr{J}})$$

is derivable in $\mathcal{S}^{i+}$.

5. If Θ', Γ' are derivable elaboration candidates for Θ, Γ and $\mathscr{D}_{\mathscr{J}}$ is a derivation of $\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} \mathscr{J}$, then

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \ell_{jb}(\Theta',\Gamma',\mathscr{D}_{\mathscr{J}})$$

is derivable in $\mathcal{S}^{i+}$.

Proof. By induction on derivations.

Part (1):

Case MCtx-Empty: We use the rule MCtx-Empty again to derive the metacontext $\vdash_{\mathcal{S}^{i+}} [] \ \mathsf{mctx}$.

Case MCtx-Extend: The derivation ends with

$$\frac{\dfrac{\mathscr{D}_1}{\vdash_{\mathcal{T}^{i+}} \Xi \ \mathsf{mctx}} \qquad \dfrac{\mathscr{D}_2}{\Xi;[] \vdash_{\mathcal{T}^{i+}} \mathscr{B}} \qquad \mathsf{M} \notin |\Xi|}{\vdash_{\mathcal{T}^{i+}} \langle \Xi, \mathsf{M}{:}\mathscr{B} \rangle \ \mathsf{mctx}}$$

By induction hypothesis on the first premise

$$\vdash_{\mathcal{S}^{i+}} \Xi' \ \mathsf{mctx} \tag{10.3}$$

is derivable for

$$\Xi' = \ell\left(\frac{\mathscr{D}_1}{\vdash_{\mathcal{T}^{i+}} \Xi \ \mathsf{mctx}}\right)$$

We can then apply induction hypothesis on the second premise with Ξ' to get a derivation of

$$\Xi';[] \vdash_{\mathcal{S}^{i+}} \mathscr{B}' \tag{10.4}$$

for

$$\mathscr{B}' = \ell\left(\Xi', [], \frac{\mathscr{D}_2}{\Xi;[] \vdash_{\mathcal{T}^{i+}} \mathscr{B}}\right).$$

We now combine (10.3) and (10.4) with MCtx-Extend to get a derivation of

$$\vdash_{\mathcal{S}^{i+}} \Theta' \ \mathsf{mctx}$$

with $\Theta' = \langle \Xi', \mathsf{M}{:}\mathscr{B}' \rangle$.

Part (2):

Case VCtx-Empty: Similarly to the case MCtx-Empty we derive the empty variable context using the rule VCtx-Empty.

Case VCtx-Extend: The derivation of the variable context ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta \vdash_{\mathcal{T}^{i+}} \Gamma \text{ vctx}} \qquad \dfrac{\mathscr{D}_2}{\Theta, \Gamma \vdash_{\mathcal{T}^{i+}} A \text{ type}} \qquad \mathsf{a} \notin |\Gamma|}{\Theta \vdash_{\mathcal{T}^{i+}} \langle \Gamma, \mathsf{a}{:}A \rangle \text{ vctx}}$$

By induction hypothesis on the first premise we get a derivation of

$$\Theta' \vdash_{\mathcal{S}^{i+}} \Gamma' \text{ vctx} \tag{10.5}$$

for

$$\Gamma' = \ell \left(\Theta', \dfrac{\mathscr{D}_1}{\Theta \vdash_{\mathcal{T}^{i+}} \Gamma \text{ vctx}} \right).$$

We can now use the induction hypothesis on the second premise (given that Θ' and Γ' are derivable elaboration candidates and $\square$ type is derivable everywhere) to get a derivation of

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} A' \text{ type} \tag{10.6}$$

for A' the elaboration of type A, obtained by the appropriate elaboration map. We can combine (10.5) and (10.6) with VCtx-Extend to get the desired derivation of

$$\Theta' \vdash_{\mathcal{S}^{i+}} \ell(\Theta', \mathscr{D}) \text{ vctx}.$$

Part (3):

Cases TT-Bdry-Ty, TT-Bdry-Tm, TT-Bdry-EqTy *or* TT-Bdry-EqTm: It is a direct use of the induction hypothesis. Let us take a look at TT-Bdry-EqTm as an example. The derivation $\mathscr{D}_{\mathscr{B}}$ ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} A \text{ type}} \qquad \dfrac{\mathscr{D}_2}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} s : A} \qquad \dfrac{\mathscr{D}_3}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} t : A}}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} s \equiv t : A \text{ by } \square}$$

By induction hypothesis we get for

$$A' \text{ type} = \ell \left(\Theta', \Gamma', \square \text{ type}, \dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} A \text{ type}} \right)$$

$$s' : A' = \ell \left(\Theta', \Gamma', \square : A', \dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} s : A} \right)$$

$$t' : A' = = \ell \left(\Theta', \Gamma', \square : A', \dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} t : A} \right)$$

derivations of

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} A' \text{ type} \qquad \Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} s' : A' \qquad \Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} t' : A'$$

We then use TT-Bdry-EqTm to get the derivation of the desired result

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} s' \equiv t' : A' \text{ by } \square.$$

Case TT-Bdry-Abstr: The derivation of the boundary ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} A\ \mathsf{type}} \quad \mathsf{a} \notin |\Gamma| \quad \dfrac{\mathscr{D}_2}{\Theta;\Gamma,\mathsf{a}{:}A \vdash_{\mathcal{T}^{i+}} \mathscr{B}'[\mathsf{a}/x]}}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} \{x{:}A\}\ \mathscr{B}'}$$

By induction hypothesis on the first premise with

$$A'\ \mathsf{type} = \ell\left(\Theta', \Gamma', \square\ \mathsf{type}, \dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} A\ \mathsf{type}}\right) \tag{10.7}$$

we get a derivation of

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} A'\ \mathsf{type}.$$

Using VCtx-Extend on (10.7) and the derivation of Γ' we derive

$$\Theta' \vdash_{\mathcal{S}^{i+}} \langle \Gamma', \mathsf{a}{:}A' \rangle\ \mathsf{vctx}.$$

Now we can use induction hypothesis on the third premise with Θ' and $\langle \Gamma', \mathsf{a}{:}A' \rangle$ to get a derivation of

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \mathscr{B}''$$

for

$$\mathscr{B}'' = \ell\left(\Theta', \langle \Gamma', \mathsf{a}{:}A' \rangle, \dfrac{\mathscr{D}_2}{\Theta;\Gamma,\mathsf{a}{:}A \vdash_{\mathcal{T}^{i+}} \mathscr{B}'[\mathsf{a}/x]}\right).$$

We can now derive

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \{x{:}A'\}\mathscr{B}''[x/\mathsf{a}]$$

by using TT-Bdry-Abstr.

Part (4):

Case TT-Abstr: The derivation $\mathscr{D}_{\mathcal{J}}$ ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} A\ \mathsf{type}} \quad \mathsf{a} \notin |\Gamma| \quad \dfrac{\mathscr{D}_2}{\Theta;\Gamma,\mathsf{a}{:}A \vdash_{\mathcal{T}^{i+}} (\mathscr{B}''[\mathsf{a}/x])\boxed{e'[\mathsf{a}/x]}}}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} \{x{:}A\}\ \mathscr{B}''\boxed{e'}}$$

The boundary $\mathscr{B}'$ is then a derivable elaboration candidate for $\{x{:}A\}\ \mathscr{B}''$ and because r preserves forms of boundaries $\mathscr{B}' = \{x{:}A'\}\ \mathscr{B}'''$, where $\mathscr{B}'''$ is a derivable[5] elaboration candidate for $\mathscr{C}''[\mathsf{a}/x]$ in context $\Theta';\Gamma',\mathsf{a}{:}A'$. Because $\mathscr{B}'$ is strongly derivable we have a derivation[6] of $\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} A'\ \mathsf{type}$, so $\langle \Gamma', \mathsf{a}{:}A' \rangle$ is a derivable variable context. By induction on the second premise we get a derivation of

$$\Theta';\langle \Gamma', \mathsf{a}{:}A' \rangle \vdash_{\mathcal{S}^{i+}} \mathscr{B}'''\boxed{e''}$$

5: We just use inversion with TT-Bdry-Abstr.

6: Again by inversion with TT-Bdry-Abstr.

for

$$\mathscr{B}'''[\boxed{e''}] = \ell_j\left(\Theta', \langle\Gamma', \mathsf{a}{:}A'\rangle, \mathscr{B}''', \frac{\mathscr{D}_2}{\Theta; \Gamma, \mathsf{a}{:}A \vdash (\mathscr{B}''[\mathsf{a}/x])[e'[\mathsf{a}/x]]}\right)$$

We conclude the proof by an application of TT-Abstr.

Case TT-Var: We again use the rule TT-Var to derive

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} \mathsf{a} : \Gamma'(\mathsf{a}).$$

Cases TT-Ty-Refl, TT-Tm-Refl,TT-Ty-Sym, TT-Tm-Sym, TT-Ty-Tran, TT-Tm-Tran, TT-Meta-Congr, TT-Conv-Eq: Since $\mathscr{B}'$ is an equality boundary with $r_*(\mathscr{B}') = \mathscr{B}$ and $\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} \mathscr{B}[\boxed{\star}]$ is derivable we obtain derivability by Lemma 10.2.5.

Cases TT-Conv-Tm: The derivation ends with

$$\frac{\dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} t : A} \qquad \dfrac{\mathscr{D}_2}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} A \equiv B}}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} t : B}$$

Using induction hypothesis with Part (5) on the first premise we get a derivation of

$$t' : A' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} t : A}\right)$$

and by induction hypothesis on the second premise we obtain a derivation of $\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} A' \equiv B'$ where $\mathscr{B}' = \square : B'$. We conclude the desired derivation using TT-Conv-Tm.

Cases TT-Meta: The derivation $\mathscr{D}_{\mathcal{J}}$ ends with

$$\dfrac{\Theta(\mathsf{M}_k) = \{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\ \mathscr{C} \qquad \dfrac{\mathscr{D}_j}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]}\ \text{for } j = 1, \ldots, m \qquad \dfrac{\mathscr{D}_{\mathscr{C}}}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} \mathscr{C}[\vec{t}/\vec{x}]}}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} (\mathscr{C}[\vec{t}/\vec{x}])[\boxed{\mathsf{M}_k(\vec{t})}]}$$

Since M_k is also in the domain of Θ', let

$$\Theta'(\mathsf{M}_k) = \{x_1{:}A'_1\}\cdots\{x_m{:}A'_m\}\mathscr{C}'$$

By induction hypothesis we get derivations of

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} t'_j : A'_j[\vec{t}'_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m$$

where

$$t'_j : A'_j[\vec{t}'_{(j)}/\vec{x}_{(j)}] = \ell\left(\Theta', \Gamma', \square : A'_j[\vec{t}'_{(j)}/\vec{x}_{(j)}], \frac{\mathscr{D}_j}{\Theta; \Gamma \vdash_{\mathcal{T}^{i+}} t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]}\right)$$

Because the strongly derivable boundaries $\mathscr{B}'$ and $\mathscr{b}'[\vec{t}'/\vec{x}]$ both get mapped to $\mathscr{b}[\vec{t}/\vec{x}]$ they are equal by Lemma 10.2.5. Using TT-Meta followed by TT-Conv-Tm we obtain a derivation of the desired judgement

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \mathscr{B}'\boxed{\mathsf{M}_k(\vec{t}')}.$$

Case Specific rule of $\mathcal{T}^{i+}$: Suppose the derivation $\mathscr{D}_{\mathcal{J}}$ ends with an application of the specific rule

$$R = \Xi \Longrightarrow \mathscr{b}\boxed{e}$$

for

$$\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$$

with a derivable instantiation $I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$. Let the elaboration candidate for R be

$$R' = [\mathsf{M}_1{:}\mathscr{B}'_1, \ldots, \mathsf{M}_n{:}\mathscr{B}'_n] \Longrightarrow \mathscr{b}'\boxed{e'}.$$

By induction hypothesis we get derivations of

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \mathscr{B}'_i\boxed{e'_i} \qquad \text{for } i = 1, \ldots, n$$

with

$$I'_{(i)*}\mathscr{B}'_i\boxed{e'_i} = \ell\left(\Theta', \Gamma', I'_{(i)*}\mathscr{B}'_i, \frac{\mathscr{D}_i}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} (I_{(i)*}\mathscr{B}_i)\boxed{e_i}}\right)$$

giving us the derivability (in theory $\mathcal{S}^{i+}$) of the instantiation

$$I' = \langle \mathsf{M}_1 \mapsto e'_1, \ldots, \mathsf{M}_n \mapsto e'_n \rangle.$$

By Theorem 5.1.4 with the rule R' and the derivable instantiation I' we derive the judgement $\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} (I'_*\mathscr{b}')\boxed{I'_*e'}$. Because r maps the boundaries $I'_*\mathscr{b}'$ and $\mathscr{B}'$ to the same boundary, they are equal by Lemma 10.2.5, and we can thus convert along this equality to derive the desired judgement. *Part (5):*

Case TT-Abstr: The proof is similar to the case TT-Bdry-Abstr.

Case TT-Var: We again use the rule TT-Var to derive

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \mathsf{a} : \Gamma'(\mathsf{a}).$$

Cases TT-Ty-Refl, TT-Tm-Refl,TT-Ty-Sym, TT-Tm-Sym: It is a direct application of the induction hypothesis. We take a look at the TT-Tm-Refl case. The derivation $\mathscr{D}_{\mathcal{J}}$ ends with

$$\dfrac{\dfrac{\mathscr{D}_1}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} t : A}}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} t \equiv t : A}$$

By induction hypothesis we get a derivation of

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} t' : A'$$

for

$$t' : A' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} t : A}\right).$$

We again use TT-Tm-Refl to derive

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} t' \equiv t' : A'.$$

Cases TT-Ty-Tran *and* TT-Tm-Tran: We just take a look at the term equality case. The type equality case is similar. The derivation $\mathscr{D}_{\mathcal{J}}$ ends with

$$\frac{\dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} s \equiv t : A} \qquad \dfrac{\mathscr{D}_2}{\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} t \equiv u : A}}{\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} s \equiv u : A}$$

By induction hypothesis we get derivations of

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} s' \equiv t' : A'$$
$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} t'' \equiv u' : A''$$

for

$$s' \equiv t' : A' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} s \equiv t : A}\right)$$
$$t'' \equiv u' : A'' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_2}{\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} t \equiv u : A}\right)$$

By Theorem 5.1.6 we have that $\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} A'$ and $\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} A''$ are derivable. Because the context $\Theta'; \Gamma'$ is derivable, the boundary

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} A' \equiv A'' \text{ by } \square \tag{10.8}$$

is strongly derivable and it is mapped by the retrogression transformation r to the boundary $\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} A \equiv A$ by $\square$, which can be filled with the head $\star$ using the rule TT-Ty-Refl. By Lemma 10.2.5 the equation pertaining to (10.8) is derivable. We can use this equation to convert t'' and u' to type A' and in a similar fashion as before get a derivation of

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} t' \equiv t'' : A'.$$

Using TT-Tm-Tran we can string the equalities together to derive the desired equality

$$\Theta'; \Gamma' \vdash_{\mathcal{S}^{i+}} s' \equiv u' : A'.$$

Cases TT-Conv-Tm,TT-Conv-Eq: The proof is similar in both cases. We take a look at TT-Conv-Tm. The derivation ends with

$$\frac{\dfrac{\mathscr{D}_1}{\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} t : A} \qquad \dfrac{\mathscr{D}_2}{\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} A \equiv B}}{\Theta; \Gamma \vdash_{\mathcal{J}^{i+}} t : B}$$

By induction hypothesis we get derivations of

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} t' : A' \qquad \Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} A'' \equiv B'$$

where

$$t' : A' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_1}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} t : A}\right)$$
$$A'' \equiv B' = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_2}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} A \equiv B}\right)$$

The use of the Lemma 10.2.5 follows the same steps as in the previous cases.

Using Lemma 10.2.5 we obtain a derivation of

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} A' \equiv A''.$$

We combine the equalities with TT-Ty-Tran and apply TT-Conv-Tm to derive the desired result

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} t' : B'.$$

Cases :TT-Meta, TT-Meta-Congr Both cases proceed similarly, we look at the case for TT-Meta. The derivation $\mathscr{D}_{\mathcal{J}}$ ends with

$$\dfrac{\Theta(\mathsf{M}_k) = \{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\ \mathcal{b} \qquad \dfrac{\mathscr{D}_j}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]}\ \text{for } j = 1,\dots,m \qquad \dfrac{\mathscr{D}_{\mathcal{b}}}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} \mathcal{b}[\vec{t}/\vec{x}]}}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} (\mathcal{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_k(\vec{t})}}$$

Since M_k is also in the domain of Θ', let

$$\Theta'(\mathsf{M}_k) = \{x_1{:}A'_1\}\cdots\{x_m{:}A'_m\}\mathcal{b}'$$

By induction hypothesis we get derivations of

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} t'_j : A''_j \quad \text{for } j = 1,\dots,m$$
$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \mathcal{b}''$$

where

$$t'_j : A''_j = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_j}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]}\right)$$

Since for $j = 1,\dots,m$ types A''_j and $A'_j[\vec{t'}_{(j)}/\vec{x}_{(j)}]$ are derivable (in a derivable context) and both get mapped to $A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]$ by r, the Lemma 10.2.5 provides us with a derivation of

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} A''_j \equiv A'_j[\vec{t'}_{(j)}/\vec{x}_{(j)}]$$

so we can convert t'_j accordingly. Because Θ' is a derivable elabora-

tion candidate for Θ, the boundary $\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \mathcal{b}'$ is also derivable and we can apply TT-META to get a derivation of the desired judgement

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} (\mathcal{b}'[\vec{t}'/\vec{x}'])\boxed{\mathsf{M}_k(\vec{t}')}.$$

Case Specific rule of $\mathcal{T}^{i+}$: Suppose the derivation $\mathscr{D}_{\mathcal{J}}$ ends with an application of the specific rule

$$\Xi \Longrightarrow \mathcal{b}\boxed{e}$$

for

$$\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$$

with a derivable instantiation $I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$. By induction hypothesis we get derivations of

$$\Theta';\Gamma' \vdash_{\mathcal{S}^{i+}} \mathscr{B}'_i\boxed{e'_i} \qquad \text{for } i = 1, \ldots, n$$

with

$$\mathscr{B}'_i\boxed{e'_i} = \ell_{jb}\left(\Theta', \Gamma', \frac{\mathscr{D}_i}{\Theta;\Gamma \vdash_{\mathcal{T}^{i+}} (I_{(i)*}\mathscr{B}_i)\boxed{e_i}}\right)$$

giving us the derivability (in theory $\mathcal{S}^{i+}$) of the instantiation

$$I' = \langle \mathsf{M}_1 \mapsto e'_1, \ldots, \mathsf{M}_n \mapsto e'_n \rangle.$$

By Theorem 5.1.4 we derive the desired judgement by applying the elaborated rule with the derivable instantiation I'. □

Corollary 10.2.7 The retrogression transformation $r\colon \mathcal{S}^{i+} \to \mathcal{T}^{i+}$ is a type-theoretic transformation.

Proof. The retrogression transformation is already a syntactic transformation. We just need to prove that specific rules $\Xi \Longrightarrow \mathcal{j}$ of $\mathcal{S}^{i+}$ are mapped to derivable judgements.

- if $\Xi \Longrightarrow \mathcal{j}$ is a specific object rule, then

  $$\mathcal{j} = \mathcal{b}'\boxed{\mathsf{S}_{\left(k,\Theta \Longrightarrow \mathcal{b}\boxed{e}\right)}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})}$$

  for some specific object rule $R_k = \Theta \Longrightarrow \mathcal{b}\boxed{e}$ in the theory $\mathcal{T}^{i+}$. We then have

  $$r_*(\Xi \Longrightarrow \mathcal{j}) = \Theta \Longrightarrow \mathcal{b}\boxed{e}$$

  which has a generic derivation in $\mathcal{T}^{i+}$, i.e. it is just an application of that same rule with generically applied metavariables.
- If $\Xi \Longrightarrow \mathcal{j}$ is a specific equality rule in $\mathcal{S}^{i+}$, then there exists $k \in \mathbb{N}$ such that $\Xi \Longrightarrow \mathcal{j}$ is a specific rule in $\mathcal{S}^i_k$, for which by definition we have a derivation of $r_*(\Xi);[] \vdash_{\mathcal{T}^{i+}} r_*(\mathcal{j})$.

□

Corollary 10.2.8 The retrogression map $r\colon \mathcal{S}^{i+} \to \mathcal{T}^{i+}$ is conservative.

Proof. Let $\Xi; [] \vdash_{\mathcal{S}^{i+}} \mathcal{b}$ be a strongly derivable boundary in the theory $\mathcal{S}^{i+}$, such that

$$r_*(\Xi); [] \vdash_{\mathcal{T}^{i+}} (r_*\mathcal{b})\boxed{e}$$

is derivable by some derivation $\mathcal{D}$. By Lemma 10.2.6 the judgement

$$\Xi; [] \vdash_{\mathcal{S}^{i+}} \mathcal{b}\boxed{\ell(\Xi, [], \mathcal{b}, \mathcal{D})}$$

is derivable. □

10.3. Algorithmic properties of elaboration

The usual use case for elaboration is in a proof assistant. We want to enable the user to input judgements in a finitary type theory (dropping some annotations), but the theory behind the curtain is a well-behaved standard type theory. So the question arises, can we algorithmically elaborate judgements from a finitary type theory into a standard one?

We can view elaboration as a procedure, an *elaborator*, mimicking the desired behavior in a proof assistant: it takes a judgement in a finitary type theory and gives its strongly derivable elaboration, if it exists, otherwise reports there is none. Implementing an elaborator in a proof assistant raises the question if the procedure is computable. But having a computable elaborator (with the above definition) also means we have a decidable algorithm for checking if judgements are strongly derivable. There is no hope of having such a decision procedure for all judgements of a type theory as metacontexts can contain too many equalities which is shown in the following example.

[124]: Post (1947), "Recursive Unsolvability of a Problem of Thue"
[95]: Markov (1947), "Impossibility of certain algorithms in the theory of associative systems"
[96]: Markov (1947), "Impossibility of certain algorithms in the theory of associative systems"
[101]: Matiyasevich (1967), "Simple examples of undecidable associative calculi"
[94]: Makanin (1966), "On the identity problem in finitely defined semigroups"

Example 10.3.1 Having equalities in metacontexts enables us to encode the semigroup word problem, which is proven to be undecidable [94–96, 101, 124]. Consider the empty type theory (with no specific rules) and suppose we have a computable elaborator for it. Since the metacontexts can contain equality metavariables, we can form the following derivable metacontext

Θ = [G : (□ type),
∘ : ({x:G}{y:G}□ : G),
assoc : ({x:G}{y:G}{y:G}$(x \circ y) \circ z \equiv x \circ (y \circ z)$: G by □),
A : (□ : G), B : (□ : G), C : (□ : G), E_1 : (CCBB ≡ BBCC : G by □),
E_2 : (BCCCBB ≡ CBBBCC : G by □),
E_3 : (ACCBB ≡ BBA : G by □),
E_4 : (ABCCCBB ≡ CBBA : G by □),
E_5 : (BBCCBBBBCC ≡ BBCCBBBBCCA : G by □)]

We write the symbol ∘ in the infix form for clarity.

In the boundaries of equational metavariables E_1, ..., E_5 the concatenation of the metavariables A, B and C is just short for applying the metavariable ∘. We also drop the parenthesis, because ∘ is associative.

specifying a semigroup with three generators and five equations. Using the elaborator on judgements of the form $\Theta; [] \vdash u \equiv v : \mathsf{G}$ for suitable expressions u and v decides whether this judgement is derivable and thus encodes the semigroup word problem for this semigroup, which was proven by Makanin [94] to be undecidable. Therefore even the empty type theory cannot have a computable elaborator for all judgements.

[94]: Makanin (1966), "On the identity problem in finitely defined semigroups"

To avoid problems with computability we restrict our view to the judgements without equational metavariables in the metavariable context. We call such (meta)contexts *equation-free (meta)contexts*.

We can now formally define what is an elaborator. As before we fix a finitary type theory $\mathcal{T}$ and its elaboration to a standard type theory $\mathcal{S}$ with the retrogression transformation r and the elaboration map ℓ.

Because we will be considering decision procedures, we also assume that the specific rules of $\mathcal{T}$ are computable (with the well-founded order beneath).

Definition 10.3.2 An *elaborator* for $\mathcal{T}$ is an algorithm that takes a judgement

$$\vdash_{\mathcal{T}} \Theta\ \mathsf{mctx} \quad \text{or} \quad \Theta \vdash_{\mathcal{T}} \Gamma\ \mathsf{vctx} \quad \text{or} \quad \Theta; \Gamma \vdash_{\mathcal{T}} \mathcal{B} \quad \text{or} \quad \Theta; \Gamma \vdash_{\mathcal{T}} \mathcal{J}$$

in the finitary theory $\mathcal{T}$, where Θ is an equation-free metacontext, and

- gives a strongly derivable judgement

  $$\vdash_{\mathcal{S}} \Theta'\ \mathsf{mctx}, \quad \Theta' \vdash_{\mathcal{S}} \Gamma'\ \mathsf{vctx}, \quad \Theta'; \Gamma' \vdash_{\mathcal{S}} \mathcal{B}', \quad \text{or} \quad \Theta'; \Gamma' \vdash_{\mathcal{S}} \mathcal{J}'$$

  respectively in the standard theory $\mathcal{S}$ and derivations exhibiting strong derivability, such that

  $$r_*(\Theta') = \Theta \qquad r_*(\Gamma') = \Gamma \qquad r_*(\mathcal{J}') = \mathcal{J} \qquad r_*(\mathcal{B}') = \mathcal{B}$$

  if such a strongly derivable judgement exists,
- reports there is no such strongly derivable judgement, otherwise.

Needelss to say an elaborator, if it exists, is *computable* for our chosen type theory. In Subsection 10.3.2 we give a characterization of when the elaborator exists, namely that the finitary type theory $\mathcal{T}$ has decidable checking.

10.3.1. Type-theoretic checking

Any user of typed programming languages is familiar with the concept of *type checking*. In proof assistants based on type theories it usually takes the form of checking that a term judgement is well-formed and derivable, namely that the given term has the prescribed type. In the meta-language of finitary type theories, type checking is a procedure

that given a (strongly) derivable term-boundary

$$\Theta; \Gamma \vdash \square : A$$

and a term expression $t \in \mathsf{Expr}_\Sigma(\mathsf{Tm}, \Theta; \Gamma)$ checks if the judgement

$$\Theta; \Gamma \vdash t : A$$

is (strongly) derivable, or reports that it is not.

Similarly, equality checking is a procedure that starts with a derivable equality boundary

$$\Theta; \Gamma \vdash A \equiv B \text{ by } \square \qquad \text{or} \qquad \Theta; \Gamma \vdash s \equiv t : A \text{ by } \square$$

and checks if the boundaries can be derivably filled with the head $\star$.

Example 10.3.1 shows us that no type theory can have decidable equality checking for all judgements, so we again restrict the condition to the judgements in equation-free metacontexts.

We observe that both type checking and equality checking have the same form in the meta-language of finitary type theories: we start with a derivable boundary and a head that syntactically fits the given boundary, and check that the judgement is derivable. We can therefore generalize these procedures in the concept of *checking*.

Definition 10.3.3 The *checking* procedure takes a judgement

$$\Theta; \Gamma \vdash \mathscr{b} \boxed{\mathscr{e}}$$

in an equation-free metacontext Θ with a strongly derivable boundary $\mathscr{b}$ and reports whether the judgement is derivable.

Note that in our definition of the checking procedure the output is a guarantee that the judgement is derivable without the actual derivation. Another option would be to take as input a derivation that $\mathscr{b}$ is strongly derivable and output the entire derivation of the judgement. The approaches are equivalent:

- If the procedure does not take a derivation as input, we can just forget it.
- If the procedure needs a derivation, but it is not given, we can run the semi-decidable procedure that searches for a derivation[7]. Since we are guaranteed the derivation exists, the search will surely terminate with an appropriate derivation, which we can feed to the checking procedure.

Similarly to checking, we define the equality checking procedure.

Definition 10.3.4 The *equality checking* procedure takes a strongly derivable equality boundary

$$\Theta; \Gamma \vdash A \equiv B \text{ by } \square \qquad \text{or} \qquad \Theta; \Gamma \vdash s \equiv t : A \text{ by } \square$$

in an equation-free metacontext Θ and either yields a derivable

7: In this meta-level reasoning we are using the Markov principle and the fact that the set of specific rules is computably enumerable. Since taking the derivations is more in line with the constructive reasoning, we work with this definition.

judgement

$$\Theta;\Gamma \vdash A \equiv B \text{ by } \star \qquad \text{or} \qquad \Theta;\Gamma \vdash s \equiv t : A \text{ by } \star$$

or reports it is not derivable.

In Part 'An equality checking algorithm' we thoroughly describe an equality checking algorithm that acts as an equality checking procedure and is adaptable for a large class of theories. However, whether a type theory has an equality checking procedure depends on the rules of the theory itself.

We say that a finitary type theory has ***decidable (equality) checking***, if there exists a computable (equality) checking procedure.

Proposition 10.3.5 If a standard type theory $\mathcal{S}$ has decidable equality checking, then it has decidable checking.

Proof. Suppose $\mathcal{S}$ is a standard type theory with decidable equality-checking. Suppose $\Theta;\Gamma \vdash \mathcal{b}$ is a strongly derivable boundary with Θ an equation-free metacontext and let $\Theta;\Gamma \vdash \mathcal{b}\boxed{e}$ be the judgement we would like to check. If $\mathcal{b}$ is an equality boundary, we can run the equality checking algorithm to get the result. So suppose $\mathcal{b}$ is an object boundary and let us first consider the case when $\mathcal{b} = \square : A$. By Theorem 5.2.2 we can confine ourselves to finding a derivation that ends with TT-Var, TT-Meta or a symbol rule, and each of those can be followed by one conversion using TT-Conv-Tm. We use recursion on the structure of e:

Case $e = \mathsf{a}$: Use equality checking algorithm to check if $A \equiv \Gamma(\mathsf{a})$. If so, then we use TT-Var combined with TT-Conv-Tm to obtain the desired derivation. Otherwise report there is none.

Case $e = \mathsf{M}(t_1, \ldots, t_m)$: Let

$$\Theta(\mathsf{M}) = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\square : B$$

Recursively check judgements

$$\Theta;\Gamma \vdash t_j : A_j[t_1/x_1, \ldots, t_{j-1}/x_{j-1}] \quad \text{for } j = 1, \ldots, m \tag{10.9}$$

If any of them fail, report there is no derivation. Otherwise using the equality checking algorithm check that

$$\Theta;\Gamma \vdash B[t_1/x_1, \ldots, t_m/x_m] \equiv A$$

is derivable. If not, then report there is no derivation. Otherwise we get the desired derivation using TT-Meta on (10.9) and combining it with TT-Conv-Tm on equation (10.3.1).

Case $e = \mathsf{S}(e_1, \ldots, e_n)$: Since $\mathcal{S}$ is a standard type theory, specific object rules are symbol rules. Let

$$[\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n] \Longrightarrow \mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n}) : B$$

be the symbol rule for S in the theory $\mathcal{S}$. Recursively check that the instantiation

$$I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$$

of $[\mathsf{M}_1 : \mathscr{B}_1, \ldots, \mathsf{M}_n : \mathscr{B}_n]$ over $\Theta; \Gamma$ is derivable, using TT-Abstr on the abstracted boundaries and recursively calling the algorithm on premises. If the instantiation is not derivable, report there is no derivation. Otherwise using the equality checking algorithm check that the natural type matches the type from the boundary, i.e. check that the equation

$$\Theta; \Gamma \vdash I_*(B) \equiv A \tag{10.10}$$

holds. If not, report there is no derivation. Otherwise we get the desired derivation using the derivable instantiation I on the symbol rule for S, followed by TT-Conv-Tm with equation (10.10).

If $\mathscr{b} = \square\ \mathsf{type}$, the procedure is similar to the term judgement. □

While standard type theories are sufficiently well-behaved for Proposition 10.3.5 to hold, finitary type theories are not: decidable equality checking for a finitary type theory does not imply decidable checking. We demonstrate this fact in Example 10.3.8. But before we show the example, we prove that a theory without specific equational rules has decidable equality checking.

Lemma 10.3.6 If $\mathcal{T}$ is a raw type theory with no specific equality rules, then every derivable equality in an equation-free metacontext that holds in $\mathcal{T}$ can be derived using TT-Ty-Refl or TT-Tm-Refl.

Proof. Let Θ be an equation-free metacontext. We consider type and term equations separately.

Let $\Theta; \Gamma \vdash A \equiv B$ be a derivable equality in $\mathcal{T}$. We proceed by induction on the derivation. By inversion we consider the following cases.

Case TT-Ty-Refl: Trivial.

Case TT-Ty-Sym: By induction we have a derivation of $\Theta; \Gamma \vdash B \equiv A$ using TT-Ty-Refl which implies that A and B are syntactically equal. Thus we can derive $\Theta; \Gamma \vdash A \equiv B$ using TT-Ty-Refl as well.

Case TT-Ty-Tran: By induction we have a derivation of $\Theta; \Gamma \vdash A \equiv C$ and $\Theta; \Gamma \vdash C \equiv B$ using TT-Ty-Refl which implies that A, B and C are syntactically equal. Thus we can derive $\Theta; \Gamma \vdash A \equiv B$ using TT-Ty-Refl as well.

Case Congruence rule: By induction hypothesis on the premises implies that the arguments (of a symbol or a metavariable) are syntactically equal, so A and B are also syntactically equal.

Note that there is no case for the metavariable rule or an instantiation of a specific rule. The first is due to Θ being equation-free and the latter is because there are no specific equality rules.

For a term equation suppose $\Theta; \Gamma \vdash s \equiv t : A$ be a derivable equality in $\mathcal{T}$. We again proceed by induction on the derivation.

Case TT-Tm-Refl, TT-Tm-Sym, TT-Tm-Tran *and congruence rule*: Similar to the type case.

Case TT-Conv-Eq: Suppose the derivation ends with

$$\frac{\Theta;\Gamma \vdash s \equiv t : A \qquad \Theta;\Gamma \vdash A \equiv B}{\Theta;\Gamma \vdash s \equiv t : B}.$$

By induction hypothesis s and t are syntactically equal, as well as A and B. So we can indeed derive $\Theta;\Gamma \vdash s \equiv t : B$ using TT-Tm-Refl.

Again there is no case for the metavariable rule or an instantiation of a specific rule. □

Lemma 10.3.7 If $\mathcal{T}$ is a raw type theory with no specific equality rules, then $\mathcal{T}$ has decidable equality checking.

Proof. Let $\Theta;\Gamma \vdash \mathcal{B}$ be a strongly derivable equational boundary in an equation-free metacontext Θ. By Lemma 10.3.6 every derivable equality can be derived by the reflexivity rule, so equality checking reduces to checking syntactic equality which is decidable. □

Example 10.3.8 Suppose $D \subseteq \mathbb{N} \times \mathbb{N}$ is a computable set such that its projection

$$\pi_1(D) = \{n \in \mathbb{N} \mid \exists m \in \mathbb{N}.\ (n,m) \in D\}$$

is semidecidable, but not computable. We construct the following finitary type theory $\mathcal{T}$:

- The signature of $\mathcal{T}$ is given by $\Sigma_{\mathcal{T}} = (\mathsf{A}_n{:}(\mathsf{Ty}, []))_{n\in\mathbb{N}}$.
- For every $(n,m) \in D$ there is a specific rule

$$R_{(n,m)} = ([] \Longrightarrow \mathsf{A}_n\ \mathsf{type}).$$

This is indeed a finitary type theory, we can take the lexicographic order on the index set D for a well-founded order. Theory $\mathcal{T}$ has decidable equality checking by Lemma 10.3.7. Note that not every symbol in the signature $\Sigma_{\mathcal{T}}$ appears in a specific rule.

Now observe that $[];[] \vdash \square\ \mathsf{type}$ is a derivable boundary. We would like to check

$$[];[] \vdash \mathsf{A}_n\ \mathsf{type}$$

for some $n \in \mathbb{N}$.

If the type theory $\mathcal{T}$ had decidable boundary-checking, we would be able to decide the semidecidable set $\pi_1(D)$ which is a contradiction. Thus we have constructed a finitary type theory with decidable equality checking, but undecidable checking.

The crux of the problem is that the conclusions of the specific rules do not record all the information stored in the index of the rule. In the elaboration of this theory as a standard type theory, only

It is easy to find an example of a computable set whose first projection is not computable, but just semidecidable. We could for instance use the halting set as the projection of

$$D = \{(n,m,k) \in \mathbb{N}^3 \mid T(n,m,k)\}$$

where $T(n,m,k)$ is the predicate which is true when the Turing machine encoded with n halts on input encoded with m with the execution trace encoded with k. Here we assume we have encodings of Turing machines, inputs and traces into $\mathbb{N}$ as well as pairs of natural numbers. We implicitly use conversion along these encodings which have been constructed [11, 143]. The set D is clearly decidable, as we can run the Turing machine n on given input m and check if the execution trace matches k.

[143]: Turing (1937), "On Computable Numbers, with an Application to the Entscheidungsproblem"
[11]: Asperti et al. (2015), "A formalization of multi-tape Turing machines"

derivable type symbols appear in the signature and therefore pose no problem to the decidability of checking.

A similar problem to the one in Example 10.3.8 can occur when not all object premises are faithfully recorded in the conclusion of a derivable judgement in a finitary type theory.

10.3.2. Algorithmic properties of elaborators

Now that we have the definition of checking, we can finally state the theorem giving characterization of when an elaborator is computable.

Theorem 10.3.9 A finitary type theory $\mathcal{T}$ has an elaborator if and only if $\mathcal{T}$ has decidable checking.

Proof. We need to prove both directions of the equivalence.

Part ($\Longrightarrow$): Suppose $\mathcal{T}$ has an elaborator. We need to prove that $\mathcal{T}$ has decidable checking. Let $\Theta;\Gamma \vdash_{\mathcal{T}} \mathcal{b}$ be a strongly derivable boundary in an equation-free metacontext Θ and e an expression that syntactically fits the boundary. We want to algorithmically check if the judgement

$$\Theta;\Gamma \vdash_{\mathcal{T}} \mathcal{b}\boxed{e} \tag{10.11}$$

is derivable in theory $\mathcal{T}$. We apply the following algorithm:

1. Run the elaborator on the judgement (10.11).
2. Depending on the result:
   - If the elaborator succeeds with some judgement
     $$\Theta';\Gamma' \vdash_{\mathcal{S}} \mathcal{J}'$$
     the judgement (10.11) is derivable.
   - If the elaborator reports there is no elaboration candidate, report that (10.11) is not derivable.

Since the elaborator is computable, running it on the judgement (10.11) will terminate. If the elaborator succeeds, we get a derivable elaboration candidate for (10.11), so the following equations hold:

$$r_*(\Theta') = \Theta \qquad r_*(\Gamma') = \Gamma \qquad r_*(\mathcal{J}') = \mathcal{b}\boxed{e}.$$

Since the retrogression transformation r is a type-theoretic transformation, it preserves derivability by Theorem 9.1.3, so (10.11) is derivable as well. If the elaborator reports there is no elaboration candidate, the checking correctly fails as well, because all strongly derivable judgements have a derivable elaboration candidate.

Part ($\Longleftarrow$): For the other direction, suppose that $\mathcal{T}$ has decidable checking. Since ℓ as defined in the proof of Lemma 10.2.3 is computable we only need to compute its inputs. Because $\mathcal{T}$ has decidable judgement-checking we can compute the input derivations recursively. □

An elaborator is thus the most general checking algorithm if any exists, as it not only checks derivability of a judgement, but also computes its elaboration thus recovering all the missing annotations. Consequently having an elaborator gives rise to checking algorithms for the elaboration.

Theorem 10.3.10 If a finitary type theory $\mathcal{T}$ has decidable equality-checking, so does its elaboration $\mathcal{S}$.

Proof. Let $\Theta; \Gamma \vdash_{\mathcal{S}} \mathcal{b}$ be a strongly derivable equality boundary in $\mathcal{S}$ in the equation-free metacontext Θ. We run checking algorithm for theory $\mathcal{T}$ on the judgement[8]

$$r_*(\Theta); r_*(\Gamma) \vdash_{\mathcal{T}} (r_*(\mathcal{b}))\boxed{\star}. \tag{10.12}$$

8: The boundary $r_*(\Theta); r_*(\Gamma) \vdash_{\mathcal{T}} r_*(\mathcal{b})$ is strongly derivable because r preserves derivability.

If the algorithm reports failure, we propagate the error report: if $\Theta; \Gamma \vdash_{\mathcal{S}} \mathcal{b}\boxed{\star}$ were derivable, (10.12) would be also. Otherwise we get a derivation $\mathcal{D}$ of (10.12). The equality $\Theta; \Gamma \vdash_{\mathcal{S}} \mathcal{b}\boxed{\star}$ is derivable by Lemma 10.2.5. □

Corollary 10.3.11 If $\mathcal{T}$ has decidable checking, so does its elaboration $\mathcal{S}$.

Proof. Since $\mathcal{T}$ has decidable checking it also has decidable equality checking. By Theorem 10.3.10 the elaboration $\mathcal{S}$ has decidable equality checking as well. Because $\mathcal{S}$ is a standard type theory by Proposition 10.3.5 it has decidable checking. □

We note that the converse of the Corollary 10.3.11 does not hold: The elaboration as a standard type theory $\mathcal{S}$ of a finitary type theory $\mathcal{T}$ having decidable checking does not imply that $\mathcal{T}$ had decidable checking as well. The counter-example is again the finitary type theory $\mathcal{T}$ from Example 10.3.8. The theory has decidable equality checking, so by Theorem 10.3.10 its elaboration $\mathcal{S}$ has decidable equality checking as well. By Proposition 10.3.5 $\mathcal{S}$ has decidable checking, but we have shown in Example 10.3.8 that the theory $\mathcal{T}$ does not.

11. Discussion

11.1. Related work

11.1.1. Translations of formal systems

There are various kinds of transformations between formal systems and they are performed on different levels: transformations between logics, from logic to type theory, providing semantic models or an interpretation etc. We only summarize some of the transformations that are most relevant to our work. Some of these transformations were already compared to our definition of type-theoretic transformations in Section 9.3.

On the level of transformations between logics is the *double negation translation* that was developed by Kolmogorov [83], Gödel [62], Gentzen [56, 57, 138], Kuroda [86] and Krivine [84] whose versions slightly differ. The idea of the translation is to take each classical proposition into its double negation and so translate a classically valid formula into an intuitionistically valid one. While this is a translation on the level of syntax, it is not a type-theoretic transformation in the sense of Definition 9.1.2 as shown in Example 9.3.7. However, we could present it as a partial map from derivations to judgements, similarly to the elaboration map.

A transformation from a classical to an intuitionistic setting can also be achieved when we combine the double negation translation with another transformation. We can for instance couple it with the *Dialectica interpretation* [14, 63], an interpretation of a formula in intuitionistic (Heyting) arithmetic in a quantifier-free formula of System T, or the *A-translation* [55] which relate to type-theoretic transformations similarly to the double-negation translation. The computational analogue of the double negation translation is the *continuation passing style translation* ([54, 126]). The Dialectica interpretation was later reformulated by Pédrot [113, 114] into the *functional functional interpretation*.

The famous translation of logic to type theory is the is *propositions as types*, or the Curry-Howard correspondence [48, 49, 74, 149], giving proofs a computatinal content, which is shown to adhere to the definition of type-theoretic transformations in Example 9.3.1 and Appendix Chapter A. The counterpart of the Curry-Howard correspondence is the *propositions as bracketed or squash types* translation [16, 44, 117, 142] that hides the computational content.

On the level of syntactic transformations between type theories there is the *elimination of equality reflection* from intentional type theory. It was implemented in Coq by Winterhalter, Sozeau and Tabareau [152]. They translate derivations of extensional type theory (ETT) into proof terms of intensional type theory (ITT), so as mentioned in Section 9.3

[83]: Kolmogorov (1925), "On the principles of excluded middle (Russian)"

[62]: Gödel (1933), "Zur intuitionistischen Arithmetik und Zahlentheorie"

[57]: Gentzen (1974), "Über das Verhältnis zwischen intuitionistischer und klassischer Arithmetik"
[138]: Szabo (1971), "The Collected Papers of Gerhard Gentzen"
[56]: Gentzen (1936), "Die Widerspruchsfreiheit der reinen Zahlentheorie"

[86]: Kuroda (1951), "Intuitionistische Untersuchungen der formalistischen Logik"
[84]: Krivine (1990), "Opérateurs de mise en mémoire et traduction de Gödel"

[63]: Gödel (1958), "Über eine bisher nicht erweiterung des finiten standpunktes"
[14]: Avigad et al. (1998), "Gödel's Functional Interpretation"
[55]: Friedman (1978), "Classically and intuitionistically provably recursive functions"

[54]: Fischer (1993), "Lambda-Calculus Schemata"
[126]: Reynolds (1972), "Definitional Interpreters for Higher-order Programming Languages"

[113]: Pédrot (2014), "A functional functional interpretation"
[114]: Pédrot (2015), "A Materialist Dialectica. (Une Dialectica matérialiste)"

[152]: Winterhalter et al. (2019), "Eliminating Reflection from Type Theory"

it is more similar to the elaboration map than a type-theoretic transformation. The translation from ETT to ITT was first done on a semantic level (categorically) by Hofmann in 1995 [70, 72], later syntactically by Oury [112] and has now been implemented.

[70]: Hofmann (1996), "Conservativity of Equality Reflection over Intensional Type Theory"
[72]: Hofmann (1997), *Extensional Constructs in Intensional Type Theory*
[112]: Oury (2005), "Extensionality in the Calculus of Constructions"

Based on Boulier's syntactic models [29, 30], Winterhalter in his PhD thesis also proposes a definition of syntactic translations [151] which is given by two maps: one from types to types and the other from terms to terms. Such a notion of a transformation is more flexible than our syntactic and type-theoretic transformations. However, as Winterhalter points out, the usual properties that one desires from a syntactic translation (type preservation; preservation of falsehood, reduction and conversion; relative consistency; substitutivity) impose enough restrictions that Winterhalter's syntactic translations in their setting coincide with the type-theoretic transformations we propose in Definition 9.1.2.

[29]: Boulier et al. (2017), "The next 700 syntactical models of type theory"
[30]: Boulier (2018), "Extending type theory with syntactic models"
[151]: Winterhalter (2020), "Formalisation and Meta-Theory of Type Theory"

On the semantic level, Dybjer [52] defines morphisms of categories with families (CwFs) which preserve the structure on-the-nose, meaning the (variable) context extension is preserved and so are substitutions and judgement forms. Similarly Uemura [144] defines morphisms of type theories as morphisms of categories with representable maps (CwRFs). These also preserve the structure: preserving finite limits leads to preservation of empty contexts and is-type judgement in our case, preservation of representable maps relates to the preservation context comprehension, preservation of pushforwards along representable maps would mean the abstractions are also preserved. There is also the notion of weak morphism of CwFs [27] also called pseudomorphism in [80] that preserves the empty context and context extension just up to isomorphism. Intuitively, these concepts on the semantic level seem to closely relate to the properties of type-theoretic transformations on the syntactic level. However, it would require to construct syntactic models of finitary type theories in order to be able to precisely relate morphisms of CwFs and CwRFs to our notion of type-theoretic transformations. We leave such endeavors for the future.

[52]: Dybjer (1995), "Internal Type Theory"

[144]: Uemura (2019), *A General Framework for the Semantics of Type Theory*

[27]: Birkedal et al. (2020), "Modal dependent type theory and dependent right adjoints"
[80]: Kaposi et al. (2019), "Gluing for Type Theory"

In their work on general type theories Bauer, Lumsdaine and Haselwarter also propose a definition of a raw type theory map that is analogous to the type-theoretic transformation from Definition 9.1.2. They use the map to construct a well-founded replacement, which shows that sufficiently nice raw type theories can be given a well-founded ordering on the rules. The definition of the well-founded replacement can be adapted to our setting and the theorem holds with essentially the same proof.

11.1.2. Type-inference and elaboration

Since writing fully annotated syntax very quickly becomes unreadable and unmanageable, the idea of elaboration as a soft concept has been around for a while, the initial idea ususally being attributed to Robert Pollack [123]. As Francisco Ferreira and Brigitte Pientka write in [53]:

[123]: Pollack (1992), "Implicit Syntax"
[53]: Ferreira et al. (2014), "Bidirectional Elaboration of Dependently Typed Programs"

> *"To make programming with dependent types practical, dependently typed systems provide a compact language for programmers where one can omit some arguments, called implicit, which can be inferred. This source language is then usually elaborated into a core language where type checking and fundamental properties such as normalization are well understood. Unfortunately, this elaboration is rarely specified and in general is illunderstood."*

and Jesper Cockx and Andreas Abel write in [40]:

> *"Dependently typed functional languages (. . .) combine programming and proving into one language, so they should be at the same time expressive enough to be useful and simple enough to be sound. These apparently contradictory requirements are addressed by having two languages: a high-level surface language that focuses on expressivity and a small core language that focuses on simplicity. The main role of the typechecker is to elaborate the high-level surface language into the low-level core. Since the difference between the surface and core languages can be quite large, the elaboration process can be, well, elaborate."*

Consequently there is a plethora of techniques for it [12, 32, 40, 53, 68, 85, 89, 107, 110, 123]. The idea of elaboration is closely related to bidirectional type-checking in programming languages [122], the concept whose origins and development is summarised in [51]. Usually elaboration is considered to begin at the stage where we are dealing with the intricacies of the core type theory rather than just syntax of the surrounding programming language.

[40]: Cockx et al. (2018), "Elaborating Dependent (Co)Pattern Matching"

[123]: Pollack (1992), "Implicit Syntax"
[68]: Harper et al. (1998), "A Type-Theoretic Interpretation of Standard ML"
[89]: Lee et al. (2007), "Towards a Mechanized Metatheory of Standard ML"
[110]: Norell (2007), "Towards a practical programming language based on dependent type theory"
[12]: Asperti et al. (2012), "A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions"
[32]: Brady (2013), "Idris, a general-purpose dependently typed programming language: Design and implementation"
[53]: Ferreira et al. (2014), "Bidirectional Elaboration of Dependently Typed Programs"
[107]: Moura et al. (2015), *Elaboration in Dependent Type Theory*
[40]: Cockx et al. (2018), "Elaborating Dependent (Co)Pattern Matching"
[85]: Kudasov (2021), *A proof assistant for synthetic ∞-categories*
[122]: Pierce et al. (2000), "Local type inference"
[51]: Dunfield et al. (2020), *Bidirectional Typing*

11.2. Future directions

The first possible next step would be to implement type-theoretic transformations for standard type theories in the Andromeda 2 proof assistant. Transformations could be thought of as a technique in proof development. However, for such a technique to be usable in practice, setting up the transformations should be convenient for the users and have as little overhead as possible. It would be interesting to see how the use of transformations impacts efficiency in terms of computational resources, length of proof scripts and effort invested by the user.

On the theoretical side, the category of type theories from Definition 9.2.1 could be submitted to a thorough investigation. While we only mention two properties, namely the initial object and coproducts, the language of category theory is very expressive and could provide the tools for further meta-analysis of type theories and how they interact. Another direction is relating the syntactic representations of finitary type theories as described in Part 'Finitary Type Theories' with the semantic notions, for example the categories with representable maps of Uemura [144], and the type-theoretic transformations with morphisms of CwRFs through syntactic models of finitary type theories.

[144]: Uemura (2019), *A General Framework for the Semantics of Type Theory*

When the ideas of the elaboration theorem were presented [116], Mike Shulman raised a question about the reverse direction of elaboration. Designing a proof assistant usually starts with a well-behaved core (standard) type theory and then some arguments are made implicit to obtain a finitary type theory, the opposite order of the elaboration theorem. It would be very useful to know under what conditions an argument can be omitted while the core theory remains an elaboration of the finitary one.

[116]: Petković Komel (2021), *Towards an elaboration theorem*

The full meta-theory of type-theoretic transformations as developed in this thesis has not yet been formalized, hence the long paper-proofs are necessary. It would certainly benefit future developments to formalize the syntax of finitary type theories and the transformations between them. First steps towards such a formalization have already been made [19, 93]. But before we dig in the formalization, we can make the following observation. As noticed in Section 7.1, the substitutions, instantiations and type-theoretic transformations seem to behave very similarly: they all instantiate some kind of "variables", be it variables from a variable context, metavariables from a metacontext or symbols from a signature. The theorems about their interactions are also very similar (for example Theorem 5.1.5 and Proposition 9.1.6). It remains to be seen if and how we can unify the syntax of all three kinds of variables (variables, metavariables and symbols) and prove the theorems about them in a uniform way. Such a simplification of the syntax would inevitably simplify the formalization as well. Another possible generalisation lies in the prescribed four judgement forms. Following the example of Uemura's definition of type theories [144] the judgement forms could be more flexible and subsume further examples like the interval judgement of cubical type theories. It would also allow for a more general notion of a type-theoretic transformation, where judgement forms are not necessarily preserved, but enough of the type-theoretic structure interacts well with the transformation.

[93]: Loutchmia et al. (2021), *Formalization of simple type theory*
[19]: Bauer (2021), *Syntax of dependent type theories*

[144]: Uemura (2019), *A General Framework for the Semantics of Type Theory*

An equality checking algorithm

12. Overview

Equality checking algorithms are essential components of proof assistants based on type theories [2, 5, 45, 58, 108, 133]. They free users from the burden of proving scores of mostly trivial judgemental equalities, and provide computation-by-normalization engines. Some systems [39, 50] go further by allowing user extensions to the built-in equality checkers.

The situation is less pleasant in a proof assistant that supports arbitrary user-definable theories, such as Andromeda 2 [9, 20], where in general no equality checking algorithm may be available. Nevertheless, the proof assistant should still provide support for equality checking that is easy to use and works well in the common, well-behaved cases. For this purpose we have developed and implemented a sound and extensible equality checking algorithm for user-definable type theories.

The generality of type theories supported by Andromeda 2 presents a significant challenge in devising a useful equality checking algorithm. Many commonly used ideas and notions that one encounters in specific type theories do not apply anymore: not every rule can be classified either as an introduction or an elimination form, not every equation as either a β- or an η-rule, all terms must be fully annotated with types to ensure soundness, there may be no reasonable notion of a normal form, or a neutral form, etc. And of course, the user may easily define a theory whose equality checking is undecidable. In order to do better than just exhaustive proof search, some compromises must therefore be made and design decisions taken:

1. We work in the fully general setting of standard type theories.
2. We prefer ease of experimentation at the expense of possible non-termination or unpredictable behavior.
3. At the same time, soundness of the algorithm is paramount: any equation verified by it must be derivable in the theory at hand.
4. The algorithm should work well on well-behaved theories, and especially those seen in practice.

The most prominent design goals missing from the above list are completeness and performance. The former cannot be achieved in full generality, as there are type theories with undecidable equality checking. We have expended enough energy looking for acceptably general sufficient conditions guaranteeing completeness to state with confidence that this task is best left for another occasion. Regarding performance, we freely admit that equality checking in Andromeda 2 is nowhere near the efficiency of established proof assistants. For this we blame not only the immaturity of the implementation, but also the generality of the situation, which simply demands that a price be paid in exchange for soundness. We console ourselves with the fact that our equality checker achieves soundness and complete user-extensibility at the same time.

[45]: (2021), *The Coq proof assistant, version 2021.02.2*
[5]: (2021), *The Agda proof assistant*
[108]: Moura et al. (2015), "The Lean Theorem Prover (System Description)"
[133]: Sozeau et al. (2019), "Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq"
[58]: Gilbert et al. (2019), "Definitional proof-irrelevance without K"
[2]: Abel et al. (2017), "Decidability of Conversion for Type Theory in Type Theory"

[50]: *The Dedukti logical framework*
[39]: Cockx et al. (2016), "Sprinkles of extensionality for your vanilla type theory"

[9]: Bauer et al. *The Andromeda proof assistant*
[20]: Bauer et al. (2018), "Design and Implementation of the Andromeda Proof Assistant"

12.1. Contributions

We present a *general equality checking algorithm* that is applicable to a large class of type theories, the *standard type theories* (Definition 4.4.5). The algorithm (Section 15.2) is fashioned after equality checking algorithms [3, 136] that have a type-directed phase for applying extensionality rules (inter-derivable with η-rules), intertwined with a normalization phase based on computation rules (β-rules). For the usual kinds of type theories (simply typed λ-calculus, Martin-Löf type theory, System F), the algorithm behaves like the well-known standard equality checkers. We prove that our algorithm is *sound* (Section 15.3).

[136]: Stone et al. (2006), "Extensional equivalence and singleton types"
[3]: Abel et al. (2012), "On Irrelevance and Algorithmic Equality in Predicative Type Theory"

We define a general notion of *computation* (Section 14.1) and *extensionality rules* (Section 14.2), using the type-theoretic concept of an *object-invertible rule* (Section 13.2). We also provide sufficient *syntactic criteria* for recognizing such rules, together with a simple pattern-matching algorithm for applying them. A third component of the algorithm is a suitable notion of normal form, which guarantees correct execution of normalization and coherent interaction of both phases of the algorithm. In our setting, normal forms are determined by a selection of *principal arguments* (Section 15.1). By varying these, we obtain known notions, such as weak head-normal and strong normal forms.

We *implemented* the algorithm in Andromeda 2 (Chapter 16). The user need only provide the equality rules they wish to use, which the algorithm automatically classifies either as computation or extensionality rules, rejects those that are of neither kind, and selects appropriate principal arguments.

Those readers who prefer to see examples before the formal development, may first take a peek at Section 16.1, where we show how our work allows one to implement extensional type theory, and use the reflection rule to derive computation rules which are only available in their propositional form in intensional type theory.

13. Patterns and Object-invertible Rules

The equality checking algorithm derives the target equation by applying inference rules. To determine whether a rule can be applied and how it pattern matches it against parts of the target equation. We therefore begin by studying the syntactic and type-theoretic properties of rules which ensure the soundness of pattern matching.

13.1. Patterns

In principle there are many ways by which a judgement can be derived. In order determine whether the judgement $\Theta;\Gamma \vdash \mathcal{J}'$ can be derived with the rule $\Xi \Longrightarrow \mathcal{J}$, we must find an instantiation[1] I of Ξ over $\Theta;\Gamma$ such that $I_*\mathcal{J} = \mathcal{J}'$. We shall be primarily interested in rules where such I is unique, when it exists.

1: The instantiation I should also be derivable. But we first focus on the syntactic part.

Definition 13.1.1 A raw rule $\Xi \Longrightarrow \mathcal{J}$ is *deterministic* when for every judgement $\Theta;\Gamma \vdash \mathcal{J}'$ there exists at most one instantiation I of Ξ over $\Theta;\Gamma$ such that $I_*\mathcal{J} = \mathcal{J}'$, called a *matching instantiation*.

Whether a raw rule is deterministic may depend on the entire type theory and not just the rule itself.

We refrain from trying to characterize the deterministic rules, and instead observe that, given a deterministic rule

$$R = (\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathcal{J})$$

and a judgement $\Theta;\Gamma \vdash \mathcal{J}'$ we may algorithmically compute I such that $I_*\mathcal{J} = \mathcal{J}'$, or decide that it does not exist. First of all, every object metavariable of R must appear in $\mathcal{J}$, or else R would match in multiple ways the judgement $\Theta, \Theta'; [] \vdash \mathcal{J}$, where Θ' is a copy of Θ in which each M_i is replaced with M'_i. Therefore, for any instantiation

$$I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$$

where $\mathsf{ar}(e_i) = \mathsf{ar}(\mathscr{B}_i) = (c_i, n_i)$ and $e_i = \{x_1, \ldots, x_{n_i}\}e'_i$, the size of $I_*\mathcal{J}$ equals or exceeds the size of each e'_i. We may therefore look for an instantiation that matches $\Theta;\Gamma \vdash \mathcal{J}$ by exhaustively searching through all e'_i's over $\Theta;\Gamma$ whose sizes are bounded by the size of $\mathcal{J}$, of which there are only finitely many. Of course, we are not suggesting that anyone should use such an exhaustive search in practice. Instead, we provide a simple syntactic criterion that makes a rule deterministic and easy to match against.

Definition 13.1.2 *Patterns* are expressions in which every metavariable occurs at most once either in an application without arguments $\mathsf{M}()$, or in an argument of the form $\{\vec{x}\}\mathsf{M}(\vec{x})$, where $\vec{x}$ are the only bound variables in scope. They are described by the grammar in Figure 13.1.

$$
\begin{array}{rll}
\text{Type pattern } P & ::= \mathsf{M}() \mid \mathsf{S}(q_1, \ldots, q_n) & \text{if } \mathsf{mv}(q_i) \cap \mathsf{mv}(q_j) = \emptyset \text{ for } i \neq j \\
\text{Term pattern } p & ::= \mathsf{S}(q_1, \ldots, q_n) & \text{if } \mathsf{mv}(q_i) \cap \mathsf{mv}(q_j) = \emptyset \text{ for } i \neq j \\
\text{Argument pattern } q & ::= \{\vec{x}\}\mathsf{M}(\vec{x}) \mid P \mid p &
\end{array}
$$

Figure 13.1.: The syntax of patterns.

Note that $\mathsf{M}()$ can only appear as a type pattern, but not as a term pattern. The reason for this lies in the definitions of computation rules (Definition 14.1.1 Definition 14.1.2) which we shall see later on.

As defined, the patterns are *linear* in the sense that a metavariable cannot appear several times, and *first-order* because patterns may not appear under abstractions. Non-linearity is not an essential limitation, as we shall see shortly. The restriction to first-order patterns arises because in general a standard type theory may not satisfy the *strengthening* principle which states that if $\vdash \{x{:}A\}\mathcal{J}$ is derivable and $x \notin \mathsf{bv}(\mathcal{J})$ then $\vdash \mathcal{J}$ is derivable. The principle allows a higher-order pattern to safely extract an expression from within an abstraction, so long as no bound variables escape their scopes.

Reminder: notation mv()

The notation $\mathsf{mv}(e)$ is for the set of metavariables that appear in the expression e.

Example 13.1.3 The head of the conclusion of a symbol rule

$$\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{C}\boxed{\mathsf{S}(\widehat{M}_1, \ldots, \widehat{M}_n)}$$

is a pattern because $\widehat{\mathsf{M}}_i$ has required form $\{x\}\mathsf{M}_i(\vec{x})$.

Example 13.1.4 Consider the β-rule for the first projection from a binary product:

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{B}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{B}}{\vdash \mathsf{fst}(\mathsf{A}, \mathsf{B}, \mathsf{pair}(\mathsf{A}, \mathsf{B}, \mathsf{s}, \mathsf{t})) \equiv \mathsf{s} : \mathsf{A}}$$

The left-hand side of the conclusion is not a pattern because the metavariables A and B occur twice each. We may linearize the pattern at the cost of equational premises:

$$\frac{\begin{array}{cccc}\vdash \mathsf{A}_1\ \mathsf{type} & \vdash \mathsf{A}_2\ \mathsf{type} & \vdash \mathsf{B}_1\ \mathsf{type} & \vdash \mathsf{B}_2\ \mathsf{type} \\ \vdash \mathsf{s} : \mathsf{A}_2 & \vdash \mathsf{t} : \mathsf{B}_2 & \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 & \vdash \mathsf{B}_1 \equiv \mathsf{B}_2\end{array}}{\vdash \mathsf{fst}(\mathsf{A}_1, \mathsf{B}_1, \mathsf{pair}(\mathsf{A}_2, \mathsf{B}_2, \mathsf{s}, \mathsf{t})) \equiv \mathsf{s} : \mathsf{A}_1} \tag{13.1}$$

The new rule is inter-derivable with the original one: From (13.1) we can derive the original rule just by instantiating A_1 and A_2 with A and similarly for B. For the other way round the derivation uses congruence rules and TT-Conv-Tm.

Example 13.1.5 Consider the rule stating that the identity function is the neutral element for composition:

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{B}\ \mathsf{type} \qquad \vdash \mathsf{f} : \mathsf{A} \longrightarrow \mathsf{B}}{\vdash \mathsf{compose}(\mathsf{A}, \mathsf{B}, \mathsf{f}, \lambda(\mathsf{A}, \mathsf{A}, \{x\}x)) \equiv \mathsf{f} : \mathsf{A} \longrightarrow \mathsf{B}}$$

The left-hand side of the conclusion is not a pattern because $\lambda(\mathsf{A}, \mathsf{A}, \{x\}x)$ is not a pattern. Once again we can remedy the situation by introducing an additional equational premise:

$$\frac{\begin{array}{cc} \vdash \mathsf{A}\ \mathsf{type} & \vdash \mathsf{B}\ \mathsf{type} \qquad \vdash \mathsf{f} : \mathsf{A} \longrightarrow \mathsf{B} \\ \vdash \mathsf{i} : \mathsf{A} \longrightarrow \mathsf{A} & \vdash \mathsf{i} \equiv \lambda(\mathsf{A}, \mathsf{A}, \{x\}x) : \mathsf{A} \longrightarrow \mathsf{A} \end{array}}{\vdash \mathsf{compose}(\mathsf{A}, \mathsf{B}, \mathsf{f}, i) \equiv \mathsf{f} : \mathsf{A} \longrightarrow \mathsf{B}}$$

At the time of writing this thesis the Andromeda 2 implementation of the equality checking algorithm does not yet support automatic generation of linearized versions of rules. However, in the common cases the user can apply the described technique by hand.

It is clear that the technique for linearizing rules to become patterns works generally.

Proposition 13.1.6 If $\Xi \Longrightarrow \mathcal{b}\boxed{p}$ is a rule such that p is a pattern and $\mathsf{mv}(p) = |\Xi|_{\mathsf{obj}}$ then the rule is deterministic.

Proof. Consider a judgement $\Theta; \Gamma \vdash \mathcal{b}'\boxed{e}$, and instantiations J and K of Ξ over $\Theta; \Gamma$ such that $J_* p = K_* p = e$. Then J and K agree on object metavariables because they all appear in p, and on equational metavariables because they must, as they can only map them to $\star$. □

Reminder: object metavariables

The notation $|\Xi|_{\mathsf{obj}}$ gives the set of object metavariables of Ξ.

We shall use patterns to find matching instantiations, when they exist. For this purpose we define the following notation.

Definition 13.1.7 Given Ξ, a pattern p over Ξ such that $\mathsf{mv}(p) = |\Xi|_{\mathsf{obj}}$, and an expression e over $\Theta; \Gamma$, we write

$$\Xi \vdash p \triangleright t \rightsquigarrow I \qquad \text{and} \qquad \Xi \vdash p \triangleright t \not\rightsquigarrow$$

respectively when I is an instantiation of Ξ over $\Theta; \Gamma$ such that $I_* p = t$, and when there is no such instantiation.

The reader should convince themselves that there is an obvious algorithm that computes from Ξ, p and t the unique I such that $\Xi \vdash p \triangleright t \rightsquigarrow I$, or decides that it does not exist.

13.2. Object-invertible rules

While patterns provide a syntactic criterion for deterministic rules, we still need to consider derivability of pattern-induced instantiations.

Rules are used not only to derive judgements, but also to *invert* derivable judgements to their premises, for the purpose of analyzing them. For example, when a term is normalized, we decide what steps to take by observing its structure, which amounts to applying an inversion principle, such as Theorem 5.2.2. In general, we may invert a derivable judgement $\Theta; \Gamma \vdash \mathcal{j}'$ using a rule

$$R = (\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathcal{j})$$

by finding a derivable instantiation $I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$ of its premises over $\Theta; \Gamma$ such that $I_* \jmath = \jmath'$. If I is found, the judgement can be derived using the instantiation $I_* R$,

$$\frac{\Theta; \Gamma \vdash (I_{(k)*} \mathscr{B}_k) \boxed{e_k} \text{ for } k = 1, \ldots, n}{\Theta; \Gamma \vdash I_* \jmath}$$

Under favorable conditions, it may happen that some of the above premises are known to be derivable ahead of time, so there is no need to rederive them. We are particularly interested in the case where all the object premises are of this kind. To be able to express that a judgement is derivable "modulo equalities" we need to explain in what metacontext such a judgement appears, as the equational metavariables need to remain in the metacontext. The following definition makes that precise.

Definition 13.2.1 Let $\Xi = [\mathsf{M}_1 : \mathscr{B}_1, \ldots, \mathsf{M}_n : \mathscr{B}_n]$ be a metacontext whose equational metavariables are $\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}$. Given an instantiation I of Ξ over $\Theta; []$ such that $|\Xi| \cap |\Theta| = \emptyset$, the *equational residue* Ξ / I is the metacontext

$$\Xi / I = [\Theta, \mathsf{M}_{i_1} : I_{(i_1)*} \mathscr{B}_{i_1}, \ldots, \mathsf{M}_{i_m} : I_{(i_m)*} \mathscr{B}_{i_m}].$$

The *residual instantiation* I^r of Ξ over Ξ / I and $[]$ is defined by

$$I^r(\mathsf{M}_i) = \begin{cases} I(\mathsf{M}_i) & \text{if } \mathsf{M}_i \in |\Xi|_{\mathsf{obj}}, \\ \widehat{\mathsf{M}_i} & \text{otherwise.} \end{cases}$$

The condition that $|\Xi| \cap |\Theta| = \emptyset$ is inessential, as we can always rename metavariables accordingly. Without loss of generality the instantiation I is over an empty variable context, because the variable context can be promoted to metacontext using Proposition 4.3.6.

With the definition of equational residues in hand we can finally define the kinds of rules that ensure derivability of object premises when pattern-matching.

Definition 13.2.2 In a raw type theory, a derivable raw rule $R = (\Xi \Longrightarrow \jmath)$ is *object-invertible* when the following holds: whenever I instantiates Ξ over $\Theta; []$, with $\vdash \Theta$ mctx and $|\Xi| \cap |\Theta| = \emptyset$, if $\Theta; [] \vdash I_* \jmath$ is derivable then so is the residual instantiation I^r.

Let us explain how object-invertible rules shall be used. Suppose $\Xi \Longrightarrow s : A$ is object-invertible, $\Xi \Longrightarrow s \equiv t : A$ is derivable, I instantiates Ξ over $\Theta; \Gamma$, and $\Theta; \Gamma \vdash I_* s : I_* A$ is a given derivable judgement. We would like to derive $\Theta; \Gamma \vdash I_* s \equiv I_* t : I_* A$ so that we may rewrite $I_* s$ to $I_* t$. Thus we must verify that I is derivable. By object-invertibility its object premises are derivable, so we only need to check its equational ones. The following proposition ensures that such a procedure is valid.

Proposition 13.2.3 Consider an object-invertible rule $\Xi \Longrightarrow \mathcal{J}$ and an instantiation I over $\Theta;\Gamma$, such that $\Theta;\Gamma \vdash I_*\mathcal{J}$ is derivable. Then I is derivable if, for every equational boundary $\mathcal{B} = \{\vec{x}{:}\vec{A}\}\ \mathcal{b}$ in Ξ, the judgement $\Theta;\Gamma \vdash (I_*\mathcal{B})\boxed{\{\vec{x}\}\star}$ is derivable.

Proof. Let J be the promotion of I to (Θ,Γ) and the empty variable context. Because the rule is object-invertible, J^r is derivable. Next, we promote each judgement from the statement to

$$(\Theta,\Gamma);[] \vdash (J_*\mathcal{B})\boxed{\{\vec{x}\}\star}. \tag{13.2}$$

and observe that $J = K \circ J^r$, where K is the instantiation of Ξ/J over (Θ,Γ) and $[]$ defined by

$$K(\mathsf{M}) = \begin{cases} \widehat{\mathsf{M}} & \text{if } \mathsf{M} \in |(\Theta,\Gamma)|, \\ \{\vec{x}\}\star & \text{otherwise.} \end{cases}$$

Because J^r is derivable, and K is derivable thanks to derivability of judgements (13.2), it follows that J is derivable. Therefore, I is derivable too. □

Example 13.2.4 Let us demonstrate how equational residues are going to be used in rewriting. Suppose we have derived

$$\Theta;[] \vdash \mathsf{fst}(U_1, V_1, \mathsf{pair}(U_2, V_2, u, v)) : U_1 \tag{13.3}$$

and would like to apply the β-rule (13.1) to it, i.e., we would like to establish

$$\Theta;[] \vdash \mathsf{fst}(U_1, V_1, \mathsf{pair}(U_2, V_2, u, v)) \equiv u : U_1 \tag{13.4}$$

First, using Theorem 5.1.6, we extract from (13.1) the derivability of its left-hand side

$$\frac{\begin{array}{cccc} \vdash \mathsf{A}_1\ \mathsf{type} & \vdash \mathsf{A}_2\ \mathsf{type} & \vdash \mathsf{B}_1\ \mathsf{type} & \vdash \mathsf{B}_2\ \mathsf{type} \\ \vdash \mathsf{s} : \mathsf{A}_2 & \vdash \mathsf{t} : \mathsf{B}_2 & \vdash \mathsf{A}_1 \equiv \mathsf{A}_2\ \mathsf{by}\ \zeta & \vdash \mathsf{B}_1 \equiv \mathsf{B}_2\ \mathsf{by}\ \xi \end{array}}{\vdash \mathsf{fst}(\mathsf{A}_1, \mathsf{B}_1, \mathsf{pair}(\mathsf{A}_2, \mathsf{B}_2, \mathsf{s}, \mathsf{t})) : \mathsf{A}_1} \tag{13.5}$$

where we labeled the equational premises with metavariables ζ and ξ. We may compare (13.5) with (13.3) to get a matching instantiation

$$I = \langle \mathsf{A}_1 \mapsto U_1, \mathsf{A}_2 \mapsto U_2, \mathsf{B}_1 \mapsto V_1, \mathsf{B}_2 \mapsto V_2, \mathsf{s} \mapsto u, \mathsf{t} \mapsto v, \zeta \mapsto \star, \xi \mapsto \star \rangle$$

of its premises over $\Theta;[]$. Now it would be a mistake to simply instantiate (13.1) with I because the equational premises ζ and ξ may not be derivable (the object premises are derivable by Theorem 5.1.6). However, because (13.5) is object-invertible by Corollary 13.2.11, proved below, the residual instantiation

$$I^r = \langle \mathsf{A}_1 \mapsto U_1, \mathsf{A}_2 \mapsto U_2, \mathsf{B}_1 \mapsto V_1, \mathsf{B}_2 \mapsto V_2, \mathsf{s} \mapsto u, \mathsf{t} \mapsto v, \zeta \mapsto \zeta, \xi \mapsto \xi \rangle,$$

is derivable. Hence, we may instantiate (13.1) with I^r to derive

$$\Theta, \zeta{:}(U_1 \equiv U_2 \text{ by } \square), \xi{:}(V_1 \equiv V_2 \text{ by } \square); [] \vdash \mathsf{fst}(U_1, V_1, \mathsf{pair}(U_2, V_2, u, v)) \equiv u : U_1.$$

Thus we must still verify $\Theta; [] \vdash U_1 \equiv U_2$ and $\Theta; [] \vdash V_1 \equiv V_2$, in order to conclude (13.4), precisely as expected.

13.2.1. The natural for variables condition

Whether a rule is object-invertible depends not just on the rule itself, but on the ambient type theory too, for it may happen that $\Theta; \Gamma \vdash I_* \mathcal{J}$ is not derivable by the rule under consideration, but by another one that instantiates to the same conclusion.

Example 13.2.5 Consider the standard type theory whose specific rules are

$$\frac{}{\vdash \mathsf{0}\ \mathsf{type}} \qquad \frac{}{\vdash \mathsf{1}\ \mathsf{type}} \qquad \frac{}{\vdash \mathsf{u} : \mathsf{1}} \qquad \frac{\vdash \mathsf{v} : \mathsf{1}}{\vdash \mathsf{T(v)}\ \mathsf{type}} \qquad \frac{\vdash \mathsf{e} : \mathsf{0}}{\vdash \mathsf{0} \equiv \mathsf{1}}$$

The derivable object rule

$$\frac{\vdash \mathsf{e} : \mathsf{0}}{\vdash \mathsf{T(e)}\ \mathsf{type}}$$

is not object-invertible, because the instantiation $I = \langle \mathsf{e} \mapsto \mathsf{u} \rangle$ yields the derivable judgement $[]; [] \vdash \mathsf{T(u)}\ \mathsf{type}$, but $[]; [] \vdash \mathsf{u} : \mathsf{0}$ is not derivable.

We think of the type 0 as the empty type or falsehood. The problematic equation $0 \equiv 1$ then happens under the assumption that we have an element of the empty type, which is an expected behaviour in the usual formal systems.

In the previous example the culprit is the application of term conversion to a metavariable. As it turns out, such conversions of variables are the principal obstruction to object-invertibility, so we define a syntactic property of judgements which prevents them.

Definition 13.2.6 An object judgement $\Theta; \Gamma \vdash \mathcal{J}$ is *natural for variables* when the relation $\Theta; \Gamma \vdash^\natural \mathcal{J}$ can be deduced using the rules in Figure 13.2.

The point of this definition as the name suggests is that (meta)variables in the derivation have natural types which is summarized in the following proposition.

Reminder: Natural type

A natural type of a term is the one that can be read off the term expression as specified in Definition 5.2.1.

Proposition 13.2.7 A derivable object judgement is natural for variables has a derivation in which any application of TT-META and TT-VAR is *not* immediately followed by a conversion, unless it appears in a subderivation of an equality judgement.

Proof. The claim is established by a straightforward induction on the derivation of $\Theta; \Gamma \vdash^\natural \mathcal{J}$ with the help of Theorem 5.2.2. □

$$\frac{\Gamma(\mathsf{a}) = A}{\Theta;\Gamma \vdash^{\natural} \mathsf{a} : A} \qquad \frac{\mathsf{a} \notin |\Gamma| \qquad \Theta;\Gamma, \mathsf{a}{:}A \vdash^{\natural} \mathcal{J}[\mathsf{a}/x]}{\Theta;\Gamma \vdash^{\natural} \{x{:}A\}\mathcal{J}}$$

$$\frac{\Theta(\mathsf{M}) = (\{\vec{x}{:}\vec{A}\}\ \square\ \mathsf{type}) \qquad \Theta;\Gamma \vdash^{\natural} t_i : A[\vec{t}_{(i)}/\vec{x}_{(i)}] \quad \text{for } i = 1,\dots,n}{\Theta;\Gamma \vdash^{\natural} \mathsf{M}(t_1,\dots,t_n)\ \mathsf{type}}$$

$$\frac{\Theta(\mathsf{M}) = (\{\vec{x}{:}\vec{A}\}\ \square : B) \qquad \Theta;\Gamma \vdash^{\natural} t_i : A[\vec{t}_{(i)}/\vec{x}_{(i)}] \quad \text{for } i = 1,\dots,n}{\Theta;\Gamma \vdash^{\natural} \mathsf{M}(t_1,\dots,t_n) : B[\vec{t}/\vec{x}]}$$

$$\frac{\begin{array}{c}\text{Rule for } \mathsf{S} \text{ is } \mathsf{M}_1{:}\mathscr{B}_1,\dots,\mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{C}\boxed{\mathsf{S}(\widehat{M}_1,\dots,\widehat{M}_n)} \\ I = \langle \mathsf{M}_1 \mapsto e_1,\dots,\mathsf{M}_n \mapsto e_n\rangle \\ \Theta;\Gamma \vdash^{\natural} (I_{(i)*}\mathscr{B}_i)\boxed{e_i} \quad \text{if } \mathscr{B}_i \text{ is an object boundary}\end{array}}{\Theta;\Gamma \vdash^{\natural} \mathscr{C}'\boxed{\mathsf{S}(e_1,\dots,e_n)}}$$

Figure 13.2.: Object judgements that are natural for variables.

In the Example 13.2.5 the last derived rule is not natural for variables, while all the other specific object rules are.

13.2.2. Sufficient conditions for object-invertibility

The obvious pattern-matching algorithm scans a pattern and compares it to a term. It instantiates metavariables one by one and possibly out of order, which results in a chain of instantiations, each of which instantiates just one metavariable. Let us study such instantiations.

Definition 13.2.8 Let $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1,\dots,\mathsf{M}_n{:}\mathscr{B}_n]$ be a metacontext, and e an argument over $\Xi_{(k)}$ and the empty variable context with $\mathsf{ar}(e) = \mathsf{ar}(\mathscr{B}_k)$. The *basic instantiation* $\mathbb{I}(\Xi, \mathsf{M}_k, e)$ is defined by

$$\mathbb{I}(\Xi, \mathsf{M}_k, e)(\mathsf{M}_i) = \begin{cases} \widehat{\mathsf{M}}_i & \text{if } \mathsf{M}_k \neq \mathsf{M}_i, \\ e & \text{if } \mathsf{M}_k = \mathsf{M}_i. \end{cases} \tag{13.6}$$

It is an instantiation of Ξ over the metacontext

$$\mathbb{E}(\Xi, \mathsf{M}_k, e) = [\mathsf{M}_1{:}\mathscr{B}'_1,\dots,\mathsf{M}_{k-1}{:}\mathscr{B}'_{k-1},\mathsf{M}_{k+1}{:}\mathscr{B}'_{k+1},\dots,\mathsf{M}_n{:}\mathscr{B}'_n]$$

and the empty variable context, where $\mathscr{B}'_j = \mathbb{I}(\Xi, \mathsf{M}_k, e)_{(j)*}\mathscr{B}_j$.

Intuition behind a basic instantiation is that it instantiates just one metavariable and leaves the rest in place.

We can now state a sufficient condition for when a basic instantiation is derivable.

Lemma 13.2.9 A basic instantiation $\mathbb{I}(\Xi, \mathsf{M}_k, e)$ is derivable if $\vdash \Xi\ \mathsf{mctx}$ and $\Xi_{(k)} \vdash \mathscr{B}_k\boxed{e}$, in which case $\vdash \mathbb{E}(\Xi, \mathsf{M}_k, e)\ \mathsf{mctx}$ also holds.

Proof. For $i < k$, the judgement $\mathbb{E}(\Xi, \mathsf{M}_k, e) \vdash (\mathbb{I}(\Xi, \mathsf{M}_k, e)_{(i)*}\mathscr{B}_i)\boxed{\widehat{\mathsf{M}}_i}$ holds by abstraction and the metavariable rule TT-META, where we

invert $\vdash \Xi$ mctx to validate the abstractions.

The judgement $\mathbb{E}(\Xi, \mathsf{M}_k, e) \vdash (\mathbb{I}(\Xi, \mathsf{M}_k, e)_{(k)} * \mathscr{B}_k)\boxed{e}$ follows by weakening from $\Xi_{(k)} \vdash \mathscr{B}_k\boxed{e}$ because $\mathbb{E}(\Xi, \mathsf{M}_k, e)_{(k)} = \Xi_{(k)}$.

For $i > k$, we again use abstraction and the metavariable rule, where abstractions are now validated by inversion of $\vdash \Xi$ mctx and Theorem 5.1.4 applied to $\mathbb{I}(\Xi, \mathsf{M}_k, e)_{(i)}$.

The derivation of $\vdash \mathbb{E}(\Xi, \mathsf{M}_k, e)$ mctx has two parts. First, $\mathbb{E}(\Xi, \mathsf{M}_k, e)_{(k)}$ coincides with $\Xi_{(k)}$ and so we just reuse $\vdash \Xi_{(k)}$ mctx. For $i > k$, we derive $\mathbb{E}(\Xi, \mathsf{M}_k, e)_{(\mathsf{M}_i)} \vdash \mathscr{B}'_i$ as the instantiation of $\Xi_{(i)} \vdash \mathscr{B}_i$ by

$$\mathbb{I}(\Xi, \mathsf{M}_k, e)_{(i)} \in \mathrm{Inst}(\Xi_{(i)}, \mathbb{E}(\Xi, \mathsf{M}_k, e)_{(\mathsf{M}_i)}, [\,]),$$

which is observed to be derivable. □

We define particular compositions of chains of basic instantiations, as follows. Given a metacontext $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ and an instantiation

$$I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$$

of Ξ over $\Theta; [\,]$, define the instantiation

$$\mathbb{J}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}) \in \mathrm{Inst}(\Xi, \mathbb{F}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}), [\,])$$

and the metacontext $\mathbb{F}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m})$ by

$$\begin{aligned}
\mathbb{F}_{\Xi,\Theta,I}() &= \langle \Theta, \Xi \rangle \\
\mathbb{J}_{\Xi,\Theta,I}() &= \langle \mathsf{M}_1 \mapsto \widehat{\mathsf{M}}_1, \ldots, \mathsf{M}_n \mapsto \widehat{\mathsf{M}}_n \rangle \\
\mathbb{F}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_{m+1}}) &= \mathbb{E}(\mathbb{F}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}), \mathsf{M}_{i_{m+1}}, e_{i_{m+1}}) \\
\mathbb{J}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_{m+1}}) &= \mathbb{I}(\mathbb{F}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}), \mathsf{M}_{i_{m+1}}, e_{i_{m+1}}) \circ \\
&\qquad \mathbb{J}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m})
\end{aligned}$$

In the above definition we require $|\Xi| \cap |\Theta| = \emptyset$ and that $\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}$ are all distinct. We elide the subscripts and write $\mathbb{J}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m})$ and $\mathbb{F}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m})$ when no confusion can arise. A straightforward induction shows that

The condition $|\Xi| \cap |\Theta| = \emptyset$ is inessential due to a simple renaming of metavariables.

$$\mathbb{F}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m})(\mathsf{M}_j) = \mathbb{J}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}) * \mathscr{B}_j$$

for any $\mathsf{M}_j \in |\Xi| \setminus \{\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}\}$. The instantiation $\mathbb{J}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m})$ plays a role in proving object-invertibility, because

$$\{\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}\} = |\Xi|_{\mathsf{obj}}$$

implies

$$\mathbb{J}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}) = I^r \quad \text{and} \quad \mathbb{F}_{\Xi,\Theta,I}(\mathsf{M}_{i_1}, \ldots, \mathsf{M}_{i_m}) = \Xi / I.$$

We are now in possession of all the ingredients necessary to relate pattern matching and object-invertibility. Recall that, given a rule $\Xi \Longrightarrow \mathscr{b}\boxed{p}$ whose head is a pattern p and an instantiation I of Ξ over $\Theta; [\,]$, we would like to show that the residual instantiation I^r is derivable. The following lemma, whose purpose we explain shortly, is a stepping stone to accomplishing the goal.

Lemma 13.2.10 In a standard type theory, let $\Xi \Longrightarrow \mathscr{b}\boxed{p}$ be a derivable object rule which is natural for variables, p a pattern, and I an instantiation of $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ over $\Theta;[]$ such that $|\Xi| \cap |\Theta| = \emptyset$, and $\Theta;[] \vdash I_*(\mathscr{b}\boxed{p})$ is derivable.

Suppose $\vec{N} = (N_1, \ldots, N_m)$ is a sequence of distinct metavariables such that $\{N_1, \ldots, N_m\} \subseteq |\Xi|$, $\mathsf{mv}(\mathscr{b}) \subseteq \{N_1, \ldots, N_m\} \cup \mathsf{mv}(p)$, and both $\vdash \mathbb{F}_{\Theta,\Xi,I}(\vec{N})$ mctx and $\mathbb{J}_{\Theta,\Xi,I}(\vec{N})$ are derivable. Then $\vec{N}$ can be extended to a sequence of distinct metavariables $\vec{N}' = (N_1, \ldots, N_\ell)$ such that $\{N_1, \ldots, N_\ell\} = \{N_1, \ldots, N_m\} \cup \mathsf{mv}(p)$, and both $\vdash \mathbb{F}(\vec{N}')$ mctx and $\mathbb{J}(\vec{N}')$ are derivable.

Let us explain how the lemma shall be used. As noted above, I^r coincides with $\mathbb{J}(\vec{N})$ when $\vec{N}$ lists all of $|\Xi|_{\mathsf{obj}}$. Therefore, we may establish derivability of I^r by starting with $\vec{N} = ()$ and extending it with metavariables until it encompasses $|\Xi|_{\mathsf{obj}}$. The lemma guarantees that one such extension step can be done with the aid of patterns in a way that preserves derivability of $\mathbb{J}(\vec{N})$.

Proof. Let $I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$. We proceed by induction on the structure of p, and elide the subscripts to keep the notation shorter.

Case $p = \mathsf{M}_k$, $\mathscr{B}_k = (\square\ \mathsf{type})$, *and* $\mathscr{b} = (\square\ \mathsf{type})$: If M_k appears in $\vec{N}$ we let $\ell = m$ and we are done. Otherwise we set $\ell = m+1$ and $N_{m+1} = \mathsf{M}_k$. Because composition of derivable instantiations is derivable, we only need to show that $\mathbb{I}(\mathbb{F}(\vec{M}), \mathsf{M}_k, e_k)$ is derivable, which by Lemma 13.2.9 reduces to

$$\mathbb{F}(\vec{N})_{(\mathsf{M}_k)}; [] \vdash (\mathbb{J}(\vec{N})_* \mathscr{B}_k)\boxed{e_k},$$

which equals

$$\mathbb{F}(\vec{N})_{(\mathsf{M}_k)}; [] \vdash e_k\ \mathsf{type}.$$

It is derivable by weakening from the assumption $\Theta;[] \vdash e_k\ \mathsf{type}$.

By definition of patterns p cannot be a term metavariable, so there is no separate case for that in the proof.

Case $p = \mathsf{S}(q_1, \ldots, q_m)$: Suppose the symbol rule for S is

$$\mathsf{M}'_1{:}\mathscr{B}'_1, \ldots, \mathsf{M}'_j{:}\mathscr{B}'_j \Longrightarrow \mathscr{b}'\boxed{\mathsf{S}(\widehat{\mathsf{M}'}_1, \ldots, \widehat{\mathsf{M}'}_j)}.$$

By applying Corollary 5.3.2 to $\Xi;[] \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{q})}$ and letting the instantiation $K = [\mathsf{M}'_1 \mapsto q_1, \ldots, \mathsf{M}'_j \mapsto q_j]$, we obtain for $i = 1, \ldots, j$ derivations of

$$\Xi;[] \vdash (K_{(i)*}\mathscr{B}'_i)\boxed{q_i}. \tag{13.7}$$

Similarly, from derivability of $\Theta;[] \vdash (I_*\mathscr{b})\boxed{\mathsf{S}(I_*\vec{q})}$ we obtain derivability of

$$\Theta;[] \vdash ((I_*K)_{(i)*}\mathscr{B}'_i)\boxed{I_*q_i}, \tag{13.8}$$

which is equal to

$$\Theta;[] \vdash I_*((K_{(i)*}\mathscr{B}'_i)\boxed{q_i}). \tag{13.9}$$

We define $\vec{L}_0, \ldots, \vec{L}_j$ such that $\vec{L}_0 = \vec{N}$, and for $i = 1, \ldots, j$, the sequence $\vec{L}_i$ extends $\vec{L}_{i-1}$ by $\mathsf{mv}(q_i)$, and both $\vdash \mathbb{F}(\vec{L}_i)$ mctx and $\mathbb{J}(\vec{L}_i)$ are derivable. We may then finish the proof by taking $\vec{N}' = \vec{L}_j$. Assuming $\vec{L}_{i-1}$ has been constructed, we consider two cases.

First, if q_i is a non-abstracted object pattern then we obtain $\vec{L}_i$ by applying the induction hypothesis to (13.7), (13.9) and $\vec{L}_{i-1}$. We may do so because $\mathsf{mv}(K_{(i)*}\mathscr{B}'_i) \subseteq \mathsf{mv}(q_1) \cup \cdots \cup \mathsf{mv}(q_{i-1})$, which is contained in $\vec{L}_{i-1}$.

Second, if $q_i = \{\vec{x}\}\mathsf{M}_k(\vec{x})$ we proceed as follows. If M_k appears in $\vec{L}_{i-1}$, we take $\vec{L}_i = \vec{L}_{i-1}$ and we are done. Otherwise, we take $\vec{L}_i = (\vec{L}_{i-1}, \mathsf{M}_k)$. We need to show derivability of $\vdash \mathbb{F}(\vec{L}_i)\ \mathsf{mctx}$ and $\mathbb{J}(\vec{L}_i)$. Because $\mathbb{J}(\vec{L}_i) = \mathbb{I}(\mathbb{F}(\vec{L}_{i-1}), \mathsf{M}_k, e_k) \circ \mathbb{J}(\vec{L}_{i-1})$ and $\mathbb{J}(\vec{L}_{i-1})$ is derivable it suffices to show that $\mathbb{I}(\mathbb{F}(\vec{L}_{i-1}), \mathsf{M}_k, e_k)$ is derivable, and therefore by Lemma 13.2.9 that

$$\mathbb{F}(\vec{L}_{i-1})_{(\mathsf{M}_k)}; [] \vdash (\mathbb{J}(\vec{L}_{i-1})_*\mathscr{B}_k)\boxed{e_k}. \tag{13.10}$$

We claim that (13.10) is just a weakening of (13.8). Obviously, $\mathbb{F}(\vec{L}_{i-1})_{(\mathsf{M}_k)}$ extends Θ and $I_*q_i = e_k$. It remains to be seen that $\mathbb{J}(\vec{L}_{i-1})_*\mathscr{B}_k$ and $(I_*K)_{(i)*}\mathscr{B}'_i$ are the same. The judgement (13.7) equals

$$\Xi; [] \vdash (K_{(i)*}\mathscr{B}'_i)\boxed{\{\vec{x}\}\mathsf{M}_k(\vec{x})}.$$

By the naturality-for-variables assumption it is derivable without conversions, which is only possible if $K_{(i)*}\mathscr{B}'_i$ is $\mathscr{B}_k$. Therefore,

$$(I_*K)_{(i)*}\mathscr{B}'_i = I_*(K_{(i)*}\mathscr{B}'_i) = \mathbb{J}(\vec{L}_{i-1})_*(K_{(i)*}\mathscr{B}'_i) = \mathbb{J}(\vec{L}_{i-1})_*\mathscr{B}_k,$$

where the second step is valid because $\mathsf{mv}(K_{(i)*}\mathscr{B}'_i) \subseteq \mathsf{mv}(q_1) \cup \cdots \cup \mathsf{mv}(q_{i-1})$, which is contained in $\vec{L}_{i-1}$. □

We can now finally state the sufficient syntactic conditions for a rule to be object-invertible.

Corollary 13.2.11 In a standard type theory, consider a derivable finitary object rule $\Xi \Longrightarrow \mathscr{C}\boxed{p}$ which is natural for variables. If p is a pattern and $\mathsf{mv}(p) = |\Xi|_{\mathsf{obj}}$ then the rule is object-invertible.

Proof. Consider an instantiation I of Ξ over $\Theta; []$, such that $\vdash \Theta\ \mathsf{mctx}$ and $\Theta; [] \vdash (I_*\mathscr{C})\boxed{I_*p}$ are derivable. Without loss of generality we may assume $|\Xi| \cap |\Theta| = \emptyset$.

We apply Lemma 13.2.10 with the empty sequence $\vec{N} = ()$, noting that $\mathsf{mv}(\mathscr{C}) \subseteq \mathsf{mv}(p)$, that $\mathbb{F}() = \langle\Theta, \Xi\rangle$ and that $\vdash \langle\Theta, \Xi\rangle\ \mathsf{mctx}$ is derivable because the rule is finitary and we assumed $\vdash \Theta\ \mathsf{mctx}$. This way we obtain a sequence $\vec{N}' = (N'_1, \ldots, N'_\ell)$ such that $\mathsf{mv}(p) = \{N'_1, \ldots, N'_\ell\}$ and $\mathbb{J}(\vec{N}')$ is derivable. Because $\mathsf{mv}(p) = |\Xi|_{\mathsf{obj}}$, it follows that $\mathbb{J}(\vec{N}')$ coincides with I^r, hence it is derivable too. □

14. Computation and Extensionality Rules

14.1. Computation rules

The equality checking algorithm uses two kinds of equational rules, which we describe here and prove that they have the desired properties. First, we have the rules that govern normalization.

Definition 14.1.1 A derivable finitary rule $\Theta \Longrightarrow A \equiv B$ is a *type computation rule* if $\Theta \Longrightarrow A$ type is deterministic and object-invertible.

Definition 14.1.2 A derivable finitary rule $\Theta \Longrightarrow u \equiv v : A$ is a *term computation rule* if u is a term symbol application and the rule $\Theta \Longrightarrow u : \tau_{\Theta;[]}(u)$ is deterministic and object-invertible.

The reason behind the first condition in the definition of a term computation rule is that for term symbol applications Proposition 5.3.1 holds, which is needed in the proof of soundness (Theorem 15.3.1). We exhibit in Example 14.2.4 what can go wrong if we allow for a metavariable as the lefthand-side of the equation. One might hope that the second condition in Definition 14.1.2 could be relaxed to $\Theta \Longrightarrow u : A$. However, the additional flexibility is only apparent, for if a term has a type then it has the natural type as well. In any case, in the proofs of soundness (Theorem 15.3.1, Theorem 15.3.2) we rely on having the natural type.

A computation rule may be recognized using the following criterion.

Proposition 14.1.3 In a standard type theory:

1. A derivable finitary rule $\Xi \Longrightarrow P \equiv B$ is a type computation rule if P is a type pattern, $\mathsf{mv}(P) = |\Xi|_{\mathsf{obj}}$, and $\Xi \Longrightarrow P$ type is natural for variables.
2. A derivable finitary rule $\Xi \Longrightarrow p \equiv v : A$ is a term computation rule if p is a term pattern, $\mathsf{mv}(p) = |\Xi|_{\mathsf{obj}}$, and $\Xi \Longrightarrow p : \tau_{\Xi;[]}(p)$ is natural for variables.

The syntactic criterion for computation rules is of course derived from the syntactic criterion for object-derivable rules from Corollary 13.2.11.

Proof. To prove the claims, observe that $\Xi \Longrightarrow P$ type is derivable by Theorem 5.1.6, and $\Xi \Longrightarrow p : \tau_{\Xi;[]}(p)$ by Theorem 5.1.6 and Corollary 5.3.3. Observe also that $\mathsf{mv}(P) = |\Xi|_{\mathsf{obj}}$ and $\mathsf{mv}(p) = |\Xi|_{\mathsf{obj}}$. Then apply Proposition 13.1.6 and Corollary 13.2.11 respectively to $\Xi \Longrightarrow P$ type and to $\Xi \Longrightarrow p : \tau_{\Xi;[]}(p)$. □

Example 14.1.4 Typical β-rules satisfy the conditions of Proposi-

tion 14.1.3, after their left-hand sides have been linearized, as in Example 13.1.4. Another example is the β-rule for application

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \{x{:}\mathsf{A}\}\ \mathsf{B}\ \mathsf{type} \qquad \vdash \{x{:}\mathsf{A}\}\ \mathsf{s} : \mathsf{B}(x) \qquad \vdash \mathsf{t} : \mathsf{A}}{\vdash \mathsf{apply}(\mathsf{A}, \{x\}\mathsf{B}(x), \lambda(\mathsf{A}, \{x\}\mathsf{B}(x), \{x\}\mathsf{s}(x)), \mathsf{t}) \equiv \mathsf{s}(\mathsf{t}) : \mathsf{B}(\mathsf{t})}$$

whose linearized form is

$$\frac{\begin{array}{cc} \vdash \mathsf{A}_1\ \mathsf{type} & \vdash \{x{:}\mathsf{A}_1\}\ \mathsf{B}_1\ \mathsf{type} \\ \vdash \mathsf{A}_2\ \mathsf{type} & \vdash \{x{:}\mathsf{A}_2\}\ \mathsf{B}_2\ \mathsf{type} \\ \multicolumn{2}{c}{\vdash \{x{:}\mathsf{A}_2\}\ \mathsf{s} : \mathsf{B}_2(x) \qquad \vdash \mathsf{t} : \mathsf{A}_1} \\ \multicolumn{2}{c}{\vdash \mathsf{A}_1 \equiv \mathsf{A}_2 \qquad \vdash \{x{:}\mathsf{A}_1\}\mathsf{B}_1(x) \equiv \mathsf{B}_2(x)} \end{array}}{\vdash \mathsf{apply}(\mathsf{A}_1, \{x\}\mathsf{B}_1(x), \lambda(\mathsf{A}_2, \{x\}\mathsf{B}_2(x), \{x\}\mathsf{s}(x)), \mathsf{t}) \equiv \mathsf{s}(\mathsf{t}) : \mathsf{B}_1(\mathsf{t})}$$

which satisfies Proposition 14.1.3.

Example 14.1.5 We also allow the somewhat unusual rule

$$\frac{}{\vdash \mathsf{U}\ \mathsf{type}} \qquad \frac{\vdash \mathsf{A}\ \mathsf{type}}{\vdash \mathsf{A} \equiv \mathsf{U}}$$

because it allows us to dispense with all questions about equality of types in case we want to work with an uni-typed theory (some would call it untyped).

Example 14.1.6 A class of rules that qualify as computational rules are also the specific equality rules added in definitional extension from Example 9.3.5. The left-hand side of the equality rule

$$\frac{\vdash \mathscr{B}_i\boxed{\mathsf{M}_i} \quad \text{for } i = 1, \ldots, n}{\vdash \mathscr{C}\boxed{\mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n}) \equiv e}}$$

constitutes a pattern because it is just a generically applied symbol and the derivation is natural for variables as it is just an application of the specific rule. Thus when extending a type theory with new definitions, we can always compute the newly defined symbol by adding the definitional equality to the set of computation rules of the equality checking algorithm.

14.2. Extensionality rules

The second kind of rules is used by the algorithm to reduce an equation to subordinate equations by matching its type.

Definition 14.2.1 An *extensionality rule* is a derivable finitary rule of the form

$$\Theta, \mathsf{s}{:}(\Box : A), \mathsf{t}{:}(\Box : A), \Phi \Longrightarrow \mathsf{s} \equiv \mathsf{t} : A$$

such that Φ contains only equational premises, and $\Theta \Longrightarrow A$ type is deterministic and object-invertible.

An extensional rule may be recognized with the following criterion.

Proposition 14.2.2 In a standard type theory, a derivable finitary rule of the form

$$\Xi, \mathsf{s}{:}(\square : P), \mathsf{t}{:}(\square : P), \Phi \Longrightarrow \mathsf{s} \equiv \mathsf{t} : P$$

is an extensionality rule if Φ contains only equational premises, P is a type pattern, $\mathsf{mv}(P) = |\Xi|_{\mathsf{obj}}$, and $\Xi \Longrightarrow P$ type is natural for variables.

Just like for computatuon rules, the syntactic criterion for extensionality rules is derived from the syntactic criterion for object-derivable rules from Corollary 13.2.11.

Proof. Apply Proposition 13.1.6 and Corollary 13.2.11 to $\Xi \Longrightarrow P$ type. □

Extensionality rules that one finds in practice typically satisfy the above syntactic condition, even without linearization. Here are a few.

Example 14.2.3 Extensionality rules typically state that elements of a type are equal when their parts are equal. For example, extensionality for simple products states that pairs are equal if their components are equal:

$$\frac{\begin{array}{c}\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{B}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \times \mathsf{B} \qquad \vdash \mathsf{t} : \mathsf{A} \times \mathsf{B} \\ \vdash \mathsf{fst}(\mathsf{A},\mathsf{B},\mathsf{s}) \equiv \mathsf{fst}(\mathsf{A},\mathsf{B},\mathsf{t}) : \mathsf{A} \qquad \vdash \mathsf{snd}(\mathsf{A},\mathsf{B},\mathsf{s}) \equiv \mathsf{snd}(\mathsf{A},\mathsf{B},\mathsf{t}) : \mathsf{B}\end{array}}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A} \times \mathsf{B}} \tag{14.1}$$

Similarly, the extensionality rule for dependent functions states that they are equal if their generic applications are equal:

$$\frac{\begin{array}{c}\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \{x{:}\mathsf{A}\}\ \mathsf{B}\ \mathsf{type} \\ \vdash \mathsf{s} : \Pi(\mathsf{A}, \{x\}\mathsf{B}(x)) \qquad \vdash \mathsf{t} : \Pi(\mathsf{A}, \{x\}\mathsf{B}(x)) \\ \vdash \{x{:}\mathsf{A}\}\ \mathsf{apply}(\mathsf{A}, \{x\}\mathsf{B}(x), \mathsf{s}, \mathsf{x}) \equiv \mathsf{apply}(\mathsf{A}, \{x\}\mathsf{B}(x), \mathsf{t}, \mathsf{x}) : \mathsf{B}(x)\end{array}}{\vdash \mathsf{s} \equiv \mathsf{t} : \Pi(\mathsf{A}, \{x\}\mathsf{B}(x))}$$

The above is not to be confused with *propositional* function extensionality, which is a certain term that maps point-wise propositional equality of functions to their propositional equality.

Example 14.2.4 Some extensionality rules have no equational premises. The first one that comes to mind is the rule stating that all elements of the unit type are equal:

$$\frac{\vdash \mathsf{s} : \mathsf{unit} \qquad \vdash \mathsf{t} : \mathsf{unit}}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{unit}}$$

The corresponding η-rule ($\star$ is the canonical inhabitant of unit)

$$\frac{\vdash t : \mathsf{unit}}{\vdash t \equiv \star}$$

cannot be incorporated as a computation rule naively because the bare metavariable on the left-hand side matches any term, even if its type is not (judgementally equal to) unit. Since our normalization procedure in Section 15.1 does not check for equality of types separately, such rules do not behave well as computation rules. Another rule of this kind is the judgemental variant of Uniqueness of identity proofs (UIP) which equates any two proofs of a propositional identity:

$$\frac{\vdash A\ \mathsf{type} \qquad \vdash a : A \qquad \vdash b : A \qquad \vdash p : \mathsf{Id}(A, a, b) \qquad \vdash q : \mathsf{Id}(A, a, b)}{\vdash p \equiv q : \mathsf{Id}(A, a, b)}$$

The corresponding η-rule is as troublesome as the one for unit:

$$\frac{\vdash A\ \mathsf{type} \qquad \vdash a : A \qquad \vdash p : \mathsf{Id}(A, a, a)}{\vdash p \equiv \mathsf{refl}(A, a) : \mathsf{Id}(A, a, a)}$$

The principle has been used, for example, in the cubical type theory XTT for Bishop sets [135].

[135]: Sterling et al. (2021), *A Cubical Language for Bishop Sets*

Here is one last example:

$$\frac{\vdash A\ \mathsf{type}}{\vdash \|A\|\ \mathsf{type}} \qquad \frac{\vdash A\ \mathsf{type} \qquad \vdash t : A}{\vdash |t| : \|A\|}$$

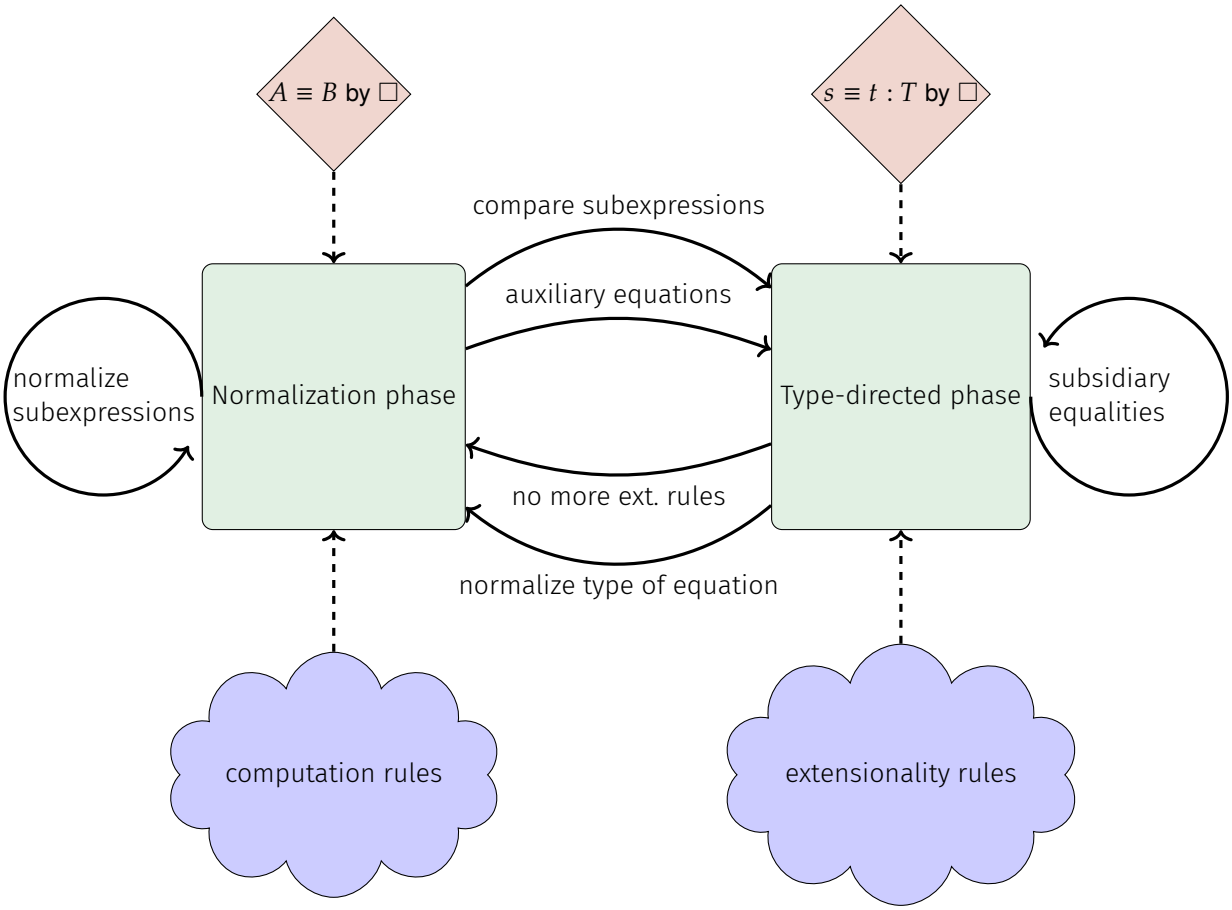
$$\frac{\vdash A\ \mathsf{type} \qquad \vdash u : \|A\| \qquad \vdash v : \|A\|}{\vdash u \equiv v : \|A\|}$$

The above rules describe a kind of "judgemental truncation", which is like the propositional truncation from homotopy type theory, except that it equates all terms of $\|A\|$ judgementally. It is unclear what elimination rule of judgemental truncation would be, but one is reminded of the *proof-irrelevant propositions* [58].

[58]: Gilbert et al. (2019), "Definitional proof-irrelevance without K"

15. The algorithm

The equality checking algorithm, precisely stated in Section 15.2, has two phases: a *type-directed phase* for applying extensionality rules, intertwined with a *normalization phase* based on computation rules. The rough outline of the algorithm is in the following diagram.



The algorithm takes as input a derivable equality boundary and either outputs a derivable equality judgement whose boundary is the one given as input, or fails to find a derivation. If the input boundary is a type equality boundary, the algorithm goes directly to the normalization phase which is parametrized by the computation rules and *principal arguments* defined in Section 15.1. For a term equality boundary the algorithm first goes to the type-directed phase which is parametrized by the extensionality rules.

In this chapter we specify the algorithm (Section 15.1, Section 15.2), prove that it is sound in Section 15.3 and discuss some of the limits and design choices in Section 15.4.

15.1. Principal arguments and normalization

Normalization rewrites an expression $\mathsf{S}(e_1, \ldots, e_n)$ by normalizing some of the arguments $e_1, \ldots, e_n$, applying a computation rule, and repeating the process. We say that an argument e_i (or more precisely, its position i) is *principal* for S if it is so normalized. By varying the selection of principal arguments we may control the algorithm to compute various kinds of normal form. For example, in λ-calculus the weak-head normal form is obtained when the only principal argument is the head of an application, while taking all arguments to be principal leads to the strong normal form. Our algorithm is flexible in this regard, as it takes the information about principality of arguments as input. In Section 15.4.1 we discuss how appropriate principal arguments can be chosen.

In specific cases normal forms are characterized by their syntactic structure, for example a normal form in the λ-calculus is an expression without β-redeces. One then proves that the normalization procedure always leads to a normal form. We are faced with a general situation in which no such syntactic characterization is available. Luckily, the algorithm never needs to recognize normal forms, although we do have to keep track of which expressions have already been subjected to the normalization procedure, so that we avoid normalizing them again.

Describing neutral forms is another approach to recognise which arguments need not be normalized. We briefly discuss this option in Section 15.4.

Normalization is parametrized by the following data:

1. a standard type theory T,
2. a family $\mathscr{C}$ of computation rules for T (Definition 14.1.1, Definition 14.1.2),
3. for each symbol S taking k arguments, a set $\wp(\mathsf{S}) \subseteq \{1, \ldots, k\}$ of its *principal arguments*,

It has three interdependent variations:

$\Theta; \Gamma \vdash \mathscr{B} \boxed{e \triangleright e'}$	normalize argument e to e',
$\Theta; \Gamma \vdash \mathscr{C} \boxed{\mathsf{S}(\vec{e}) \triangleright_{\mathrm{p}} \mathsf{S}(\vec{e}')}$	normalize the principal arguments of S,
$\Theta; \Gamma \vdash \mathscr{C} \boxed{e \triangleright_{\mathrm{c}} e'}$	use a computation rule to rewrite e to e'.

Specifically,

$$\Theta; \Gamma \vdash (A \triangleright A')\ \mathsf{type} \qquad \text{and} \qquad \Theta; \Gamma \vdash t \triangleright t' : B$$

respectively express the facts that the type A normalizes to A' and the term t to t'. Figure 15.1 specifies the normalization procedure. Note that normalization is mutually recursive with equality checking, because the rule for $\triangleright_{\mathrm{c}}$ resolves equational premises using equational checking from Figure 15.2. We omitted the clauses for metavariable applications, as they are analogous to symbol applications. That is, for the purposes of normalization and equality checking, an object metavariable M with boundary $\mathscr{B}$ and arity $\mathsf{ar}(\mathscr{B}) = (c, n)$ is construed as a primitive symbol of syntactic class c taking n term arguments.

Normalization of arguments is syntax-directed and deterministic, and so is normalization of principal arguments. However, the applications

of computation rules need not terminate, and the computation rules may be a source of non-determinism when several apply to the same expression. We discuss strategies for dealing with these issues in Section 15.4.2.

$$\frac{(\Xi \Longrightarrow \mathscr{b}'\boxed{p \equiv v}) \in \mathscr{C} \qquad \Xi \vdash p \triangleright s \rightsquigarrow I \qquad \Theta;\Gamma \vdash I_*(\mathscr{B}\boxed{e \sim e'}) \quad \text{for } (\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e' \text{ by } \square}) \in \Xi}{\Theta;\Gamma \vdash \mathscr{b}\boxed{s \triangleright_{\mathrm{c}} I_* v}}$$

$$\frac{\text{Rule for } \mathsf{S} \text{ is } \mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{b}'\boxed{\mathsf{S}(\widehat{\mathsf{M}}_1, \ldots, \widehat{\mathsf{M}}_n)} \qquad \Theta;\Gamma \vdash (\langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_{i-1} \mapsto e_{i-1}\rangle \mathscr{B}_i)\boxed{e_i \triangleright e_i'} \quad \text{if } i \in \wp(\mathsf{S}) \qquad e_i = e_i' \quad \text{if } i \notin \wp(\mathsf{S})}{\Theta;\Gamma \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{e}) \triangleright_{\mathrm{p}} \mathsf{S}(\vec{e}')}}$$

$$\frac{\Theta;\Gamma \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{e}) \triangleright_{\mathrm{p}} \mathsf{S}(\vec{e}')} \qquad \Theta;\Gamma \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{e}') \triangleright_{\mathrm{c}} e''} \qquad \Theta;\Gamma \vdash \mathscr{b}\boxed{e'' \triangleright e'''}}{\Theta;\Gamma \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{e}) \triangleright e'''}}$$

$$\frac{\Theta;\Gamma \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{e}) \triangleright_{\mathrm{p}} \mathsf{S}(\vec{e}')} \qquad \Xi \vdash p \triangleright \mathsf{S}(\vec{e}') \not\rightsquigarrow \quad \text{for } (\Xi \Longrightarrow \mathscr{b}'\boxed{p \equiv v}) \in \mathscr{C}}{\Theta;\Gamma \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{e}) \triangleright \mathsf{S}(\vec{e}')}}$$

$$\frac{\mathsf{a} \notin |\Gamma| \qquad \Theta;\Gamma, \mathsf{a} : A \vdash (\mathscr{B}[\mathsf{a}/x])\boxed{e[\mathsf{a}/x] \triangleright e'}}{\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}\boxed{\{x\}e \triangleright \{x\}e'[x/\mathsf{a}]}}$$

$$\frac{}{\Theta;\Gamma \vdash \mathsf{a} \triangleright \mathsf{a} : A} \qquad \frac{}{\Theta;\Gamma \vdash \mathscr{b}\boxed{\star \triangleright \star}}$$

Figure 15.1.: Normalization with computation rules $\mathscr{C}$ and principal arguments $\wp$.

15.2. Type-directed and normalization phase

We are finally ready to describe equality checking, which is performed by several mutually recursive phases:

$\Theta;\Gamma \vdash \mathscr{B}\boxed{e \sim e'}$	e and e' are equal arguments
$\Theta;\Gamma \vdash s \sim_{\mathrm{e}} t : A$	s and t are extensionally equal
$\Theta;\Gamma \vdash s \sim_{\mathrm{n}} t : A$	normalized terms s and t are equal
$\Theta;\Gamma \vdash A \sim_{\mathrm{n}} B$	normalized types A and B are equal

The first one is the general comparison of arguments e and e' of an object boundary $\mathscr{B}$, the second one the *type-directed phase* which applies extensionality rules by matching the type, and the third the *normalization phase* which compares normalized expressions. We review the inductive clauses specifying these, shown in Figure 15.2. They are parametrized by a standard type theory T, a family of extensionality rules $\mathscr{E}$ over T, a family of computation rules $\mathscr{C}$ over T, and a specification of principal arguments $\wp$. We again treat metavariables as primitive symbols.

General checking $\Theta;\Gamma \vdash \mathscr{B}\boxed{e \sim e'}$ descends under abstractions. It compares types by normalizing them, as there are no extensionality rules

$$\frac{\Theta;\Gamma \vdash (A \triangleright A')\ \mathsf{type} \qquad \Theta;\Gamma \vdash u \sim_{\mathrm{e}} v : A'}{\Theta;\Gamma \vdash u \sim v : A}$$

$$\frac{\Theta;\Gamma \vdash (A \triangleright A')\ \mathsf{type} \qquad \Theta;\Gamma \vdash (B \triangleright B')\ \mathsf{type} \qquad \Theta;\Gamma \vdash (A' \sim_{\mathrm{n}} B')\ \mathsf{type}}{\Theta;\Gamma \vdash (A \sim B)\ \mathsf{type}}$$

$$\frac{\mathsf{a} \notin |\Gamma| \qquad \Theta;\Gamma, \mathsf{a}{:}A \vdash (\mathscr{B}[\mathsf{a}/x])\boxed{e[\mathsf{a}/x] \sim e'[\mathsf{a}/x]}}{\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}\boxed{\{x\}e \sim \{x\}e'}}$$

$$\frac{\begin{array}{l}(\Xi, \mathsf{s}{:}(\square : P), \mathsf{t}{:}(\square : P), \Phi \Longrightarrow \mathsf{s} \equiv \mathsf{t} : P) \in \mathscr{C} \qquad \Xi \vdash P \triangleright A \rightsquigarrow I \\ \Theta;\Gamma \vdash I_*(\mathscr{B}\boxed{e \sim e'}) \qquad \text{for } (\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e'\ \mathsf{by}\ \square}) \in \Xi \\ \Theta;\Gamma \vdash \langle I, \mathsf{s}{\mapsto}u, \mathsf{t}{\mapsto}v\rangle_*(\mathscr{B}\boxed{e \sim e'}) \quad \text{for } (\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e'\ \mathsf{by}\ \square}) \in \Phi\end{array}}{\Theta;\Gamma \vdash u \sim_{\mathrm{e}} v : A}$$

$$\frac{\begin{array}{l}\Xi \vdash P \triangleright A \not\rightsquigarrow \quad \text{for } (\Xi, \mathsf{s}{:}(\square : P), \mathsf{t}{:}(\square : P), \Phi \Longrightarrow \mathsf{s} \equiv \mathsf{t} : P) \in \mathscr{C} \\ \Theta;\Gamma \vdash u \triangleright u' : A \qquad \Theta;\Gamma \vdash v \triangleright v' : A \qquad \Theta;\Gamma \vdash u' \sim_{\mathrm{n}} v' : A\end{array}}{\Theta;\Gamma \vdash u \sim_{\mathrm{e}} v : A}$$

$$\frac{}{\Theta;\Gamma \vdash \mathsf{a} \sim_{\mathrm{n}} \mathsf{a} : A}$$

$$\frac{\begin{array}{l}\text{Rule for } \mathsf{S} \text{ is } \mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{C}\boxed{\mathsf{S}(\widehat{\mathsf{M}}_1, \ldots, \widehat{\mathsf{M}}_n)} \\ \Theta;\Gamma \vdash (\langle \mathsf{M}_1{\mapsto}e_1, \ldots, \mathsf{M}_{i-1}{\mapsto}e_{i-1}\rangle_*\mathscr{B}_i)\boxed{e_i \sim_{\mathrm{n}} e_i'} \qquad \text{if } i \in \wp(\mathsf{S}) \\ \Theta;\Gamma \vdash (\langle \mathsf{M}_1{\mapsto}e_1, \ldots, \mathsf{M}_{i-1}{\mapsto}e_{i-1}\rangle_*\mathscr{B}_i)\boxed{e_i \sim e_i'} \qquad \text{if } i \notin \wp(\mathsf{S})\end{array}}{\Theta;\Gamma \vdash \mathscr{C}\boxed{\mathsf{S}(\vec{e}) \sim_{\mathrm{n}} \mathsf{S}(\vec{e'})}}$$

Figure 15.2.: Equality checking with extensionality rules $\mathscr{C}$ and principal arguments $\wp$.

for types. Terms are compared by the type-directed phase, where the type is first normalized so that it can be matched against extensionality rules.

The type-directed phase checks $\Theta;\Gamma \vdash u \sim_{\mathrm{e}} v : A$ by looking for an extensionality rule that matches A, and applying the rule to reduce the task to verification of the equational premises of the rule. The clause uses the notation $\mathscr{B}\boxed{e \equiv e'\ \mathsf{by}\ \square}$, which turns an object boundary into an equation boundary, as follows:

$$\begin{aligned}(\square : A)\boxed{s \equiv t\ \mathsf{by}\ \square} &= (s \equiv t : A\ \mathsf{by}\ \square),\\ (\square\ \mathsf{type})\boxed{A \equiv B\ \mathsf{by}\ \square} &= (A \equiv B\ \mathsf{by}\ \square),\\ (\{x{:}A\}\ \mathscr{B})\boxed{\{x\}e \equiv \{x\}e'\ \mathsf{by}\ \square} &= \{x{:}A\}(\mathscr{B}\boxed{e \equiv e'\ \mathsf{by}\ \square}).\end{aligned}$$

If no extensionality rule applies, the terms u and v are normalized and compared by the normalization phase.

The normalization phase compares normalized expressions $\mathsf{S}(\vec{e})$ and $\mathsf{S}(\vec{e'})$ by comparing their arguments, where the principal arguments are compared by the normalization phase because they have already been normalized, while the non-principal ones are subjected to general equality comparison.

If the normalization phase compares (normalized) expressions with different head-symbols or a metavariable and a symbol, it of course reports the equation does not hold.

The clauses in Figure 15.2 are readily turned into an equality-checking algorithm, because they are directed by the syntax of their goals. Application of extensionality rules is a possible source of non-determinism,

as a type may match several extensionality rules, and also a source of non-termination, as there is no guarantee that eventually no extensionality rules will be applicable. We discuss strategies for dealing with these issues in Section 15.4.2.

15.3. Soundness

In this section we prove that the normalization and equality checking algorithms are sound. Because normalization and equality checking are intertwined, we prove Theorem 15.3.1 and Theorem 15.3.2 by mutual structural induction.

Theorem 15.3.1 (Soundness of normalization) In a standard type theory, given a family $\mathscr{C}$ of computation rules, and a specification of principal arguments $\wp$, the following hold, where $\mathscr{B}$ and $\mathscr{b}$ are object boundaries:

1. If $\Theta;\Gamma \vdash \mathscr{B}\boxed{e}$ and $\Theta;\Gamma \vdash \mathscr{B}\boxed{e \triangleright e'}$ then $\Theta;\Gamma \vdash \mathscr{B}\boxed{e \equiv e'}$ and $\Theta;\Gamma \vdash \mathscr{B}\boxed{e'}$.
2. If $\Theta;\Gamma \vdash \mathscr{b}\boxed{e}$ and $\Theta;\Gamma \vdash \mathscr{b}\boxed{e \triangleright_{\mathrm{p}} e'}$ then $\Theta;\Gamma \vdash \mathscr{b}\boxed{e \equiv e'}$ and $\Theta;\Gamma \vdash \mathscr{b}\boxed{e'}$.
3. If $\Theta;\Gamma \vdash \mathscr{b}\boxed{e}$ and $\Theta;\Gamma \vdash \mathscr{b}\boxed{e \triangleright_{\mathrm{c}} e'}$ then $\Theta;\Gamma \vdash \mathscr{b}\boxed{e \equiv e'}$ and $\Theta;\Gamma \vdash \mathscr{b}\boxed{e'}$.

Proof. We establish soundness of the rules from Figure 15.1 by mutual structural induction on the derivations. Derivability of $\Theta;\Gamma \vdash \mathscr{B}\boxed{e'}$ in (1) and of $\Theta;\Gamma \vdash \mathscr{b}\boxed{e'}$ in (2) and (3) follows already from Theorem 5.1.6, but we include these nonetheless as they will be needed in Theorem 15.3.2.

Part (1): The case of free variables follows by reflexivity and the variable rule.

If the derivation ends with

$$\frac{\mathsf{a} \notin |\Gamma| \qquad \Theta;\Gamma,\mathsf{a}{:}A \vdash (\mathscr{B}[\mathsf{a}/x])\boxed{e[\mathsf{a}/x] \triangleright e'}}{\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}\boxed{\{x\}e \triangleright \{x\}e'[x/\mathsf{a}]}}$$

then by induction hypothesis

$$\Theta;\Gamma,\mathsf{a}{:}A \vdash (\mathscr{B}[\mathsf{a}/x])\boxed{e'},$$
$$\Theta;\Gamma,\mathsf{a}{:}A \vdash (\mathscr{B}[\mathsf{a}/x])\boxed{e[\mathsf{a}/x] \equiv e'}.$$

We may apply TT-Abstr to these, because $\Theta;\Gamma \vdash A\ \mathsf{type}$ holds by inversion on the assumption $\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}\boxed{\{x\}e}$.

If the derivation ends with

$$\frac{\Theta;\Gamma \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{e}) \triangleright_{\mathrm{p}} \mathsf{S}(\vec{e}')} \qquad \Xi \vdash p \triangleright \mathsf{S}(\vec{e}') \not\rightarrow \quad \text{for } (\Xi \Longrightarrow \mathscr{b}'\boxed{p \equiv v}) \in \mathscr{C}}{\Theta;\Gamma \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{e}) \triangleright \mathsf{S}(\vec{e}')}}$$

then the claim follows by the induction hypothesis (2) for the first premise.

The remaining case is

$$\frac{\Theta;\Gamma \vdash \mathcal{C}\boxed{\mathsf{S}(\vec{e}) \triangleright_{\mathrm{p}} \mathsf{S}(\vec{e}')} \qquad \Theta;\Gamma \vdash \mathcal{C}\boxed{\mathsf{S}(\vec{e}') \triangleright_{\mathrm{c}} e''} \qquad \Theta;\Gamma \vdash \mathcal{C}\boxed{e'' \triangleright e'''}}{\Theta;\Gamma \vdash \mathcal{C}\boxed{\mathsf{S}(\vec{e}) \triangleright e'''}}$$

The induction hypothesis for the last premise secures $\Theta;\Gamma \vdash \mathcal{C}\boxed{e'''}$, while the induction hypotheses for all three premises yield

$$\Theta;\Gamma \vdash \mathcal{C}\boxed{\mathsf{S}(\vec{e}) \equiv \mathsf{S}(\vec{e}')}, \qquad \Theta;\Gamma \vdash \mathcal{C}\boxed{\mathsf{S}(\vec{e}') \equiv e''}, \qquad \Theta;\Gamma \vdash \mathcal{C}\boxed{e'' \equiv e'''}.$$

We may string these together using transitivity to derive $\Theta;\Gamma \vdash \mathcal{C}\boxed{\mathsf{S}(\vec{e}) \equiv e'''}$.

Part (2): Suppose the rule for S is

$$\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathcal{C}'\boxed{\mathsf{S}(\widehat{\mathsf{M}}_1, \ldots, \widehat{\mathsf{M}}_n)},$$

and consider normalization of principal arguments

$$\frac{\begin{array}{rl}\Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i \triangleright e_i'} & \text{if } i \in \wp(\mathsf{S}) \\ e_i = e_i' & \text{if } i \notin \wp(\mathsf{S})\end{array}}{\Theta;\Gamma \vdash \mathcal{C}\boxed{\mathsf{S}(\vec{e}) \triangleright_{\mathrm{p}} \mathsf{S}(\vec{e}')}}$$

where $I = \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle$. For $i = 1, \ldots, n$, we have

$$\Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i \equiv e_i'} \qquad \text{and} \qquad \Theta;\Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i}.$$

Indeed, for $i \in \wp(\mathsf{S})$ the above are just the induction hypotheses of a premise, while for $i \notin \wp(\mathsf{S})$ they respectively hold by reflexivity and an application of Corollary 5.3.2 to $\Theta;\Gamma \vdash \mathcal{C}\boxed{\mathsf{S}(\vec{e})}$. Therefore, the instantiation $J = \langle \mathsf{M}_1 \mapsto e_1', \ldots, \mathsf{M}_n \mapsto e_n' \rangle$ is judgementally equal to I, and because I is derivable, J is derivable by Proposition 5.3.6. From these facts we conclude

$$\begin{array}{ll}\Theta;\Gamma \vdash (I_*\mathcal{C}')\boxed{\mathsf{S}(\vec{e}) \equiv \mathsf{S}(\vec{e}')} & \text{by the congruence rule for } S, \\ \Theta;\Gamma \vdash (J_*\mathcal{C}')\boxed{\mathsf{S}(\vec{e}')} & \text{by the rule for } S.\end{array}$$

If $\mathcal{C}' = (\square\ \mathsf{type})$, we are done. If $\mathcal{C}' = (\square : A)$ and $\mathcal{C} = (\square : B)$ then we derive $\Theta;\Gamma \vdash I_*A \equiv J_*A$ by Theorem 5.1.5 and $\Theta;\Gamma \vdash I_*A \equiv B$ by Theorem 5.2.3 on $\Theta;\Gamma \vdash \mathcal{C}\boxed{\mathsf{S}(\vec{e})}$ and convert the judgements along them.

Part (3): Consider an application of a type computation rule

$$\frac{\begin{array}{c}(\Xi \Longrightarrow P \equiv B) \in \mathscr{C} \qquad \Xi \vdash P \triangleright A \rightsquigarrow I \\ \Theta;\Gamma \vdash I_*(\mathscr{B}\boxed{e \sim e'}) \quad \text{for } (\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e' \text{ by } \square}) \in \Xi\end{array}}{\Theta;\Gamma \vdash A \triangleright_{\mathrm{c}} I_*B}$$

Theorem 15.3.2 ensures $\Theta;\Gamma \vdash I_*(\mathscr{B}\boxed{e \equiv e'})$ for every $(\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e' \text{ by } \square}) \in \Xi$. Therefore, since $\Xi \Longrightarrow P\ \mathsf{type}$ is object-invertible and $\Theta;\Gamma \vdash I_*P\ \mathsf{type}$ has been assumed (note that $I_*P = A$), it follows by Proposition 13.2.3 that I is derivable. We now instantiate the computation rule $\Xi \Longrightarrow P \equiv B$ by I to get $\Theta;\Gamma \vdash A \equiv I_*B$ and appeal to Theorem 5.1.6 for $\Theta;\Gamma \vdash I_*B\ \mathsf{type}$.

It remains to establish the soundness of a derivation ending with a term computation rule

$$\frac{(\Xi \Longrightarrow p \equiv v : B) \in \mathscr{C} \qquad \Xi \vdash p \triangleright s \rightsquigarrow I \qquad \Theta;\Gamma \vdash I_*(\mathscr{B}\boxed{e \sim e'}) \quad \text{for } (\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e' \text{ by } \square}) \in \Xi}{\Theta;\Gamma \vdash s \triangleright_{\mathrm{c}} I_* v : A}$$

Theorem 15.3.2 ensures $\Theta;\Gamma \vdash I_*(\mathscr{B}\boxed{e \equiv e'})$ for every $(\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e' \text{ by } \square}) \in \Xi$. Observe that since by Definition 14.1.2 p is a term symbol application, $\mathsf{mv}(\tau_{\Xi;[]}(p)) \subseteq \mathsf{mv}(p)$ and $I_* p = s$ imply $I_*(\tau_{\Xi;[]}(p)) = \tau_{\Theta;\Gamma}(s)$ by Proposition 5.3.1. Because $\Theta;\Gamma \vdash s : A$ is derivable, so is $\Theta;\Gamma \vdash s : \tau_{\Theta;\Gamma}(s)$ by Corollary 5.3.3, which equals $\Theta;\Gamma \vdash I_* p : I_*(\tau_{\Theta;\Gamma}(p))$. We may apply Proposition 13.2.3 to the object-invertible rule $\Xi \Longrightarrow p : \tau_{\Theta;\Gamma}(p)$ to establish that I is derivable. By instantiating the computation rule $\Xi \Longrightarrow p \equiv v : B$ with I we obtain

$$\Theta;\Gamma \vdash s \equiv I_* v : I_* B$$

and convert it along $\Theta;\Gamma \vdash I_* B \equiv A$ to the desired form, because Theorem 5.1.6 implies $\Theta;\Gamma \vdash s : I_* B$ and Theorem 5.2.3 that $\Theta;\Gamma \vdash I_* B \equiv A$. The last claim follows once again from Theorem 5.1.6. □

Reminder: Term patterns.

Unlike type patterns that can take the form of a (generically applied) metavariable, term patterns are necessarily term symbol applications.

Theorem 15.3.2 (Soundness of equality checking) In a standard type theory, given families $\mathscr{C}$ and $\mathscr{E}$ of computation and extensionality rules, and a specification of principal arguments $\wp$, the following hold, where $\mathscr{B}$ is an object boundary:

1. $\Theta;\Gamma \vdash \mathscr{B}\boxed{e \equiv e'}$ holds if

   $$\Theta;\Gamma \vdash \mathscr{B}\boxed{e}, \qquad \Theta;\Gamma \vdash \mathscr{B}\boxed{e'}, \quad \text{and} \quad \Theta;\Gamma \vdash \mathscr{B}\boxed{e \sim e'}.$$

2. $\Theta;\Gamma \vdash u \equiv v : A$ holds if

   $$\Theta;\Gamma \vdash u : A, \qquad \Theta;\Gamma \vdash v : A, \quad \text{and} \quad \Theta;\Gamma \vdash u \sim_{\mathrm{e}} v : A.$$

3. $\Theta;\Gamma \vdash A \equiv B$ holds if

   $$\Theta;\Gamma \vdash A\ \mathsf{type}, \qquad \Theta;\Gamma \vdash B\ \mathsf{type}, \quad \text{and} \quad \Theta;\Gamma \vdash A \sim_{\mathrm{n}} B.$$

4. $\Theta;\Gamma \vdash u \equiv v : A$ holds if

   $$\Theta;\Gamma \vdash u : A, \qquad \Theta;\Gamma \vdash v : A, \quad \text{and} \quad \Theta;\Gamma \vdash u \sim_{\mathrm{n}} v : A.$$

Proof. We proceed by mutual structural induction on the derivation.

Part (1): Consider a derivation ending with an abstraction

$$\frac{\mathsf{a} \notin |\Gamma| \qquad \Theta;\Gamma, \mathsf{a}{:}A \vdash (\mathscr{B}[\mathsf{a}/x])\boxed{e[\mathsf{a}/x] \sim e'[\mathsf{a}/x]}}{\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}\boxed{\{x\}e \sim \{x\}e'}}$$

By inverting the assumptions we get

$$\Theta;\Gamma, \mathsf{a}{:}A \vdash \mathscr{B}[\mathsf{a}/x]\boxed{e[\mathsf{a}/x]} \qquad \text{and} \qquad \Theta;\Gamma, \mathsf{a}{:}A \vdash \mathscr{B}[\mathsf{a}/x]\boxed{e'[\mathsf{a}/x]},$$

as well as $\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}\boxed{\{x\}e}$. Now the induction hypothesis for the premise yields

$$\Theta;\Gamma, \mathsf{a}{:}A \vdash (\mathscr{B}[\mathsf{a}/x])\boxed{e[\mathsf{a}/x] \equiv e'[\mathsf{a}/x]},$$

which we may abstract with TT-Abstr.

If the derivation ends with

$$\frac{\Theta;\Gamma \vdash (A \triangleright A')\ \mathsf{type} \qquad \Theta;\Gamma \vdash (B \triangleright B')\ \mathsf{type} \qquad \Theta;\Gamma \vdash (A' \sim_{\mathrm{n}} B')\ \mathsf{type}}{\Theta;\Gamma \vdash (A \sim B)\ \mathsf{type}}$$

then Theorem 15.3.1 applied to the first two premises gives

$$\begin{aligned} &\Theta;\Gamma \vdash A \equiv A', && \Theta;\Gamma \vdash A'\ \mathsf{type}, \\ &\Theta;\Gamma \vdash B \equiv B', && \Theta;\Gamma \vdash B'\ \mathsf{type}, \end{aligned}$$

and then the induction hypothesis for the last premise $\Theta;\Gamma \vdash A' \equiv B'$. From these we may derive $\Theta;\Gamma \vdash A \equiv B$ easily.

Suppose the derivation ends with

$$\frac{\Theta;\Gamma \vdash (A \triangleright A')\ \mathsf{type} \qquad \Theta;\Gamma \vdash u \sim_{\mathrm{e}} v : A'}{\Theta;\Gamma \vdash u \sim v : A}$$

By Theorem 5.1.6 applied to the assumption we see that $\Theta;\Gamma \vdash A\ \mathsf{type}$, hence we may apply Theorem 15.3.1 to the first premise and get

$$\Theta;\Gamma \vdash A \equiv A' \qquad \text{and} \qquad \Theta;\Gamma \vdash A'\ \mathsf{type}$$

We convert the assumptions along the above equation to

$$\Theta;\Gamma \vdash u : A' \qquad \text{and} \qquad \Theta;\Gamma \vdash v : A'$$

so that we may apply the induction hypothesis to the second premise and obtain $\Theta;\Gamma \vdash u \equiv v : A'$. One more conversion is then needed to derive $\Theta;\Gamma \vdash u \equiv v : A$.

Part (2): If the derivation ends with

$$\frac{\begin{array}{l}\Xi \vdash A \triangleright P \not\rightarrow \quad \text{for } (\Xi, \mathsf{s}{:}(\square : P), \mathsf{t}{:}(\square : P), \Phi \Longrightarrow \mathsf{s} \equiv \mathsf{t} : P) \in \mathscr{C} \\ \Theta;\Gamma \vdash u \triangleright u' : A \qquad \Theta;\Gamma \vdash v \triangleright v' : A \qquad \Theta;\Gamma \vdash u' \sim_{\mathrm{n}} v' : A\end{array}}{\Theta;\Gamma \vdash u \sim_{\mathrm{e}} v : A}$$

then Theorem 15.3.1 applied to the first two premises establishes

$$\begin{aligned} &\Theta;\Gamma \vdash u \equiv u' : A && \Theta;\Gamma \vdash u' : A \\ &\Theta;\Gamma \vdash v \equiv v' : A && \Theta;\Gamma \vdash v' : A \end{aligned}$$

Then the induction hypothesis tells us that $\Theta;\Gamma \vdash u' \equiv v' : A$. It is now easy to combine the derived equalities into $\Theta;\Gamma \vdash u \equiv v : A$.

If the derivation ends with an application of an extensionality rule

$$\frac{\begin{array}{ll}(\Xi, \mathsf{s}{:}(\square : P), \mathsf{t}{:}(\square : P), \Phi \Longrightarrow \mathsf{s} \equiv \mathsf{t} : P) \in \mathscr{C} & \Xi \vdash A \triangleright P \rightsquigarrow I \\ \Theta; \Gamma \vdash I_*(\mathscr{B}\boxed{e \sim e'}) & \text{for } (\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e' \text{ by } \square}) \in \Xi \\ \Theta; \Gamma \vdash \langle I, \mathsf{s}{\mapsto}u, \mathsf{t}{\mapsto}v\rangle_*(\mathscr{B}\boxed{e \sim e'}) & \text{for } (\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e' \text{ by } \square}) \in \Phi\end{array}}{\Theta; \Gamma \vdash u \sim_{\mathrm{e}} v : A}$$

then $\Theta; \Gamma \vdash A$ type follows from $\Theta; \Gamma \vdash u : A$ by Theorem 5.1.6. Induction hypotheses for the premises give

$$\Theta; \Gamma \vdash I_*(\mathscr{B}\boxed{e \equiv e'}) \qquad \text{for } (\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e' \text{ by } \square}) \in \Xi \tag{15.1}$$

$$\Theta; \Gamma \vdash \langle I, \mathsf{s}{\mapsto}u, \mathsf{t}{\mapsto}v\rangle_*(\mathscr{B}\boxed{e \equiv e'}) \qquad \text{for } (\mathsf{M}{:}\mathscr{B}\boxed{e \equiv e' \text{ by } \square}) \in \Phi \tag{15.2}$$

Because $\Xi \Longrightarrow P$ type is object-invertible, and $I_*P = A$ and $\Theta; \Gamma \vdash A$ type is derivable, by Proposition 13.2.3 the instantiation I is derivable too. We extend I to the instantiation

$$J = \langle I, \mathsf{s}{\mapsto}u, \mathsf{t}{\mapsto}v, \Phi{\mapsto}\star\rangle$$

of the premises of the extensionality rule over $\Theta; \Gamma$, where $\Phi{\mapsto}\star$ signifies that the metavariables of Φ are instantiated with (suitably abstracted) dummy values. We claim that J is derivable: we already know that I is derivable; derivability at s and t reduces to the assumptions $\Theta; \Gamma \vdash u : A$ and $\Theta; \Gamma \vdash u : A$; and derivability at Φ holds by the induction hypotheses (15.2). When we instantiate the extensionality rule with J, we obtain the desired equation.

Parts (3) and (4): The variable case $\Theta; \Gamma \vdash \mathsf{a} \sim_{\mathrm{n}} \mathsf{a} : A$ is trivial.

Suppose the rule for symbol S is

$$\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{C}\boxed{\mathsf{S}(\widehat{\mathsf{M}_1}, \ldots, \widehat{\mathsf{M}_n})}$$

and the derivation ends with

$$\frac{\begin{array}{ll}\Theta; \Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i \sim_{\mathrm{n}} e_i'} & \text{if } i \in \wp(\mathsf{S}) \\ \Theta; \Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i \sim e_i'} & \text{if } i \notin \wp(\mathsf{S})\end{array}}{\Theta; \Gamma \vdash \mathscr{C}\boxed{\mathsf{S}(\vec{e}) \sim_{\mathrm{n}} \mathsf{S}(\vec{e'})}}$$

where $I = \langle \mathsf{M}_1{\mapsto}e_1, \ldots, \mathsf{M}_n{\mapsto}e_n\rangle$, and define $J = \langle \mathsf{M}_1{\mapsto}e_1', \ldots, \mathsf{M}_n{\mapsto}e_n'\rangle$. We first derive

$$\Theta; \Gamma \vdash (I_*\mathscr{C}')\boxed{\mathsf{S}(\vec{e}) \equiv \mathsf{S}(\vec{e'})} \tag{15.3}$$

by the congruence rule associated with S, whose premises are derived as follows:

1. For each $i = 1, \ldots, n$ the premise $\Theta; \Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i}$ is derivable by Corollary 5.3.2 applied to $\Theta; \Gamma \vdash \mathscr{C}\boxed{\mathsf{S}(\vec{e})}$. This also shows that I is derivable.
2. For each $i = 1, \ldots, n$ such that $\mathscr{B}_i$ is an object boundary, the premise $\Theta; \Gamma \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i \equiv e_i'}$ is one of the induction hypotheses. This also shows that I and J are judgementally equal, therefore J is derivable by Proposition 5.3.6.

3. For each $i = 1, \ldots, n$ the premise $\Theta; \Gamma \vdash (J_{(i)*}\mathcal{B}_i)\boxed{e'_i}$ is derivable because J is derivable.

If $\mathcal{C} = \square$ type, we are done. If $\mathcal{C}' = (\square : A)$ and $\mathcal{C} = (\square : B)$, we convert (15.3) along $\Theta; \Gamma \vdash I_*A \equiv B$. The equation holds by Theorem 5.2.3 applied to $\Theta; \Gamma \vdash \mathsf{S}(\vec{e}) : B$ and $\Theta; \Gamma \vdash \mathsf{S}(\vec{e}) : I_*A$, where the latter is derived by Theorem 5.1.6 and the former by the rule for S. □

15.4. Discussion

The relations defined by the inductive clauses from Figure 15.1 and Figure 15.2 serve as the basis of an equality checking algorithm. In order to obtain a working and useful implementation, we need to address several issues.

15.4.1. Classification of rules and principal arguments

An experienced designer of type theories is quite able to recognize computation and extensionality rules, and stitch them together by picking correct principal arguments. There is no need for such manual work, because Proposition 14.1.3 and Proposition 14.2.2 provide easily verifiable syntactic criteria for recognizing computation and extensionality rules. The principal arguments must be chosen correctly, lest the equality checking procedure fail unnecessarily or enter an infinite loop, as shown by the following example.

Example 15.4.1 Consider the computation and extensionality rules for simple products shown in Figure 15.3, where we ignore the linearity requirements, as they just obscure the point we wish to make. Without any principal arguments, the algorithm fails to apply the first computation rule to $\mathsf{fst}(A, B, u)$ in case u normalizes to a pair. More ominous is the infinite loop that is entered on checking

$$[]; x{:}A \times B, y{:}A \times B \vdash x \equiv y : A \times B,$$

where we assume that A and B are already normalized. The algorithm performs the following steps (where all judgements are placed in the context $[]; x{:}A \times B, y{:}A \times B$). First, the extensionality phase reduces the equation to

$$\mathsf{fst}(A, B, x) \equiv \mathsf{fst}(A, B, y) : A, \qquad \mathsf{snd}(A, B, x) \equiv \mathsf{snd}(A, B, y) : B.$$

after which the normalization verifies the first equation by comparing

$$A \equiv A, \qquad B \equiv B, \qquad x \equiv y : A \times B.$$

We may short-circuit the first two equalities, but checking the third one leads back to the original one, *unless* the third argument of fst is principal, in which case the algorithm persists in the normalization phase and fails immediately, as it should.

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{B}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{B}}{\vdash \mathsf{fst}(\mathsf{A}, \mathsf{B}, \mathsf{pair}(\mathsf{A}, \mathsf{B}, \mathsf{s}, \mathsf{t})) \equiv \mathsf{s} : \mathsf{A}}$$

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{B}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{B}}{\vdash \mathsf{snd}(\mathsf{A}, \mathsf{B}, \mathsf{pair}(\mathsf{A}, \mathsf{B}, \mathsf{s}, \mathsf{t})) \equiv \mathsf{t} : \mathsf{B}}$$

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{B}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \times \mathsf{B} \qquad \vdash \mathsf{t} : \mathsf{A} \times \mathsf{B} \qquad \vdash \mathsf{fst}(\mathsf{A}, \mathsf{B}, \mathsf{s}) \equiv \mathsf{fst}(\mathsf{A}, \mathsf{B}, \mathsf{t}) : \mathsf{A} \qquad \vdash \mathsf{snd}(\mathsf{A}, \mathsf{B}, \mathsf{s}) \equiv \mathsf{snd}(\mathsf{A}, \mathsf{B}, \mathsf{t}) : \mathsf{B}}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A} \times \mathsf{B}}$$

Figure 15.3.: Computation and extensionality rules for simple products.

The previous example suggests that we can read off the principal arguments either from extensionality rules, by looking for occurrences of the left and right-hand sides in the subsidiary equalities, or from computation rules, by inspecting the syntactic form of the left-hand side of the rule. We have analyzed a number of standard computation and extensionality rules and identified the following strategy for automatic determination of principal arguments, which we also implemented:

> *The i-th argument of* S *is principal if there is a computation rule* $\Xi \Longrightarrow \mathcal{C}[\boxed{p \equiv v}]$ *such that* $\mathsf{S}(e_1, \ldots, e_n)$ *appears as a sub-pattern of* p *and* e_i *is neither of the form* $\mathsf{M}()$ *nor* $\{\vec{x}\}\mathsf{M}(\vec{x})$.

In many cases, among others the simply-typed λ-calculus, inductive types, and intensional Martin-Löf type theory, the strategy leads to weak head-normal forms. We postpone the pursuit of deeper understanding of this phenomenon to another time.

It would be interesting to combine principal arguments with another common technique for controlling applications of extensionality rules, namely *neutral forms*. Roughly, the principal arguments would still tell which arguments are normalized, but not whether they are compared structurally. Instead, we always compare them recursively, but skip the type-directed phase in $s \equiv t : A$ when the syntactic forms of s and t are neutral, i.e., they indicate that application of extensionality rules cannot lead to further progress. For instance, when checking $x \equiv y : A \times B$ in Example 15.4.1, there is no benefit to applying projections to the variables x and y. Each specific type theory has its own neutral forms, if any, so the user would have to describe these. In some cases it might even be possible to detect the neutral forms automatically.

15.4.2. Determinism, termination and completeness

The inductive clauses in Figure 15.1 and Figure 15.2 could be implemented either as proof search, or as a streamlined algorithm based on normalization. Proof assistants typically implement the latter strategy, because they work with type theories whose normalization is confluent and terminating, and equality checking requires no backtracking. We use the same strategy, so we ought to address non-determinism and non-termination.

A computation or extensionality rule cannot be the source of non-determinism on its own, because Definition 14.1.1, Definition 14.1.2 and Definition 14.2.1 prescribe determinism. However, in either phase of the algorithm several rules may be applicable at the same time, which leads to non-determinism, and we saw in Example 15.4.1 that a poor choice of principal arguments causes non-termination. This is all quite familiar, and so are techniques for ensuring that all is well, including confluence checking and termination arguments based on well-founded relations. While these are doubtlessly important issues, we are not addressing them because they are independent of the algorithm itself. Instead, we aim to provide equality checking that favors generality and extensibility, while still providing soundness through Theorem 15.3.2 and Theorem 15.3.1. In this regard we are in good company, as recent version of Agda allow potentially unsafe user-defined computation rules, a point further discussed in Chapter 17.

A related question is completeness of equality checking, i.e., does the algorithm succeed in checking every derivable equation? Once again, our position is the same: completeness is important, both theoretically and from a practical point of view, but is not the topic of the present thesis. Numerous techniques for establishing completeness of equality checking are known, and these can be applied to any specific instantiation of our algorithm. An interesting direction to pursue would be adaptation of such techniques to our general setting.

16. An implementation in Andromeda 2

Having laid out the algorithm, we report on our experience with its implementation in the Andromeda 2 proof assistant [9, 20, 23], in which the user may define any work in any standard type theory. It is an LCF style proof assistant, i.e., a meta-level programming language with abstract datatypes of judgements, boundaries, and derived rules whose construction and application is controlled by a trusted nucleus (consisting of around 4200 lines of OCaml code).

[20]: Bauer et al. (2018), "Design and Implementation of the Andromeda Proof Assistant"
[9]: Bauer et al. *The Andromeda proof assistant*
[23]: Bauer et al. (2020), "Equality Checking for General Type Theories in Andromeda 2"

The nucleus implements *context-free type theory*, a variant of type theory in which there are no metacontexts and variable contexts. Instead, each free variable is tagged with its type and each metavariable with its boundary, as explained in [69]. Since there are no contexts, a mechanism is needed for tracking proof-irrelevant uses of metavariables and variables, which may occur in derivations of equalities. For this purpose, equality judgements take the form

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

$$A \equiv B \text{ by } \alpha \qquad \text{and} \qquad s \equiv t : A \text{ by } \alpha$$

where α is an *assumption set* whose elements are those metavariables and variables that are used to derive the equality but do not appear in its boundary. The assumptions sets are also recorded in term conversions. As far as the equality checking algorithm is concerned, this is an annoying but inessential complication, because all conversions must be performed explicitly and carefully accounted for.

The implementation of the equality checking algorithm comprises around 1400 lines of OCaml code which reside outside of the trusted nucleus, so that each reasoning step must be passed to the nucleus for validation. The overhead of such a policy is significant, but worth paying in exchange for keeping the nucleus small and uncorrupted, at least in the initial, experimental phase.

Our rudimentary implementation is quite inefficient and cannot compete with the equality checkers found in mature proof assistants. The interesting question is not whether we could try harder to significantly speed up the algorithm, which presumably we could, but whether the design of the algorithm makes it inherently inefficient. We argue that this is not the case. First, we may trade safety for efficiency by placing equality checking into the trusted nucleus, as many proof assistants do, so that we need not check every single step of the algorithm. Second, even though term equality is typed, the normalization procedure is essentially untyped. Indeed, when the rules in Figure 15.1 are used to normalize $\Theta; \Gamma \vdash t : A$ they never modify A, and only ever inspect t, which allows us to ignore A while rewriting t. The soundness of the algorithm guarantees that the normalized term will still have type A.

16.1. Examples

The example in Figure 16.1 shows how dependent products are formalized in Andromeda 2. The rules are direct transcriptions of the usual ones. We linearize the β-rule as shown in Example 13.1.4 to make it a computation rule. We do so by explicitly converting `λ ₂A ₂B s` along the equality `Π ₂A ₂B ≡ Π ₁A ₁B`, which holds by a congruence rule and the premises `ξ` and `ζ`.

```
require eq ;;

rule Π (A type) ({x : A} B type) type ;;
rule λ (A type) ({x : A} B type) ({x : A} e : B{x}) : Π A B ;;
rule app (A type) ({x : A} B type) (s : Π A B) (a : A) : B{a} ;;

rule Πβ
  (A type) ({x:A} B type)
  ({x : A} s : B{x}) (t : A) :
  app A B λ( A B s) t ≡ s{t} : B{t} ;;

rule sym_ty (A type) (B type) (A ≡ B) : B ≡ A ;;

rule Πβ_linear
   (₁A type) ({x:₁A} ₁B type)
   (₂A type) ({x:₂A} ₂B type)
   ({x:₂A} s : ₂B{x}) (t : ₁A)
   (₂A ≡ ₁A by ξ) ({x : ₂A} ₂B{x} ≡ ₁B{convert x ξ} by ζ)
   : app ₁A ₁B (convert λ( ₂A ₂B s)
     (congruence Π( ₂A ₂B) Π( ₁A ₁B) ξ ζ)) t
     ≡ convert s{convert t (sym_ty ₂A ₁A ξ)}
               ζ{convert t (sym_ty ₂A ₁A ξ)} : ₁B{t} ;;

eq.add_rule Πβ_linear ;;

rule Π_ext (A type) ({x : A} B type)
  (f : Π A B) (g : Π A B)
  ({x : A} app A B f x ≡ app A B g x : B{x})
  : f ≡ g : Π A B;;

eq.add_rule Π_ext;;
```

Figure 16.1.: Dependent products in Andromeda 2.

The calls to `eq.add_rule` pass equality rules to the equality checking algorithm, which employs Proposition 14.1.3 and Proposition 14.2.2 to automatically classify the inputs as computation or extensionality rules. It also determines which arguments are principal by using the technique from Section 15.4.1. In the example shown, the linearized rule `Πβ_linear` is classified by the algorithm as computation rule, `Π_ext` as extensionality rule, and the the third argument of `app` is declared principal.

To declare principal arguments Andromeda 2 uses the automatic techinque described in Section 15.4.

Many a newcomer to Martin-Löf type theory is disappointed to learn that only one of equalities $0 + n = n$ and $n + 0 = n$ holds judgementally. In fact, there is strong temptation to pass to extensional type theory just so that a more symmetric notion of equality is recovered, but then one has to give up decidable equality checking. The example in Figure 16.3 and Figure 16.2 shows how our algorithm combines the best of both worlds and demonstrates further capabilities of the implementation.

First, Figure 16.2 shows a formalization of extensional equality types, whose distinguishing feature is the equality reflection principle called `equality_reflection` in the code, which states that the equality type `Eq`

reflects into judgemental equality. Instead of postulating the familiar eliminator `J`, it is more convenient to use an equivalent formulation that uses the judgemental uniqueness of equality proofs `uip`, see Example 14.2.4. Note that `uip` is installed as an extensionality rule into the equality checker. It is well known that equality reflection makes equality checking undecidable, so the equality checker will not be able to prove all equalities. Nevertheless, we expect it to be still quite useful and well behaved.

```
require eq ;;

rule Eq (A type) (a : A) (b : A) type ;;
rule refl (A type) (a : A) : Eq A a a ;;

rule equality_reflection
  (A type) (a : A) (b : A) (_ : Eq A a b)
  : a ≡ b : A ;;

rule uip (A type) (a : A) (b : A)
     (p : Eq A a b) (q : Eq A a b)
     : p ≡ q : Eq A a b ;;

eq.add_rule uip ;;
```

Figure 16.2.: Extensional equality type in Andromeda 2.

We continue our example in Figure 16.3 by postulating the natural numbers `N`. Everything up to the definition of addition is standard, where we also install the computation rules for the induction principle `N_ind` into the equality checker. We then define addition by postulating a term symbol `+` with the defining equality `plus_def` which expresses addition by primitive recursion. We could use `plus_def` as a global computation rule, but we choose to use it only *locally*, with the help of the function `eq.add_locally`.

In the remainder of the code we prove *judgemental* equalities

$$n + 0 \equiv n, \qquad m + \mathsf{succ}(n) \equiv \mathsf{succ}(m + n), \quad \text{and} \quad 0 + n = n.$$

The first one is derived as `plus_zero_right` using `plus_def` as a local computation rule together with `eq.prove` which takes an equational boundary (where $\square$ is written as `??`) and runs the equality checking algorithm to generate a witness for it. The second equality is derived as `plus_succ` in much the same way. The derivation of the third equality relies on equality reflection to convert a term of the equality type `Eq N (zero + n) n` to the corresponding judgemental equality `zero + n ≡ n : N`. We install all three equalities as computation rules.

In addition to proving equalities, we can also normalize terms with `eq.normalize`, and compute strong normal forms (all arguments are principal) with `eq.compute`. In both cases we obtain not only the result, but also a certifying equality. For example, when given `succ zero + succ zero`, the normalizer outputs the weak head-normal form `succ ((succ zero) + zero)`, together with a certificate for the judgemental equality `(succ zero) + (succ zero) ≡ succ ((succ zero) + zero) : N`. Because we installed both neutrality laws for 0 as computation rules, strong normalization reduces `(zero + x) + succ (succ zero + zero)` to `succ (succ x) : N`, where `x` is a free variable of type `N`.

```
rule N type ;;
rule zero : N ;;
rule succ (n : N) : N ;;

rule N_ind
  ({_ : N} C type) (x : C{zero})
  ({n : N} {u : C{n}} f : C{succ n}) (n : N)
  : C{n} ;;

rule βN_zero
  ({_ : N} C type) (x : C{zero})
  ({n : N} {u : C{n}} f : C{succ n})
  : N_ind C x f zero ≡ x : C{zero} ;;

eq.add_rule βN_zero ;;

rule βN_succ
  ({_ : N} C type) (x : C{zero})
  ({n : N} {u : C{n}} f : C{succ n}) (n : N)
  : N_ind C x f (succ n) ≡ f{n, N_ind C x f n} : C{succ n} ;;

eq.add_rule βN_succ ;;

rule (+) (_ : N) (_ : N) : N ;;
rule plus_def (m : N) (n : N) :
  (m + n) ≡ N_ind ({_} N) m ({_ : N} {u : N} succ u) n : N ;;

let plus_zero_right = derive (n : N) →
  eq.add_locally plus_def
    (fun () → eq.prove ((n + zero) ≡ n : N by ??)) ;;

eq.add_rule plus_zero_right ;;

let plus_succ = derive (m : N) (n : N) →
  eq.add_locally plus_def
    (fun () →
      eq.prove ((m + succ n) ≡ (succ (m + n)) : N by ??)) ;;

eq.add_rule plus_succ ;;

let plus_zero_left = derive (k : N) →
  let ap_succ = derive (m : N) (n : N) (p : Eq N m n) →
    eq.add_locally (derive → equality_reflection N m n p)
      (fun () → refl N (succ m) : Eq N (succ m) (succ n)) in
  eq.add_locally plus_def
    (fun () →
       equality_reflection N (zero + k) k
         (N_ind ({n} Eq N (zero + n) n) (refl N zero)
                ({n} {ih} ap_succ (zero + n) n ih) k)) ;;

eq.add_rule plus_zero_left ;;
```

Figure 16.3.: Addition for natural numbers in Andromeda 2.

17. Related work

Designing a user-extensible equality checking algorithm for type theory is a balancing act between flexibility, safety, and automation. We compare ours to that of several proof assistants that support user-extensible equality checking.

The overall design of our algorithm is similar to the equality checking and simplification phases used in the type-reconstruction algorithm of MMT [104, 125], a meta-meta-language for description of formal theories. In MMT inference rules are implemented as trusted low-level executable code, which gives the system an extremely wide scope but also requires care and expertise by the user. In Andromeda 2 the user writes down the desired inference rules directly. The nucleus checks them for compliance with Definition 4.4.5 of a standard type theory before accepting them, which prevents the user from breaking the meta-theoretic properties that the nucleus relies on.

[104]: *The MMT Language and System*
[125]: Rabe (2018), "A Modular Type Reconstruction Algorithm"

Dedukti [50] is a type-checker founded on the logical framework $\lambda\Pi$, extended with user-defined conversion rules. Because equality in Dedukti is based on convertibility of terms, there is no support for user-defined extensionality or η-rules. The Dedukti rewriting system supports higher-order patterns and includes a confluence checker. We see no obstacle to adding some form of confluence checking to Andromeda 2 in the future, while support for higher-order patterns would first have to overcome lack of strengthening, see the discussion following Definition 13.1.2.

[50]: *The Dedukti logical framework*

Recent versions of the proof assistant Agda support user-definable computation rules [38, 39, 41]. Like Dedukti, Agda allows higher-order patterns and provides a confluence checker. It accepts non-linear patterns, which it linearizes and generates suitable equational premises. In addition, it applies built-in η-rules for functions and record types during a type-directed matching phase. It seems to us that the phase could equally well use extensionality rules, which might more easily enable user-defined extensionality principles. Agda designers point out in [38] that having *local* rewrite rules would improve modularity. For example, one could parameterize code by an abstract type, together with rewrite rules it satisfies. This sort of functionality is already present in Andromeda 2, which treats all judgement forms as first-class values, so we may simply pass judgemental equalities as parameters and use them as local computation and extensionality rules.

[39]: Cockx et al. (2016), "Sprinkles of extensionality for your vanilla type theory"
[38]: Cockx (2020), "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules"
[41]: Cockx et al. (2021), "The Taming of the Rew: A Type Theory with Computational Assumptions"
[38]: Cockx (2020), "Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules"

In order to make our equality checking algorithm realistically useful, we ought to combine it with other techniques, such as existential variables, unification, and implicit arguments. Whether that can be done in full generality remains to be seen.

18. Conclusion

The study of meta-theory of type theory is certainly an important step towards making type theories, and the proof assistants they enable, more compatible. With the fast development of computer science and software tools, it is imperative that the proofs stay on solid grounds, which we can achieve by a deep understanding of the mathematical content beneath. While in this thesis we took steps in the directions of transformations and equality checking for type theories, there are certainly many more areas to investigate and improve on. We are looking forward to reading the possible next chapters on the meta-analysis of type theories and other formal systems. Hopefully the story of Carla will inspire new young (and old) minds to join the effort.

Appendix

A. Propositions as (small) types

As an example of a type-theoretic transformation we give the familiar *propositions as (small) types* transformation ([48, 49, 74, 149]) from first-order logic (FOL) to the Martin-Löf type theory ([97–100]) with one universe of (small) types. We give the specific rules of FOL and MLTT following the syntax of finitary type theories. Since both type theories are standard, the object rules are symbol rules, so the signature can be read off the premises.

[149]: Wadler (2015), "Propositions as Types"
[48]: Curry (1934), "Functionality in Combinatory Logic"
[49]: Curry et al. (1958), *Combinatory logic. Vol. I*
[74]: Howard (1980), "The Formulae-as-Types Notion of Construction"
[75]: Paulson et al. *Isabelle/FOL – First-Order Logic*

The rules of first-order logic (FOL) are given according to [75]. Figure A.1 gives the base types, the type of propositions and the type of individuals. Next we give logical connectives in Figure A.2 and the rules for the quantifiers in Figure A.3.

We only describe the fragment of MLTT that is relevant for the transformation, using the type U for the universe and El for the decoding as shown in Figure A.4. We also pose a base type base to which we can map the type of individuals in FOL. Rules for the empty type are in Figure A.5, for the binary sum (dijsoint union) in Figure A.6, for dependent sums in Figure A.7 and for dependent products in Figure A.8. Note that for clarity in binary sums we use the usual infix notation on the symbol + in all but the symbol rule.

[100]: Martin-Löf (1998), "An intuitionistic theory of types"
[97]: Martin-Löf (1975), "An intuitionistic theory of types: predicative part"
[98]: Martin-Löf (1982), "Constructive mathematics and computer programming"
[99]: Martin-Löf (1984), *Intuitionistic type theory*

The syntactic transformation from FOL to MLTT is given in the table below. For every symbol we assign the expression that it is mapped to and the metavariable context over which the expression is syntactically valid. We can read off the metacontext from the symbol rules of FOL.

It is easy to see that this is a valid type-theoretic transformation, as the derivations pertaining the specific rules are mostly very easy to deduce from the shape of the expressions. As an example we take a look at specific rules of FOL: the symbol rule for conj and existsE.

Case conj: The specific rule

$$\frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o}}{\vdash \mathsf{conj}(\mathsf{p}, \mathsf{q}) : \mathsf{o}}$$

gets mapped to

$$\frac{\vdash \mathsf{p} : \mathsf{U} \qquad \vdash \mathsf{q} : \mathsf{U}}{\vdash \sigma(\mathsf{p}, \{x\}\mathsf{q}) : \mathsf{U}} \quad .$$

This can be derived using the symbol rule for σ with the instantiation

$$I = \langle \mathsf{a} \mapsto \mathsf{p}, \mathsf{P} \mapsto \{x\}\mathsf{q} \rangle$$

which is clearly derivable since

$$[\mathsf{p}{:}\mathsf{U}, \mathsf{q}{:}\mathsf{U}]; [] \vdash \{x{:}\mathsf{El}(\mathsf{p})\}\mathsf{q} : \mathsf{U}$$

is derived using TT-Abstr[1] and the specific rule for El.

1: The use of TT-Abstr is trivial, since q does not depend on x. It is in essence just a weakening.

Case existsE: The specific symbol rule

$$\frac{\vdash \mathsf{a} : \mathsf{true}(\mathsf{exists}(\{x\}\mathsf{P}(x))) \qquad \vdash \{x : \mathsf{i}\}\{y : \mathsf{true}(\mathsf{P}(x))\}\mathsf{b} : \mathsf{true}(\mathsf{R}) \qquad \vdash \{x{:}\mathsf{i}\}\mathsf{P} : \mathsf{o} \qquad \vdash \mathsf{R} : \mathsf{o}}{\vdash \mathsf{existsE}(\{x\}\mathsf{P}(x), \mathsf{R}, \mathsf{a}, \{x\}\{y\}\mathsf{b}(x, y)) : \mathsf{true}(\mathsf{R})}$$

is mapped to[2]

$$\frac{\vdash \mathsf{a} : \mathsf{El}(\sigma(\mathsf{base}, \{x\}\mathsf{P}(x))) \qquad \vdash \{x{:}\mathsf{El}(\mathsf{base})\}\{y : \mathsf{El}(\mathsf{P}(x))\}\mathsf{b} : \mathsf{El}(\mathsf{R}) \qquad \vdash \{x{:}\mathsf{El}(\mathsf{base})\}\mathsf{P} : \mathsf{U} \qquad \vdash \mathsf{R} : \mathsf{U}}{\vdash \mathsf{b}(\pi_1(\mathsf{El}(\mathsf{base}), \{x\}\mathsf{El}(\mathsf{P}(x)), \mathsf{a}), \pi_2(\mathsf{El}(\mathsf{base}), \{x\}\mathsf{El}(\mathsf{P}(x)), \mathsf{a})) : \mathsf{El}(\mathsf{R})} \tag{A.1}$$

2: We display the judgement in the traditional farction form. The premises are just the metacontext.

Let Ξ be the metacontext from (A.1). Using TT-Conv-Tm we convert a along the equation

$$\mathsf{El}(\sigma(\mathsf{base}, \{x\}\mathsf{P}(x))) \equiv \Sigma(\mathsf{El}(\mathsf{base}), \{x\}\mathsf{El}(\mathsf{P}))$$

that arises from El-sigma. We then apply the symbol rule for π_1 to derive

$$\Xi; [] \vdash \pi_1(\mathsf{El}(\mathsf{base}), \{x\}\mathsf{El}(\mathsf{P}(x)), \mathsf{a}) : \mathsf{El}(\mathsf{base})$$

Similarly we derive

$$\Xi; [] \vdash \pi_2(\mathsf{El}(\mathsf{base}), \{x\}\mathsf{El}(\mathsf{P}(x)), \mathsf{a}) : \mathsf{El}(\mathsf{P}(\pi_1(\mathsf{El}(\mathsf{base}), \{x\}\mathsf{El}(\mathsf{P}(x)), \mathsf{a})))$$

and conclude the derivation of (A.1) using TT-Meta for b.

Symbol in FOL	Metavariable shape in MLTT	Expression in MLTT
o	[]	U
true	[p:(Tm, 0)]	El(p)
i	[]	El(base)
false	[]	$\bot$
falseE	[p:(Tm, 0), a:(Tm, 0)]	Empty_ind({x}El(p), a)
conj	[p:(Tm, 0), q:(Tm, 0)]	σ(p, {x}q)
conjI	[p:(Tm, 0), q:(Tm, 0), a:(Tm, 0), b:(Tm, 0)]	pair(El(p), {x}El(q), a, b)
conjunct1	[p:(Tm, 0), q:(Tm, 0), a:(Tm, 0)]	π_1(El(p), {x}El(q), a)
conjunct2	[p:(Tm, 0), q:(Tm, 0), a:(Tm, 0)]	π_2(El(p), {x}El(q), a)
disj	[p:(Tm, 0), q:(Tm, 0)]	plus(p, q)
disjI1	[p:(Tm, 0), q:(Tm, 0), a:(Tm, 0)]	inl(El(p), El(q), a)
disjI2	[p:(Tm, 0), q:(Tm, 0), a:(Tm, 0)]	inr(El(p), El(q), a)
disjE	[p:(Tm, 0), q:(Tm, 0), r:(Tm, 0), a:(Tm, 0), b:(Tm, 1), c:(Tm, 1)]	cases(El(p), El(q), {x}El(r), a, {x}b(x), {x}c(x))
imp	[p:(Tm, 0), q:(Tm, 0)]	π(p, {x}q)
impI	[p:(Tm, 0), q:(Tm, 0), a:(Tm, 1)]	λ(El(p), {x}El(q), {x}a)
mp	[p:(Tm, 0), q:(Tm, 0), a:(Tm, 0), b:(Tm, 0)]	app(El(p), {x}El(q), a, b)
all	[P:(Tm, 1)]	π(base, {x}P(x))
allI	[P:(Tm, 1), a:(Tm, 1)]	λ(El(base), {x}El(P(x)), {x}a(x))
spec	[P:(Tm, 1), a:(Tm, 0), M:(Tm, 0)]	app(El(base), {x}El(P(x)), a, M)
exists	[P:(Tm, 1)]	σ(base, {x}P(x))
existsI	[P:(Tm, 1), M:(Tm, 0), a:(Tm, 0)]	pair(El(base), {x}El(P(x)), M, a)
existsE	[P:(Tm, 1), R:(Tm, 0), a:(Tm, 0), b:(Tm, 2)]	b(π_1(El(base), {x}El(P(x)), a), π_2(El(base), {x}El(P(x)), a))

$$\frac{}{\vdash \text{o type}} \qquad \frac{\vdash \text{p} : \text{o}}{\vdash \text{true(p) type}} \qquad \frac{}{\vdash \text{i type}}$$

Figure A.1.: Base types of FOL: A type o of propositions and a type i of individuals.

$$\frac{}{\vdash \mathsf{false} : \mathsf{o}} \qquad \frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{a} : \mathsf{true}(\mathsf{false})}{\vdash \mathsf{falseE}(\mathsf{p}, \mathsf{a}) : \mathsf{true}(\mathsf{p})}$$

$$\frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o}}{\vdash \mathsf{conj}(\mathsf{p}, \mathsf{q}) : \mathsf{o}} \qquad \frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o} \qquad \vdash \mathsf{a} : \mathsf{true}(\mathsf{p}) \qquad \vdash \mathsf{b} : \mathsf{true}(\mathsf{q})}{\vdash \mathsf{conjI}(\mathsf{p}, \mathsf{q}, \mathsf{a}, \mathsf{b}) : \mathsf{true}(\mathsf{conj}(\mathsf{p}, \mathsf{q}))}$$

$$\frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o} \qquad \vdash \mathsf{a} : \mathsf{true}(\mathsf{conj}(\mathsf{p}, \mathsf{q}))}{\vdash \mathsf{conjunct1}(\mathsf{p}, \mathsf{q}, \mathsf{a}) : \mathsf{true}(\mathsf{p})}$$

$$\frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o} \qquad \vdash \mathsf{a} : \mathsf{true}(\mathsf{conj}(\mathsf{p}, \mathsf{q}))}{\vdash \mathsf{conjunct2}(\mathsf{p}, \mathsf{q}, \mathsf{a}) : \mathsf{true}(\mathsf{q})}$$

$$\frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o}}{\vdash \mathsf{disj}(\mathsf{p}, \mathsf{q}) : \mathsf{o}} \qquad \frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o} \qquad \vdash \mathsf{a} : \mathsf{true}(\mathsf{p})}{\vdash \mathsf{disjI1}(\mathsf{p}, \mathsf{q}, \mathsf{a}) : \mathsf{true}(\mathsf{disj}(\mathsf{p}, \mathsf{q}))}$$

$$\frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o} \qquad \vdash \mathsf{a} : \mathsf{true}(\mathsf{q})}{\vdash \mathsf{disjI2}(\mathsf{p}, \mathsf{q}, \mathsf{a}) : \mathsf{true}(\mathsf{disj}(\mathsf{p}, \mathsf{q}))}$$

$$\frac{\begin{array}{c}\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o} \qquad \vdash \mathsf{r} : \mathsf{o} \\ \vdash \mathsf{a} : \mathsf{true}(\mathsf{disj}(\mathsf{p}, \mathsf{q})) \qquad \vdash \{x{:}\mathsf{true}(\mathsf{p})\}\mathsf{b} : \mathsf{true}(\mathsf{r}) \qquad \vdash \{x{:}\mathsf{true}(\mathsf{q})\}\mathsf{c} : \mathsf{true}(\mathsf{r})\end{array}}{\vdash \mathsf{disjE}(\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{a}, \{x\}\mathsf{b}(x), \{x\}\mathsf{c}(x)) : \mathsf{true}(\mathsf{r})}$$

$$\frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o}}{\vdash \mathsf{imp}(\mathsf{p}, \mathsf{q}) : \mathsf{o}} \qquad \frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o} \qquad \vdash \{x{:}\mathsf{true}(\mathsf{p})\}\mathsf{a} : \mathsf{true}(\mathsf{q})}{\vdash \mathsf{impI}(\mathsf{p}, \mathsf{q}, \{x\}\mathsf{a}(x)) : \mathsf{true}(\mathsf{imp}(\mathsf{p}, \mathsf{q}))}$$

$$\frac{\vdash \mathsf{p} : \mathsf{o} \qquad \vdash \mathsf{q} : \mathsf{o} \qquad \vdash \mathsf{a} : \mathsf{true}(\mathsf{imp}(\mathsf{p}, \mathsf{q}))) \qquad \vdash \mathsf{b} : \mathsf{true}(\mathsf{p})}{\vdash \mathsf{mp}(\mathsf{p}, \mathsf{q}, \mathsf{a}, \mathsf{b}) : \mathsf{true}(\mathsf{q})}$$

Figure A.2.: Logical connectives of FOL.

$$\frac{\vdash \{x{:}\mathsf{i}\}\mathsf{P} : \mathsf{o}}{\vdash \mathsf{all}(\{x\}\mathsf{P}(x)) : \mathsf{o}} \qquad \frac{\vdash \{x{:}\mathsf{i}\}\mathsf{P} : \mathsf{o} \qquad \vdash \{x{:}\mathsf{i}\}\mathsf{a} : \mathsf{true}(\mathsf{P}(x))}{\vdash \mathsf{allI}(\{x\}\mathsf{P}(x), \{x\}\mathsf{a}(x)) : \mathsf{true}(\mathsf{all}(\{x\}\mathsf{P}(x)))}$$

$$\frac{\vdash \{x{:}\mathsf{i}\}\mathsf{P} : \mathsf{o} \qquad \vdash \mathsf{a} : \mathsf{true}(\mathsf{all}(\{x\}\mathsf{P}x)) \qquad \vdash \mathsf{M} : \mathsf{i}}{\vdash \mathsf{spec}(\{x\}\mathsf{P}(x), \mathsf{a}, \mathsf{M}) : \mathsf{true}(\mathsf{P}(\mathsf{M}))} \qquad \frac{\vdash \{x{:}\mathsf{i}\}\mathsf{P} : \mathsf{o}}{\vdash \mathsf{exists}(\{x\}\mathsf{P}(x)) : \mathsf{o}}$$

$$\frac{\vdash \{x{:}\mathsf{i}\}\mathsf{P} : \mathsf{o} \qquad \vdash \mathsf{M} : \mathsf{i} \qquad \vdash \mathsf{a} : \mathsf{true}(\mathsf{P}(\mathsf{M}))}{\vdash \mathsf{existsI}(\{x\}\mathsf{P}(x), \mathsf{M}, \mathsf{a}) : \mathsf{true}(\mathsf{exists}(\{x\}\mathsf{P}(x)))}$$

$$\frac{\begin{array}{c}\vdash \{x{:}\mathsf{i}\}\mathsf{P} : \mathsf{o} \qquad \vdash \mathsf{R} : \mathsf{o} \\ \vdash \mathsf{a} : \mathsf{true}(\mathsf{exists}(\{x\}\mathsf{P}(x))) \qquad \vdash \{x : \mathsf{i}\}\{y : \mathsf{true}(\mathsf{P}(x))\}\mathsf{b} : \mathsf{true}(\mathsf{R})\end{array}}{\vdash \mathsf{existsE}(\{x\}\mathsf{P}(x), \mathsf{R}, \mathsf{a}, \{x\}\{y\}\mathsf{b}(x, y)) : \mathsf{true}(\mathsf{R})}$$

Figure A.3.: Quantifiers in FOL.

$$\frac{}{\vdash \mathsf{U}\ \mathsf{type}} \qquad \frac{\vdash \mathsf{t} : \mathsf{U}}{\vdash \mathsf{El}(\mathsf{t})\ \mathsf{type}} \qquad \frac{}{\vdash \mathsf{base} : \mathsf{U}}$$

Figure A.4.: Base types of MLTT.

$$\frac{}{\vdash \bot : \mathsf{U}} \qquad \frac{}{\vdash \mathsf{Empty\ type}} \qquad \text{EL-BOT}\ \frac{}{\vdash \mathsf{El}(\bot) \equiv \mathsf{Empty}}$$

$$\frac{\vdash \{x{:}\mathsf{Empty}\}\mathsf{C\ type} \qquad \vdash \mathsf{a} : \mathsf{Empty}}{\vdash \mathsf{Empty\_ind}(\{x\}\mathsf{C}(x), \mathsf{a}) : \mathsf{C(a)}}$$

Figure A.5.: The empty type of MLTT

$$\frac{\vdash \mathsf{a} : \mathsf{U} \qquad \vdash \mathsf{b} : \mathsf{U}}{\vdash \mathsf{plus(a, b)} : \mathsf{U}} \qquad \frac{\vdash \mathsf{A\ type} \qquad \vdash \mathsf{B\ type}}{\vdash +\mathsf{(A, B)\ type}}$$

$$\text{EL-PLUS}\ \frac{\vdash \mathsf{a} : \mathsf{U} \qquad \vdash \mathsf{b} : \mathsf{U}}{\vdash \mathsf{El(plus(a, b))} \equiv \mathsf{El(a)}+\mathsf{El(b)}}$$

$$\frac{\vdash \mathsf{A\ type} \qquad \vdash \mathsf{B\ type} \qquad \vdash \mathsf{s} : \mathsf{A}}{\vdash \mathsf{inl(A, B, s)} : \mathsf{A+B}}$$

$$\frac{\vdash \mathsf{A\ type} \qquad \vdash \mathsf{B\ type} \qquad \vdash \mathsf{t} : \mathsf{B}}{\vdash \mathsf{inr(A, B, t)} : \mathsf{A+B}}$$

$$\frac{\begin{array}{c}\vdash \mathsf{A\ type} \qquad \vdash \mathsf{B\ type} \qquad \vdash \{x{:}\mathsf{A+B}\}\mathsf{C\ type} \\ \vdash \mathsf{s} : \mathsf{A+B} \qquad \vdash \{x{:}\mathsf{A}\}\mathsf{c}_1{:}\mathsf{C(inl(A, B, }x\mathsf{))} \qquad \vdash \{x{:}\mathsf{B}\}\mathsf{c}_2{:}\mathsf{C(inr(A, B, }x\mathsf{))}\end{array}}{\vdash \mathsf{cases(A, B, }\{x\}\mathsf{C}(x)\mathsf{, s, }\{x\}\mathsf{c}_1(x)\mathsf{, }\{x\}\mathsf{c}_2(x)\mathsf{)} : \mathsf{C(s)}}$$

$$\text{PLUS-BETA-1}\ \frac{\begin{array}{c}\vdash \mathsf{A\ type} \qquad \vdash \mathsf{B\ type} \qquad \vdash \{x{:}\mathsf{A+B}\}\mathsf{C\ type} \\ \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \{x{:}\mathsf{A}\}\mathsf{c}_1{:}\mathsf{C(inl(A, B, }x\mathsf{))} \qquad \vdash \{x{:}\mathsf{B}\}\mathsf{c}_2{:}\mathsf{C(inr(A, B, }x\mathsf{))}\end{array}}{\vdash \mathsf{cases(A, B, }\{x\}\mathsf{C}(x)\mathsf{, inl(A, B, s), }\{x\}\mathsf{c}_1(x)\mathsf{, }\{x\}\mathsf{c}_2(x)\mathsf{)} \equiv \mathsf{c}_1\mathsf{(s)} : \mathsf{C(inl(A, B, s))}}$$

$$\text{PLUS-BETA-2}\ \frac{\begin{array}{c}\vdash \mathsf{A\ type} \qquad \vdash \mathsf{B\ type} \qquad \vdash \{x{:}\mathsf{A+B}\}\mathsf{C\ type} \\ \vdash \mathsf{t} : \mathsf{B} \qquad \vdash \{x{:}\mathsf{A}\}\mathsf{c}_1{:}\mathsf{C(inl(A, B, }x\mathsf{))} \qquad \vdash \{x{:}\mathsf{B}\}\mathsf{c}_2{:}\mathsf{C(inr(A, B, }x\mathsf{))}\end{array}}{\vdash \mathsf{cases(A, B, }\{x\}\mathsf{C}(x)\mathsf{, inr(A, B, t), }\{x\}\mathsf{c}_1(x)\mathsf{, }\{x\}\mathsf{c}_2(x)\mathsf{)} \equiv \mathsf{c}_2\mathsf{(t)} : \mathsf{C(inr(A, B, t))}}$$

Figure A.6.: The disjoint unions in MLTT.

$$\frac{\vdash a : U \qquad \vdash \{x{:}El(a)\}P : U}{\vdash \sigma(a, \{x\}P) : U} \qquad \frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type}{\vdash \Sigma(A, \{x\}B(x))\ type}$$

EL-SIGMA
$$\frac{\vdash a : U \qquad \vdash \{x{:}El(a)\}P : U}{\vdash El(\sigma(a, \{x\}P(x))) \equiv \Sigma(El(a), \{x\}El(P(x)))}$$

$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash s : A \qquad \vdash t : B(s)}{\vdash pair(A, \{x\}B(x), s, t) : \Sigma(A, B)}$$

$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash s : \Sigma(A, \{x\}B(x))}{\vdash \pi_1(A, \{x\}B(x), s) : A}$$

$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash s : \Sigma(A, \{x\}B(x))}{\vdash \pi_2(A, \{x\}B(x), s) : B(\pi_1(A, \{x\}B(x), s))}$$

SIGMA-BETA-1
$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash s : A \qquad \vdash t : B(s)}{\vdash \pi_1(A, \{x\}B(x), pair(A, B, s, t)) \equiv s : A}$$

SIGMA-BETA-2
$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash s : A \qquad \vdash t : B(s)}{\vdash \pi_1(A, \{x\}B(x), pair(A, B, s, t)) \equiv t : B(s)}$$

SIGMA-EXT
$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash s : \Sigma(A, \{x\}B(x)) \qquad \vdash t : \Sigma(A, \{x\}B(x)) \qquad \vdash \pi_1(A, \{x\}B(x), s) \equiv \pi_1(A, \{x\}B(x), t) : A \qquad \vdash \pi_2(A, \{x\}B(x), s) \equiv \pi_2(A, \{x\}B(x), t) : B(\pi_1(A, \{x\}B(x), s))}{\vdash s \equiv t : \Sigma(A, \{x\}B(x))}$$

Figure A.7.: Dependent sums in MLTT.

$$\frac{\vdash a : U \qquad \vdash \{x{:}El(a)\}b : U}{\vdash \pi(a, \{x\}b(x)) : U} \qquad \frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type}{\vdash \Pi(A, \{x\}B(x))\ type}$$

EL-PI
$$\frac{\vdash a : U \qquad \vdash \{x{:}El(a)\}b : U}{\vdash El(\pi(a, \{x\}b(x))) \equiv \Pi(El(a), \{x\}El(b(x)))}$$

$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash \{x{:}A\}e : B(x)}{\vdash \lambda(A, \{x\}B(x), \{x\}e(x)) : \Pi(A, \{x\}B(x))}$$

$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash f : \Pi(A, \{x\}B(x)) \qquad \vdash s{:}A}{\vdash app(A, \{x\}B(x), f, s) : B(s)}$$

PI-BETA
$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash \{x{:}A\}e : B(x) \qquad \vdash s{:}A}{\vdash app(A, \{x\}B(x), \lambda(A, \{x\}B(x), \{x\}e(x)), s) \equiv e(s) : B(s)}$$

PI-EXT
$$\frac{\vdash A\ type \qquad \vdash \{x{:}A\}B\ type \qquad \vdash f : \Pi(A, \{x\}B(x)) \qquad \vdash g : \Pi(A, \{x\}B(x)) \qquad \vdash \{x{:}A\}app(A, \{x\}B(x), f, x) \equiv app(A, \{x\}B(x), g, x) : B(x)}{\vdash f \equiv g : \Pi(A, \{x\}B(x))}$$

Figure A.8.: Dependent products in MLTT.

A union of a chain of well-founded orders

B.

Let us re-state the Lemma 4.4.2 about a union of a chain of well-founded orders:

Lemma B.0.1 Let $(A_n, \sqsubset_n)_{n\in\mathbb{N}}$ be well-founded orders such that for every $n \in \mathbb{N}$, it holds that $A_n \subseteq A_{n+1}$, the order $\sqsubset_n$ is included in $\sqsubset_{n+1}$ and $\sqsubset_n$ is an initial segment of $\sqsubset_{n+1}$, i.e.

$$\forall x, y \in A_{n+1}.\ (x \sqsubset_{n+1} y \wedge y \in A_n) \implies x \sqsubset_n y.$$

Then $A_\infty = \cup_{n\in\mathbb{N}} A_n$ ordered by $\sqsubset$ with

$$\forall x, y \in A_\infty.\ x \sqsubset y \Leftrightarrow \exists n \in \mathbb{N}.\ x, y \in A_n \wedge x \sqsubset_n y$$

is also a well-founded order.

Recall that a *well-founded order* on a set I is an irreflexive transitive relation $\sqsubset$ for which the following holds: for every subset $A \subseteq I$

$$(\forall i \in I.(\forall j \sqsubset i. j \in A) \implies i \in A) \implies A = I.$$

We call a set A for which

$$(\forall i \in I.(\forall j \sqsubset i. j \in A) \implies i \in A)$$

holds a $\sqsubset$-*progressive* set.

Proof. First we observe that from

$$\forall x, y \in A_{n+1}.\ (x \sqsubset_{n+1} y \wedge y \in A_n) \implies x \sqsubset_n y.$$

it follows that for every $n \in \mathbb{N}$

$$\forall x, y \in A_\infty.\ (x \sqsubset y \wedge y \in A_n) \implies x \sqsubset_n y.$$

We then observe that for every $n \in \mathbb{N}$, subset $S \subseteq A_\infty$ and $y \in A_\infty$

$$\forall x \in A_\infty.\ x \sqsubset y \wedge y \in A_n \implies x \in S$$

holds if and only if

$$\forall x \in A_\infty.\ x \sqsubset y \wedge y \in A_n \implies x \in S \cap A_n$$

holds. From that we can deduce that for every $n \in \mathbb{N}$, if a set $S \subseteq A_\infty$ is $\sqsubset$-progressive, then $S \cap A_n$ is $\sqsubset_n$-progressive: Suppose $S \subseteq A_\infty$ is $\sqsubset$-progressive, i.e.

$$\forall y \in A_\infty.\ (\forall x \in A_\infty.\ x \sqsubset y \implies x \in S) \implies y \in S.$$

We want to prove $S \cap A_n$ is $\sqsubset_n$-progressive, i.e.

$$\forall y \in A_n.\ (\forall x \in A_n.\ x \sqsubset_n y \implies x \in S \cap A_n) \implies y \in S \cap A_n.$$

Suppose $y \in A_n$ and $\forall x \in A_n.\ x \sqsubset_n y \implies x \in S \cap A_n$. We want to show $y \in S \cap A_n$. Since $y \in A_n$ by assumption, we only need to show $y \in S$. We use that S is $\sqsubset$-progressive, so we show $\forall x \in A_\infty.\ x \sqsubset y \implies x \in S$. Let $x \in A_\infty$ such that $x \sqsubset y$. Since $y \in A_n$ we have that $x \in A_n$ and $x \sqsubset_n y$. We use the third assumption to deduce $x \in S \cap A_n$, which implies $x \in S$.

Now suppose $S \subseteq A_\infty$ is $\sqsubset$-progressive. Then for every $n \in \mathbb{N}$ we have that $S \cap A_n = A_n$. We can now compute

$$S = \cup_{n\in\mathbb{N}}(S \cap A_n) = \cup_{n\in\mathbb{N}} A_n = A_\infty$$

which concludes the proof. □

Bibliography

[1] Andreas Abel. *foetus - termination checker for simple functional programs*. http://www.informatik.uni-muenchen.de/~abel/foetus/. 1998 (cited on page 70).

[2] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. "Decidability of Conversion for Type Theory in Type Theory". In: *Proceedings of the ACM on Programming Languages* 2.POPL (Dec. 2017) (cited on pages 32, 112, 168, 173).

[3] Andreas Abel and Gabriel Scherer. "On Irrelevance and Algorithmic Equality in Predicative Type Theory". In: *Logical Methods in Computer Science* 8.1 (2012) (cited on pages 113, 173).

[4] Peter Aczel and Nicola Gambino. "Collection Principles in Dependent Type Theory". In: *Types for Proofs and Programs*. Ed. by Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–23 (cited on page 69).

[5] *The Agda proof assistant*. https://wiki.portal.chalmers.se/agda/. 2021 (cited on pages xiv, 5, 32, 70, 112, 168, 171, 173).

[6] Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. "Innovations in computational type theory using Nuprl". In: *Journal of Applied Logic* 4.4 (2006), pp. 428–469 (cited on page 6).

[7] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. "Extending Homotopy Type Theory with Strict Equality". In: *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*. Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 21:1–21:17. DOI: 10.4230/LIPIcs.CSL.2016.21 (cited on page 8).

[8] Thosten Altenkirch, James Chapman, and Tarmo Uustalu. "Monads need not be endofunctors". In: *Logical Methods in Computer Science* 11.1 (Mar. 2015). Ed. by LukeEditor Ong. DOI: 10.2168/lmcs-11(1:3)2015 (cited on pages 35, 36).

[9] Andrej Bauer, Philipp G. Haselwarter, and Anja Petković Komel. *The Andromeda proof assistant*. http://www.andromeda-prover.org/ (cited on pages xvi, 1, 7, 112, 140, 164, 173, 178).

[10] Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. *Two-Level Type Theory and Applications*. 2019. arXiv: 1705.03307 [cs.LO] (cited on page 8).

[11] Andrea Asperti and Wilmer Ricciotti. "A formalization of multi-tape Turing machines". In: *Theoretical Computer Science* 603 (2015), pp. 23–42. DOI: 10.1016/j.tcs.2015.07.013 (cited on page 104).

[12] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. "A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions". In: *Logical Methods in Computer Science* 8.1 (2012). DOI: 10.2168/LMCS-8(1:18)2012 (cited on page 109).

[13] Jeremy Avigad and Solomon Feferman. "Gödel's Functional Interpretation". In: *Handbook of Proof Theory*. Ed. by Samuel R. Buss. Elsevier, 1998. Chap. VI (cited on page 68).

[14] Jeremy Avigad and Solomon Feferman. "Gödel's Functional Interpretation". In: *Handbook of Proof Theory*. Ed. by Samuel R. Buss. Elsevier, 1998. Chap. VI (cited on page 107).

[15] Steve Awodey. "Natural models of homotopy type theory". In: *Mathematical Structures in Computer Science* 28.2 (2018), pp. 241–286 (cited on pages xvi, 6, 7).

[16] Steven Awodey and Andrej Bauer. "Propositions as [Types]". In: *Journal of Logic and Computation* 14.4 (2004), pp. 447–471 (cited on pages 65, 107).

[17] Henk Barendregt. "Lambda Calculi with Types". In: *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures*. USA: Oxford University Press, Inc., 1993, pp. 117–309 (cited on page 8).

[18] Bruno Barras, Jean-Pierre Jouannaud, Pierre-Yves Strub, and Qian Wang. "CoQMTU: A Higher-Order Type Theory with a Predicative Hierarchy of Universes Parametrized by a Decidable First-Order Theory". In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science, LICS 2011, June 21-24, 2011, Toronto, Ontario, Canada*. IEEE Computer Society, 2011, pp. 143–151 (cited on page 7).

[19] Andrej Bauer. *Syntax of dependent type theories*. https://github.com/andrejbauer/dependent-type-theory-syntax/. 2021 (cited on pages 33, 45, 47, 110).

[20] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Christopher A. Stone. "Design and Implementation of the Andromeda Proof Assistant". In: *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*. Vol. 97. LIPIcs. 2018, 5:1–5:31 (cited on pages 112, 140, 173, 178).

[21] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Mike Shulman, Matthieu Sozeau, and Bas Spitters. *The HoTT Library: A formalization of homotopy type theory in Coq*. 2016. arXiv: 1610.04591 [cs.LO] (cited on page 6).

[22] Andrej Bauer, Philipp G. Haselwarter, and Peter LeFanu Lumsdaine. *A general definition of dependent type theories*. 2020. arXiv: 2009.05539 [math.LO] (cited on pages xvi, 1, 7, 23, 53, 164, 165).

[23] Andrej Bauer, Philipp G. Haselwarter, and Anja Petković. "Equality Checking for General Type Theories in Andromeda 2". In: *Mathematical Software – ICMS 2020*. 2020, pp. 253–259 (cited on pages 140, 178).

[24] Andrej Bauer and Anja Petković Komel. *An extensible equality checking algorithm for dependent type theories*. 2021. arXiv: 2103.07397 [cs.LO] (cited on pages ix, 2, 3, 8, 28, 168).

[25] Marc Bezem, Thierry Coquand, and Simon Huber. "A Model of Type Theory in Cubical Sets". In: *19th International Conference on Types for Proofs and Programs (TYPES 2013)*. Ed. by Ralph Matthes and Aleksy Schubert. Vol. 26. Leibniz International Proceedings in Informatics. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 107–128 (cited on pages 6, 69).

[26] Marc Bezem, Thierry Coquand, and Simon Huber. "The Univalence Axiom in Cubical Sets". In: *Journal of Automated Reasoning* 63.2 (2019), pp. 159–171 (cited on pages 1, 6, 164).

[27] Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. "Modal dependent type theory and dependent right adjoints". In: *Mathematical Structures in Computer Science* 30.2 (2020), pp. 118–138. DOI: 10.1017/S0960129519000197 (cited on page 108).

[28] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. "The next 700 syntactical models of type theory". In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*. Ed. by Yves Bertot and Viktor Vafeiadis. ACM, 2017, pp. 182–194 (cited on page 7).

[29] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. "The next 700 syntactical models of type theory". In: *Certified Programs and Proofs (CPP 2017)*. Paris, France, Jan. 2017, pp. 182–194 (cited on page 108).

[30] Simon Pierre Boulier. "Extending type theory with syntactic models". PhD thesis. Ecole nationale supérieure Mines-Télécom Atlantique, Nov. 2018 (cited on page 108).

[31] Simon Pierre Boulier. "Extending type theory with syntactic models. (Etendre la théorie des types à l'aide de modèles syntaxiques)". PhD thesis. Ecole nationale supérieure Mines-Télécom Atlantique Bretagne Pays de la Loire, France, 2018 (cited on page 7).

[32] Edwin C. Brady. "Idris, a general-purpose dependently typed programming language: Design and implementation". In: *Journal of Functional Programming* 23.5 (2013), pp. 552–593. DOI: 10.1017/S095679681300018X (cited on page 109).

[33] John W. Cartmell. "Generalised algebraic theories and contextual categories". PhD thesis. Oxford University, 1978 (cited on page 6).

[34] John W. Cartmell. “Generalised algebraic theories and contextual categories”. In: *Annals of Pure and Applied Logic* 32 (1986), pp. 209–243 (cited on pages xvi, 6).

[35] Arthur Charguéraud. “The Locally Nameless Representation”. In: *Journal of Automated Reasoning* 49 (2012), pp. 363–408 (cited on page 10).

[36] Alonzo Church. “A Formulation of the Simple Theory of Types”. In: *Journal of Symbolic Logic* 5.2 (June 1940), pp. 56–68 (cited on pages 1, 5, 164).

[37] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press, 1941 (cited on page 5).

[38] Jesper Cockx. “Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules”. In: *25th International Conference on Types for Proofs and Programs (TYPES 2019)*. Ed. by Marc Bezem and Assia Mahboubi. Vol. 175. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, 2:1–2:27 (cited on pages 7, 144).

[39] Jesper Cockx and Andreas Abel. “Sprinkles of extensionality for your vanilla type theory”. In: *22nd International Conference on Types for Proofs and Programs TYPES 2016*. University of Novi Sad, 2016 (cited on pages 7, 112, 144, 173).

[40] Jesper Cockx and Andreas Abel. “Elaborating Dependent (Co)Pattern Matching”. In: *Proceedings of the ACM on Programming Languages* 2.ICFP (July 2018). DOI: 10.1145/3236770 (cited on page 109).

[41] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. “The Taming of the Rew: A Type Theory with Computational Assumptions”. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021) (cited on pages 7, 144).

[42] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. “Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom”. In: *TYPES 2015*. Vol. 69. Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015, 5:1–5:34 (cited on pages 1, 6, 69, 164).

[43] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. *The ‘cubicaltt’ theorem prover*. https://github.com/mortberg/cubicaltt. 2018 (cited on pages xiv, 6, 32, 168).

[44] Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system*. Prentice Hall, 1986 (cited on pages 6, 65, 107).

[45] *The Coq proof assistant, version 2021.02.2*. https://coq.inria.fr/. 2021 (cited on pages xiv, xv, 6, 32, 70, 112, 168, 171, 173).

[46] Thierry Coquand and Gérard P. Huet. “The Calculus of Constructions”. In: *Information and Computation* 76.2/3 (1988), pp. 95–120 (cited on page 6).

[47] Thierry Coquand and Christine Paulin. “Inductively defined types”. In: *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*. Ed. by Per Martin-Löf and Grigori Mints. Vol. 417. Lecture Notes in Computer Science. Springer, 1988, pp. 50–66 (cited on pages 1, 6, 164).

[48] Haskell B. Curry. “Functionality in Combinatory Logic”. In: *Proceedings of the National Academy of Sciences of the United States of America* 20.11 (1934), pp. 584–590 (cited on pages xiv, 33, 65, 107, 147, 171).

[49] Haskell B. Curry and Robert Feys. *Combinatory logic. Vol. I*. Studies in Logic and the Foundations of Mathematics. North-Holland Publishing Co., Amsterdam, 1958, pp. xvi+417 (cited on pages xiv, 33, 65, 107, 147, 171).

[50] *The Dedukti logical framework*. https://deducteam.github.io (cited on pages 7, 112, 144, 173).

[51] Jana Dunfield and Neel Krishnaswami. *Bidirectional Typing*. 2020. arXiv: 1908.05839 [cs.PL] (cited on page 109).

[52] Peter Dybjer. “Internal Type Theory”. In: *TYPES 1995*. Vol. 1158. Lecture Notes in Computer Science. Springer, 1995, pp. 120–134 (cited on pages xvi, 6, 108).

[53] Francisco Ferreira and Brigitte Pientka. “Bidirectional Elaboration of Dependently Typed Programs”. In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. PPDP ’14. Canterbury, United Kingdom: Association for Computing Machinery, 2014, pp. 161–174. DOI: 10.1145/2643135.2643153 (cited on pages 108, 109).

[54] Michael J. Fischer. “Lambda-Calculus Schemata”. In: *Lisp and Symbolic Computation* 6.3-4 (1993), pp. 259–288 (cited on pages 68, 107).

[55] Harvey Friedman. “Classically and intuitionistically provably recursive functions”. In: *Higher Set Theory*. Ed. by Gert H. Müller and Dana S. Scott. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 21–27 (cited on pages 68, 107).

[56] Gerhard Gentzen. “Die Widerspruchsfreiheit der reinen Zahlentheorie”. In: *Mathematische Annalen* 112 (1936), pp. 493–565 (cited on pages 67, 107).

[57] Gerhard Gentzen. “Über das Verhältnis zwischen intuitionistischer und klassischer Arithmetik”. In: *Archiv für mathematische Logik und Grundlagenforschung* 16 (1974), pp. 119–132 (cited on pages 67, 107).

[58] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. “Definitional proof-irrelevance without K”. In: *Proceedings of the ACM on Programming Languages* 3.POPL (2019), 3:1–3:28 (cited on pages 32, 112, 127, 168, 173).

[59] Jean-Yves Girard. “Une Extension De L’Interpretation De Gödel a L’Analyse, Et Son Application a L’Elimination Des Coupures Dans L’Analyse Et La Theorie Des Types”. In: *Proceedings of the Second Scandinavian Logic Symposium*. Ed. by J.E. Fenstad. Vol. 63. Studies in Logic and the Foundations of Mathematics. Elsevier, 1971, pp. 63–92 (cited on page 8).

[60] Jean-Yves Girard. “Interprétation Fonctionelle et Élimination Des Coupures de l’arithmétique d’ordre Supérieur”. PhD thesis. Université Paris VII, 1972 (cited on page 5).

[61] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. USA: Cambridge University Press, 1989 (cited on page 8).

[62] Kurt Gödel. “Zur intuitionistischen Arithmetik und Zahlentheorie”. In: *Ergebnisse eines Mathematischen Kolloquiums* (4 1933), pp. 34–38 (cited on pages 67, 107).

[63] Kurt Gödel. “Über eine bisher nicht erweiterung des finiten standpunktes”. In: *Dialectica* 12.3 - 4 (1958), pp. 280–287 (cited on pages 68, 107).

[64] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O’Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. “A Machine-Checked Proof of the Odd Order Theorem”. In: *Interactive Theorem Proving*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 163–179 (cited on page xiv).

[65] Michael J. C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. “A Metalanguage for Interactive Proof in LCF”. In: *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*. Ed. by Alfred V. Aho, Stephen N. Zilles, and Thomas G. Szymanski. ACM Press, 1978, pp. 119–130 (cited on page 5).

[66] Robert Harper. *An Equational Logical Framework for Type Theories*. 2021. arXiv: 2106.01484 [math.LO] (cited on pages xvi, 1, 7, 164, 165).

[67] Robert Harper, Furio Honsell, and Gordon Plotkin. “A Framework for Defining Logics”. In: *Journal of the ACM* 40.1 (Jan. 1993), pp. 143–184 (cited on page 7).

[68] Robert Harper and Christopher Stone. “A Type-Theoretic Interpretation of Standard ML”. In: *In Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998 (cited on page 109).

[69] Philipp G. Haselwarter and Andrej Bauer. *Finitary type theories with and without contexts*. 2021. arXiv: 2112.00539 [math.LO] (cited on pages xvi, 1, 2, 7–9, 25, 28, 140, 164, 165, 168, 178).

[70] Martin Hofmann. “Conservativity of Equality Reflection over Intensional Type Theory”. In: *Selected Papers from the International Workshop on Types for Proofs and Programs*. TYPES 1995. Berlin, Heidelberg: Springer-Verlag, 1996, pp. 153–164 (cited on pages 68, 108).

[71] Martin Hofmann. “Extensional Constructs in Intensional Type Theory”. In: *CPHC/BCS Distinguished Dissertations*. 1997 (cited on page 7).

[72] Martin Hofmann. *Extensional Constructs in Intensional Type Theory*. Berlin, Heidelberg: Springer-Verlag, 1997 (cited on pages 68, 108).

[73] Martin Hofmann and Thomas Streicher. “The Groupoid Model Refutes Uniqueness of Identity Proofs”. In: *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994*. IEEE Computer Society, 1994, pp. 208–212 (cited on page 6).

[74] William Alvin Howard. “The Formulae-as-Types Notion of Construction”. In: *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Ed. by Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan. Academic Press, 1980 (cited on pages xiv, 33, 65, 107, 147, 171).

[75] Larry Paulson and Markus Wenzel. *Isabelle/FOL – First-Order Logic*. https://www.cl.cam.ac.uk/research/hvg/Isabelle/dist/library/FOL/FOL/outline.pdf (cited on page 147).

[76] *Isabelle*. https://isabelle.in.tum.de/. 2016 (cited on page 5).

[77] Valery Isaev. *Algebraic Presentations of Dependent Type Theories*. 2017. arXiv: 1602.08504 [math.LO] (cited on page 7).

[78] Valery Isaev. *Arend Standard Library*. https://github.com/JetBrains/arend-lib. 2021 (cited on pages xiv, 6, 32, 168).

[79] Bart Jacobs. “Comprehension categories and the semantics of type dependency”. In: *Theoretical Computer Science* 107.2 (1993), pp. 169–207 (cited on page 6).

[80] Ambrus Kaposi, Simon Huber, and Christian Sattler. “Gluing for Type Theory”. In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 25:1–25:19 (cited on page 108).

[81] Mikhail M. Kapranov and Vladimir A. Voevodsky. “∞-groupoids and homotopy types”. eng. In: *Cahiers de Topologie et Géométrie Différentielle Catégoriques* 32.1 (1991), pp. 29–46 (cited on page xiii).

[82] Krzysztof Kapulkin and Peter LeFanu Lumsdaine. “The simplicial model of Univalent Foundations (after Voevodsky)”. In: *Journal of the European Mathematical Society* 23.6 (2021), pp. 2071–2126 (cited on page 6).

[83] Andrei Nikolaevich Kolmogorov. “On the principles of excluded middle (Russian)”. In: *Matematičeski Sbornik* 32 (4 1925), pp. 646–667 (cited on pages 67, 107).

[84] Jean-Louis Krivine. “Opérateurs de mise en mémoire et traduction de Gödel”. In: *Archive for Mathematical Logic* 30.4 (July 1990), pp. 241–267 (cited on pages 67, 107).

[85] Nikolai Kudasov. *A proof assistant for synthetic ∞-categories*. https://types21.liacs.nl/download/a-proof-assistant-for-synthetic-∞-categories. TYPES 2021, 27th International Conference on Types for Proofs and Programs, June 2021. 2021 (cited on page 109).

[86] Sigekatu Kuroda. “Intuitionistische Untersuchungen der formalistischen Logik”. In: *Nagoya Mathematical Journal* 2 (1951), pp. 35–47 (cited on pages 67, 107).

[87] Joachim Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986 (cited on page 6).

[88] *The Lean theorem prover*. https://leanprover.github.io (cited on page 6).

[89] Daniel K. Lee, Karl Crary, and Robert Harper. “Towards a Mechanized Metatheory of Standard ML”. In: *SIGPLAN Not.* 42.1 (Jan. 2007), pp. 173–184. DOI: 10.1145/1190215.1190245 (cited on page 109).

[90] Xavier Leroy. “A formally verified compiler back-end”. In: *Journal of Automated Reasoning* 43.4 (2009), pp. 363–446 (cited on page xiv).

[91] Daniel R. Licata and Guillaume Brunerie. “A Cubical Approach to Synthetic Homotopy Theory”. In: *Proceedings of the 2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. LICS 2015. Washington, DC, USA: IEEE Computer Society, 2015, pp. 92–103 (cited on pages 6, 69).

[92] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. “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*. Ed. by Hélène Kirchner. Vol. 108. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 22:1–22:17 (cited on page 6).

[93] Gaïa Loutchmia, Jure Taslak, Danel Ahman, and Andrej Bauer (supervisor). *Formalization of simple type theory*. https://github.com/Chaaaos/formaltt. 2021 (cited on pages 37, 39, 40, 110).

[94] Gennady Semenovich Makanin. “On the identity problem in finitely defined semigroups”. In: *Dokl. Akad. Nauk SSSR* 171.2 (1966), pp. 285–287 (cited on pages 99, 100).

[95] Andrey A. Markov. “Impossibility of certain algorithms in the theory of associative systems”. In: *Dokl. Akad. Nauk SSSR* 55 (7 1947), pp. 587–590 (cited on page 99).

[96] Andrey A. Markov. “Impossibility of certain algorithms in the theory of associative systems”. In: *Dokl. Akad. Nauk SSSR* 58 (1947), pp. 353–356 (cited on page 99).

[97] Per Martin-Löf. “An intuitionistic theory of types: predicative part”. In: *Logic Colloquium 1973, Proceedings of the Logic Colloquium*. Ed. by H.E. Rose and J.C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. North-Holland, 1975, pp. 73–118 (cited on pages 1, 5, 65, 147, 164).

[98] Per Martin-Löf. “Constructive mathematics and computer programming”. In: *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science, Hannover 1979*. Ed. by L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski. Vol. 104. Studies in Logic and the Foundations of Mathematics. North-Holland, 1982, pp. 153–175 (cited on pages 1, 5, 65, 147, 164).

[99] Per Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin*. Vol. 1. Studies in Proof Theory. Bibliopolis, 1984, pp. iv+91 (cited on pages 1, 5, 65, 147, 164).

[100] Per Martin-Löf. “An intuitionistic theory of types”. In: *Twenty-five years of constructive type theory (Venice, 1995)*. Ed. by Giovanni Sambin and Jan M. Smith. Vol. 36. Oxford Logic Guides. Oxford University Press, 1998, pp. 127–172 (cited on pages 1, 5, 65, 147, 164).

[101] Yuri Vladimirovich Matiyasevich. “Simple examples of undecidable associative calculi”. In: *Dokl. AN SSSR* 173.16 (1967), pp. 555–557 (cited on page 99).

[102] James McKinna and Robert Pollack. “Pure Type Systems Formalized”. In: *International Conference on Typed Lambda Calculi and Applications (TLCA)*. Ed. by Mark Bezem and Jan F. Groote. Vol. 664. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 1993 (cited on page 10).

[103] Robin Milner. *Logic for Computable Functions: description of a machine implementation*. Tech. rep. Defence Technical Information Center, 1972 (cited on page 5).

[104] *The MMT Language and System*. https://uniformal.github.io// (cited on page 144).

[105] Eugenio Moggi. “A category-theoretic account of program modules”. In: *Mathematical Structures in Computer Science* 1.1 (1991), pp. 103–139 (cited on page 6).

[106] Leonardo de Moura. “Formalizing Mathematics using the Lean Theorem Prover”. In: *International Symposium on Artificial Intelligence and Mathematics*. 2016 (cited on page 6).

[107] Leonardo de Moura, Jeremy Avigad, Soonho Kong, and Cody Roux. *Elaboration in Dependent Type Theory*. 2015. arXiv: 1505.04324 [cs.LO] (cited on page 109).

[108] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. “The Lean Theorem Prover (System Description)”. In: *25th International Conference on Automated Deduction (CADE 25)*. Aug. 2015 (cited on pages xiv, 32, 112, 168, 173).

[109] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. “The Lean Theorem Prover (System Description)”. In: *CADE*. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015, pp. 378–388 (cited on page 6).

[110] Ulf Norell. “Towards a practical programming language based on dependent type theory”. Technical Report 33D. PhD thesis. Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007 (cited on page 109).

[111] Ian Orton and Andrew M. Pitts. “Axioms for Modelling Cubical Type Theory in a Topos”. In: *Logical Methods in Computer Science* Volume 14, Issue 4 (Dec. 2018) (cited on page 6).

[112] Nicolas Oury. “Extensionality in the Calculus of Constructions”. In: *Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics*. TPHOLs 2005. Oxford, UK: Springer-Verlag, 2005, pp. 278–293 (cited on pages 68, 108).

[113] Pierre-Marie Pédrot. “A functional functional interpretation”. In: *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS ’14, Vienna, Austria, July 14 - 18, 2014*. Ed. by Thomas A. Henzinger and Dale Miller. ACM, 2014, 77:1–77:10 (cited on pages 68, 107).

[114] Pierre-Marie Pédrot. “A Materialist Dialectica. (Une Dialectica matérialiste)”. PhD thesis. Paris Diderot University, France, 2015 (cited on pages 68, 107).

[115] Pierre-Marie Pédrot and Nicolas Tabareau. “An effectful way to eliminate addiction to dependence”. In: *Logic in Computer Science 2017*. IEEE Computer Society, 2017, pp. 1–12 (cited on page 7).

[116] Anja Petković Komel. *Towards an elaboration theorem*. Invited Talk. HoTT/UF 2021. 2021 (cited on page 110).

[117] Frank Pfenning. “Intensionality, Extensionality, and Proof Irrelevance in Modal Type Theory”. In: *Logic in Computer Science*. IEEE Computer Society, 2001, pp. 221–230 (cited on pages 65, 107).

[118] Frank Pfenning. “Logical Frameworks”. In: *Handbook of Automated Reasoning*. Elsevier Science Publishers B. V., 2001, pp. 1063–1147 (cited on page 7).

[119] Frank Pfenning and Carsten Schürmann. “System Description: Twelf — A Meta-Logical Framework for Deductive Systems”. In: *Automated Deduction — CADE-16*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 202–206 (cited on page 7).

[120] Brigitte Pientka and Jana Dunfield. “Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)”. In: *Automated Reasoning*. Ed. by Jürgen Giesl and Reiner Hähnle. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–2 (cited on page 7).

[121] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002 (cited on page 5).

[122] Benjamin C. Pierce and David N. Turner. “Local type inference”. In: *ACM Transactions on Programming Languages and Systems* 22.1 (2000), pp. 1–44 (cited on page 109).

[123] Robert Pollack. “Implicit Syntax”. In: *Informal proceedings of first workshop on logical frameworks*. 1992 (cited on pages 108, 109).

[124] Emil L. Post. “Recursive Unsolvability of a Problem of Thue”. In: *The Journal of Symbolic Logic* 12.1 (1947), pp. 1–11 (cited on page 99).

[125] Florian Rabe. “A Modular Type Reconstruction Algorithm”. In: *ACM Transactions on Computational Logic* 19.4 (2018), 24:1–24:43 (cited on page 144).

[126] John C. Reynolds. “Definitional Interpreters for Higher-order Programming Languages”. In: *Proceedings of the ACM Annual Conference - Volume 2*. ACM 1972. Boston, Massachusetts, USA: Association for Computing Machinery, 1972, pp. 717–740 (cited on pages 68, 107).

[127] John C. Reynolds. “Towards a Theory of Type Structure”. In: *Programming Symposium, Proceedings Colloque Sur La Programmation*. Berlin, Heidelberg: Springer-Verlag, 1974, pp. 408–423 (cited on page 8).

[128] John C. Reynolds. “Towards a theory of type structure”. In: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*. Ed. by Bernard Robinet. Vol. 19. Lecture Notes in Computer Science. Springer, 1974, pp. 408–423 (cited on page 5).

[129] Bertrand Russell. “Mathematical Logic as Based on the Theory of Types”. In: *American Journal of Mathematics* 30.3 (1908), pp. 222–262 (cited on page 5).

[130] Robert Andrew George Seely. “Locally cartesian closed categories and type theory”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 95.1 (1984), pp. 33–48 (cited on page 6).

[131] Carlos Simpson. *Homotopy types of strict 3-groupoids*. 1998. arXiv: 9810059 [math.CT] (cited on page xiii).

[132] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. “The MetaCoq Project”. In: *Journal of Automated Reasoning* 64.5 (2020), pp. 947–999 (cited on page 8).

[133] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. “Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq”. In: *Proceedings of the ACM on Programming Languages* 4.POPL (Dec. 2019) (cited on pages 8, 32, 112, 168, 173).

[134] Matthieu Sozeau and Nicolas Tabareau. “Universe Polymorphism in Coq”. In: *Interactive Theorem Proving*. Ed. by Gerwin Klein and Ruben Gamboa. Cham: Springer International Publishing, 2014, pp. 499–514 (cited on page 70).

[135] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. *A Cubical Language for Bishop Sets*. 2021. arXiv: 2003.01491 [cs.LO] (cited on page 127).

[136] Christopher A. Stone and Robert Harper. “Extensional equivalence and singleton types”. In: *ACM Transactions on Computational Logic* 7.4 (2006), pp. 676–722 (cited on pages 113, 173).

[137] Thomas Streicher. In: *Journal of Applied Logic* 12.1 (2014). Logic Categories Semantics, pp. 45–49 (cited on page 6).

[138] Margaret E. Szabo. “The Collected Papers of Gerhard Gentzen”. In: *Journal of Philosophy* 68.8 (1971), pp. 238–265 (cited on pages 67, 107).

[139] Paul Taylor. “Internal Completeness of Categories of Domains”. In: *Proceedings of a Tutorial and Workshop on Category Theory and Computer Programming*. Guildford, United Kingdom: Springer-Verlag, 1986, pp. 449–465 (cited on page 6).

[140] The Agda development team. *Agda standard library*. https://github.com/agda/agda-stdlib/blob/master/src/Induction/Lexicographic.agda. 2021 (cited on page 89).

[141] The RedPRL Development Team. *The ‘redtt’ theorem prover*. https://github.com/RedPRL/redtt. 2020 (cited on pages xiv, 6, 32, 168).

[142] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013 (cited on pages 1, 6, 65, 107, 164).

[143] Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* s2-42.1 (Jan. 1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230. eprint: https://academic.oup.com/plms/article-pdf/s2-42/1/230/4317544/s2-42-1-230.pdf (cited on page 104).

[144] Taichi Uemura. *A General Framework for the Semantics of Type Theory*. 2019. arXiv: 1904.04097 [math.CT] (cited on pages xvi, 1, 7, 108–110, 164, 165).

[145] Taichi Uemura. “Abstract and Concrete Type Theories”. PhD thesis. Institute for Logic, Language and Computation, University of Amseterdam, 2021 (cited on pages 7, 8).

[146] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. “Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types”. In: *Proceedings of the ACM Programming Languages* 3.ICFP (July 2019). DOI: 10.1145/3341691 (cited on pages xiv, 6, 32, 168).

[147] Vladimir Voevodsky. “Univalent Foundations of Mathematics”. In: *Logic, Language, Information and Computation - 18th International Workshop, WoLLIC 2011, Philadelphia, PA, USA, May 18-20, 2011. Proceedings*. Ed. by Lev D. Beklemishev and Ruy J. G. B. de Queiroz. Vol. 6642. Lecture Notes in Computer Science. Springer, 2011, p. 4 (cited on page 6).

[148] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath — a computer-checked library of univalent mathematics*. available at https://unimath.org. URL: https://github.com/UniMath/UniMath (cited on pages xvi, 6).

[149] Philip Wadler. “Propositions as Types”. In: *Communications of ACM* 58.12 (Nov. 2015), pp. 75–84. DOI: 10.1145/2699407 (cited on pages xiv, 33, 65, 107, 147, 171).

[150] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1925 (cited on page 5).

[151] Théo Winterhalter. “Formalisation and Meta-Theory of Type Theory”. PhD thesis. L’Université de Nantes, France, 2020 (cited on pages xvi, 108).

[152] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. “Eliminating Reflection from Type Theory”. In: *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Certified Programs and Proofs 2019. Cascais, Portugal: Association for Computing Machinery, 2019, pp. 91–103 (cited on pages 41, 68, 107, 168).

Razširjeni povzetek v slovenščini

V tem povzetku so opisane poglavitne ideje, definicije in izreki doktorske disertacije. Natančni dokazi in podrobnosti niso prevedeni.

1. Uvod

Čeprav je znanih veliko konkretnih primerov teorij tipov [26, 36, 42, 47, 97–100, 142], so bile splošne sintaktične definicije postavljene šele nedavno [22, 66, 69, 144]. Te definicije so odprle pot za analizo teorij tipov na novem meta-nivoju, na katerem lahko študiramo, kaj imajo teorije tipov skupnega in kakšne so med njimi interakcije. V tej doktorski disertaciji razvijemo meta-analizo teorij tipov na podlagi sintaktične definicije iz [69], ki je povzeta v prvem delu disertacije (Part 'Finitary Type Theories'). Posebej se posvetimo interakcijam med teorijami tipov in definiramo več pojmov za transformacije ter dokažemo nekaj njihovih meta-teoretičnih lastnosti. Uporabna vrednost transformacij se pokaže v definiciji dopolnitve in dokazu izreka o dopolnitvi. Ker velik del motivacije za študijo teorij tipov prihaja iz njihove uporabe v dokazovalnih pomočnikih, naslovimo tudi meta-teoretični vidik, ki je tesno povezan z implementacijami. To je splošen algoritem za preverjanje enakosti (equality checking algorithm), ki je tudi implementiran v dokazovalnem pomočniku Andromeda 2 [9].

[98]: Martin-Löf (1982), "Constructive mathematics and computer programming"
[100]: Martin-Löf (1998), "An intuitionistic theory of types"
[97]: Martin-Löf (1975), "An intuitionistic theory of types: predicative part"
[99]: Martin-Löf (1984), *Intuitionistic type theory*
[47]: Coquand et al. (1988), "Inductively defined types"
[36]: Church (1940), "A Formulation of the Simple Theory of Types"
[142]: The Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*
[42]: Cohen et al. (2015), "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom"
[26]: Bezem et al. (2019), "The Univalence Axiom in Cubical Sets"
[22]: Bauer et al. (2020), *A general definition of dependent type theories*
[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*
[144]: Uemura (2019), *A General Framework for the Semantics of Type Theory*
[66]: Harper (2021), *An Equational Logical Framework for Type Theories*
[9]: Bauer et al. *The Andromeda proof assistant*

Doktorska disertacija je sestavljena iz treh delov. Prvi del (Part 'Finitary Type Theories') opiše formalno definicijo teorij tipov in pripravi podlago iz meta-izrekov, drugi del (Part 'Transformations of type theories') analizira transformacije in tretji del (Part 'An equality checking algorithm') poda algoritem za preverjanje enakosti. Vsak del ima svoj uvod (poglavja 2, 6 in 12), kjer so natančno podani tudi prispevki disertacije.

1.1. Cilji doktorske disertacije

Namen doktorske disertacije je podati meta-analizo interakcij med teorijami tipov v obliki transformacij, ki zadoščajo naslednjim kriterijem:

- so dovolj splošne, da zajemajo končne teorije tipov (finitary type theories) iz [69],
- so sintaktične narave, da dopuščajo implementacije v dokazovalnih pomočnikih,
- ohranjajo izpeljivost,
- zajemajo nekatere uporabne primere transformacij,

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

in razviti meta-teorijo teorij tipov za oblikovanje splošnega algoritma za preverjanje enakosti, ki

- deluje za končne teorije tipov (finitary type theories) iz [69],
- zadošča izreku skladnosti,

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

- dopušča implementacije,
- deluje po pričakovanjih na teorijah tipov, ki jih srečamo v praksi.

Prvi cilj izpolnimo z definicijami pojmov *sintaktične transformacije*, *transformacije teorij tipov* in *dopolnitvene preslikave*, za katere dokažemo, da ustrezajo želenim lastnostim, in pokažemo njihovo uporabnost v izreku o dopolnitvi. Drugi cilj dosežemo z oblikovanjem pogoja *objektne obrnljivosti*, ki identificira primerna pravila za uporabo v algoritmu za preverjanje enakosti, in z algoritmom, ki zadošča izreku skladnosti in je implementiran v dokazovalnem pomočniku Andromeda 2.

2. Končne teorije tipov

V prvem delu disertacije povzamemo sintakso za končne teorije tipov, kot jo uvajata Haselwarter in Bauer v [69]. V poglavju 2 povzamemo ključne točke v razvoju teorij tipov in primerjamo definicijo končnih teorij tipov s sorodnimi definicijami splošnih teorij tipov [22, 66, 144].

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

[22]: Bauer et al. (2020), *A general definition of dependent type theories*
[144]: Uemura (2019), *A General Framework for the Semantics of Type Theory*
[66]: Harper (2021), *An Equational Logical Framework for Type Theories*

2.1. Sintaksa končnih teorij tipov

Kot v [69] teorije tipov predstavimo s sintaktičnimi konstrukcijami (poglavje 3). Meta-izreke nato dobimo z analizo abstraktne sintakse. Glavna motivacija za ta pristop je implementacija v dokazovalnih pomočnikih kot je Andromeda 2. Definicija teorij tipov, ki jo podamo, zajema odvisne teorije tipov v stilu Martin-Löfa. To so teorije, ki slonijo na štirih *vrstah sodb*:

- "A type" pove, da je A tip,
- "$t : A$" pove, da je term t tipa A,
- "$A \equiv B$ by $\star_{\mathsf{Ty}}$" pove, da sta tipa A in B enaka in
- "$s \equiv t : A$ by $\star_{\mathsf{Tm}}$" pove, da sta terma s in t enaka pri tipu A,

in na *hipotetičnih sodbah* $\Theta; \Gamma \vdash \mathcal{J}$, ki se nahajajo v kontekstu spremenljivk Γ in kontekstu metaspremenljivk Θ. V sodbah nastopajo izrazi za tipe in terme. *Izraz za tip* je oblike $\mathsf{S}(e_1, \ldots, e_n)$, kjer je S *primitiven simbol* uporabljen na argumentih $e_1, \ldots, e_n$, ali pa je oblike $\mathsf{M}(t_1, \ldots, t_n)$, kjer je M *metaspremenljivka* uporabljena na termih. *Izraz za term* je lahko spremenljivka, uporaba primitivnega simbola na argumentih, ali pa uporaba metaspremenljivke na termih. Celoten opis sintakse se nahaja v poglavju 3.

Ta reperezentacija zajema širok nabor teorij tipov, kot na primer intencionalna in ekstenzionalna Martin-Löfova teorija tipov (z univerzumi v stilu Tarskega), homotopska teorija tipov, Churchev λ-račun s preprostimi tipi in mnoge druge teorije tipov. Lahko je najti tudi primere teorij tipov, ki ne ustrezajo naši definiciji, na primer v kubičnih teorijah tipov ima interval posebne vrste sodbo, polimorfni λ-računi kvantificirajo čez vse tipe, itd.

2.2. Teorije tipov

Ključni sestavni del teorije tipov je deduktivni sistem (razdelek 4.1), ki ga podamo s pravili, predlogami, ki generirajo zaprta pravila za izpeljevanje sodb. Teorije tipov opišemo v treh stopnjah. Začnemo s surovimi pravili in teorijami tipov. *Surovo pravilo* je hipotetična sodba oblike $\Theta; [] \vdash \mathcal{J}$, ki jo zapišemo[1] kot

$$\Theta \Longrightarrow \mathcal{J}.$$

1: Včasih za lažje razumevanje še vedno uporabljamo tradicionalen način pisanja pravil v obliki ulomkov.

Elementi Θ so premise pravila, $\mathcal{J}$ pa je zaključek. Pravilo je *objektno pravilo*, kadar je $\mathcal{J}$ objektna sodba, in *pravilo za enakost*, kadar je $\mathcal{J}$ sodba enakosti.

Primer 2.1 Tradicionalno obliko pravila vpeljave za odvisne produkte

$$\frac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma, x{:}A \vdash B \ \mathsf{type}}{\Gamma \vdash \Pi(A, \{x\}B) \ \mathsf{type}}$$

lahko prevedemo v obliko za surovo pravilo kot

$$\mathsf{A}{:}(\square \ \mathsf{type}), \mathsf{B}{:}(\{x{:}\mathsf{A}\} \ \square \ \mathsf{type}) \Longrightarrow \Pi(\mathsf{A}, \{x\}\mathsf{B}(x)) \ \mathsf{type}.$$

Pri tem je $\square$ sestavni del sintaktičnega pojma *meje* (boundary), ki opiše vrsto metaspremenljivke.

Surovo teorijo tipov sestavljajo strukturna pravila, ki so prisotna v vsaki teoriji tipov, in specifična pravila.

Definicija 2.2 *Surova teorija tipov* $\mathcal{T}$ nad signaturo Σ je družina surovih pravil nad Σ, ki jim pravimo *specifična pravila* teorije $\mathcal{T}$. *Pridružen deduktivni sistem* teoriji $\mathcal{T}$ je sestavljen iz:

1. *strukturnih pravil* nad Σ:
   a) pravila za *spremenljivke*, *metaspremenljivke* in *abstrakcije* (definicija 4.2.3, slika 4.1),
   b) pravila za *enakost* (slika 4.2),
   c) pravila za *meje* (slika 4.3);
2. instanciacije specifičnih pravil teorije $\mathcal{T}$;
3. za vsako specifično objektno pravilo teorije $\mathcal{T}$, instanciacije pripadajočih pravil kongruence (definicija 4.2.2).

Definicija surove teorije tipov se nahaja v definiciji 4.3.1.

Ker surove teorije tipov ne postavljajo nobenih pogojev za dobro tipiziranost premis in zaključka pravil, uvedemo končne teorije tipov.

Definicija 2.3 Surovo pravilo $\Theta \Longrightarrow \mathcal{C}\boxed{e}$ je *končno pravilo* za surovo teorijo tipov $\mathcal{T}$, kadar sta $\vdash \Theta \ \mathsf{mctx}$ in $\Theta; [] \vdash \mathcal{C}$ izpeljivi sodbi. *Končna teorija tipov* je surova teorija tipov $\mathcal{T} = (T_i)_{i \in \mathrm{I}}$, za katero obstaja dobro osnovana urejenost $(\mathrm{I}, \sqsubset)$, pri kateri je vsako pravilo T_i končno v fragmentu $(T_j)_{j \sqsubset i}$.

Definicija končne teorije tipov se nahaja v definiciji 4.4.3.

Končne teorije tipov že zajemajo primere, ki si jih želimo opisati, pa tudi nekatere bolj nenavadne primere.

Primer 2.4 Končne teorije tipov imajo mnoge lastnosti, ki si jih želimo, vendar pa se lahko obnašajo "nestandardno". Naj bodo N, O in S konstanta za tip, konstanta za term in eniški simbol za term. Naslednja pravila zadoščajo pogojem za končno teorijo tipov

$$[] \Longrightarrow \mathsf{N\ type}, \qquad [] \Longrightarrow \mathsf{O} : \mathsf{N}, \qquad \mathsf{n}{:}(\square : \mathsf{N}) \Longrightarrow \mathsf{S(S(n))} : \mathsf{N}$$

Vendar pa je tretje pravilo nekoliko težavno, saj postulira sestavljen term S(S(n)).

V izogib takšnim nenavadnim primerom teorij tipov uvedemo še standardne teorije tipov, ki poskrbijo, da ima vsak simbol iz signature svoje pravilo.

Definicija 2.5 Naj bo

$$\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{C}$$

surovo objektno pravilo za mejo nad Σ. *Pridruženo simbolno pravilo* za $\mathsf{S} \notin |\Sigma|$ je surovo pravilo

$$\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow \mathscr{C}[\mathsf{S}(\widehat{\mathsf{M}}_1, \ldots, \widehat{\mathsf{M}}_n)]$$

nad razširjeno signaturo $\langle \Sigma, \mathsf{S} \mapsto (c, [\mathsf{ar}(\mathscr{B}_1), \ldots, \mathsf{ar}(\mathscr{B}_n)]) \rangle$, kjer je $\widehat{\mathsf{M}}$ *generična uporaba* metaspremenljivke M s pridruženo mejo $\mathscr{B}$, definirana kot

1. $\widehat{M} = \{x_1\} \cdots \{x_k\}\mathsf{M}(x_1, \ldots, x_k)$, če je $\mathsf{ar}(\mathscr{B}) = (c, k)$ in $c \in \{\mathsf{Ty}, \mathsf{Tm}\}$,
2. $\widehat{M} = \{x_1\} \cdots \{x_k\}\star$, če je $\mathsf{ar}(\mathscr{B}) = (c, k)$ in $c \in \{\mathsf{EqTy}, \mathsf{EqTm}\}$.

Definicija 2.6 Končna teorija tipov je *standardna*, če so njena specifična objektna pravila simbolna pravila in ima vsak simbol natanko eno pridruženo pravilo.

2.3. Meta-izreki

Za surove, končne in standardne teorije tipov veljajo pričakovani meta-izreki, kot so dopustnost substitucije in enakosti substitucij (izrek 5.1.3), dopustnost instanciacije metaspremenljivk (izrek 5.1.4) in enakosti instanciacij (izrek 5.1.5), izpeljivost predpostavk (izrek 5.1.6), principi inverzije (izrek 5.2.2) in enoličnost tipiziranja (izrek 5.2.3). V razdelku 5.3 dokažemo še dodatne meta-izreke, ki jih uporabimo pri algoritmu za preverjanje enakosti.

2.4. Prispevki

Namen prvega dela doktorske disertacije je opisati ozadje meta-analize in uvesti notacije, ki jih uporabljamo tekom disertacije. Definicije sintaktičnih entitet (poglavje 3) in teorij tipov (poglavji 4 in 5) sta podala Haselwarter in Bauer v [69] in so opisane v obliki, ki je bila objavljena v [24].

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

[24]: Bauer et al. (2021), *An extensible equality checking algorithm for dependent type theories*

Originalni prispevki so dodatni meta-izreki: izrek 5.1.5 o sodbeno enakih instanciacijah je bil dokazan skupaj s Haselwarterjem in Bauerjem. Izreki iz razdelka 5.3 so prav tako originalni prispevek:

- meta-izreki o naravnem tipu: trditev 5.3.1, posledica 5.3.2, posledica 5.3.3,
- meta-izreki o sodbeno enakih instanciacijah: lema 5.3.4, lema 5.3.5 in trditev 5.3.6.

3. Transformacije

Drugi del doktorske disertacije je posvečen transformacijam med teorijami tipov. Motivacija za razvoj transformacij izhaja iz uporabe teorij tipov v dokazovalnih pomočnikih. Ko se lotimo formalizacije dokazov se najprej srečamo z dilemo, kateri dokazovalni pomočnik uporabiti. Tudi med dokazovalnimi pomočniki, ki slonijo na teorijah tipov, je veliko možnosti [2, 5, 43, 45, 58, 78, 108, 133, 141, 146]. Izbira lahko temelji na več faktorjih, od izkušenj s posameznim dokazovalnim pomočnikom, narave problema, ki ga formaliziramo, ekspresivnosti teorije tipov, razpoložljivosti potrebnih knjižnic, zmogljivosti dokazovalnega pomočnika, itd. Ko se enkrat odločimo in začnemo formalizacijo, si je ponavadi težko premisliti in nadaljevati v drugem dokazovalnem pomočniku, saj moramo ves razvoj dokazov ročno prevesti v novo teorijo tipov in nov jezik, če je takšna prevedba sploh mogoča.

[45]: (2021), *The Coq proof assistant, version 2021.02.2*
[5]: (2021), *The Agda proof assistant*
[108]: Moura et al. (2015), "The Lean Theorem Prover (System Description)"
[133]: Sozeau et al. (2019), "Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq"
[58]: Gilbert et al. (2019), "Definitional proof-irrelevance without K"
[2]: Abel et al. (2017), "Decidability of Conversion for Type Theory in Type Theory"
[146]: Vezzosi et al. (2019), "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types"
[141]: The RedPRL Development Team (2020), *The 'redtt' theorem prover.*
[43]: Cohen et al. (2018), *The 'cubicaltt' theorem prover.*
[78]: Isaev (2021), *Arend Standard Library*

Korak proti analizi kompatibilnosti formalizacij je študija kompatibilnosti teorij tipov. Pogosto dokaz uporablja le fragment teorije tipov in ko je tak fragment skupen drugi teoriji je prevedba morda izvedljiva. Študije transformacij formalnih sistemov so se začele že davno, nekaj najbolj relevantnih je opisanih v razdelku 11.1.

V tem delu doktorske disertacije predlagamo tri pojme transformacije: *sintaktične transformacije* (definicija 8.2.1), *transformacije teorij tipov* (definicija 9.1.2) in *dopolnitveno preslikavo* (definicija 10.1.2). Da bi lažje opisali, kako sintaktične transformacije delujejo na sintakso, v poglavju 7 spomnimo na definicijo relativne monade in opišemo posebne vrste monad za sintakso teorij tipov (razdelek 7.1).

3.1. Sintaktične transformacije

Ko preučujemo transformacije med teorijami tipov, imamo več možnosti. Lahko slikamo simbole v simbole, simbole v izraze ali celo sodbe v sodbe drugačne vrste, kot na primer v [152]. Osredotočimo se predvsem na prvi dve možnosti.

[152]: Winterhalter et al. (2019), "Eliminating Reflection from Type Theory"

Definicija 3.1 *Preimenovanje simbolov* je preslikava med signaturama $f : \Sigma_1 \to \Sigma_2$, ki ohranja članosti simbolov: za vsak $\mathsf{S} \in \Sigma_1$ velja

$$\mathsf{ar}(\mathsf{S}) = \mathsf{ar}(f(\mathsf{S})).$$

Primer 3.2 Preimenovanja simbolov ponavadi vidimo kot utemeljitev, da imena simbolov niso pomembna. Na primer, ni pomembno, kako označimo tip naravnih števil, dokler postavimo prava pravila. Zlahka si zamislimo dve signaturi

$$\begin{aligned}
\Sigma_1 =&[\mathbb{N} : (\mathsf{Ty}, []),\\
&\mathsf{z} : (\mathsf{Tm}, []),\\
&\mathsf{s} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])]\\
\Sigma_2 =&[\mathsf{Nat} : (\mathsf{Ty}, []),\\
&\mathsf{zero} : (\mathsf{Tm}, []),\\
&\mathsf{succ} : (\mathsf{Tm}, [(\mathsf{Tm}, 0)])],
\end{aligned}$$

ki obe predstavljata naravna števila. Preimenovanje simbolov

$$\begin{aligned}
\Sigma_1 &\to \Sigma_2\\
\mathbb{N} &\mapsto \mathsf{Nat}\\
\mathsf{z} &\mapsto \mathsf{zero}\\
\mathsf{s} &\mapsto \mathsf{succ}
\end{aligned}$$

pokaže korespondenco med obema signaturama.

Kompozitum $g \circ f$ preimenovanj simbolov $f : \Sigma_1 \to \Sigma_2$ in $g : \Sigma_2 \to \Sigma_2$ kot preslikav je prav tako preimenovanje simbolov, saj f in g ohranjata članosti. Kompozitum je očitno asociativen in identična preslikava je identično preimenovanje simbolov. Tako dobimo kategorijo signatur.

Preimenovanja simbolov so že prva definicija sintaktičnih transformacij, vendar je ta definicija zelo restriktivna. Potrebujemo bolj fleksibilno definicijo, ki bo zajela več uporabnih primerov transformacij.

Definicija 3.3 *Sintaktična transformacija* $f : \Sigma_1 \to \Sigma_2$ je preslikava, ki slika simbol $\mathsf{S} \in \Sigma_1$ v izraz iz $\mathsf{Expr}_{\Sigma_2}(\mathsf{cl}(\mathsf{S}), \vartheta; [])$, kjer je članost $\mathsf{ar}(\mathsf{S}) = (c, \vartheta)$.

Sintaktične transformacije še vedno ohranjajo nekaj strukture. Simbole slikamo v izraze enakega sintaktičnega razreda in članost simbola definira obliko metaspremenljivk v izrazu. Poseben primer sintaktične transformacije je identična transformacija, ki slika simbole v njihove generične uporabe. Več primerov sintaktičnih transformacij je v razdelkih 8.2 in 9.3.

Vsaka sintaktična transformacija deluje na izrazih na naraven način. Tako dvignemo sintaktično transformacijo na preslikavo med izrazi nad signaturami.

Definicija 3.4 *Delovanje* sintaktične transformacije $f\colon \Sigma_1 \to \Sigma_2$ je preslikava

$$f_*\colon \mathsf{Expr}_{\Sigma_1}(\mathsf{c}, \vartheta; \gamma) \to \mathsf{Expr}_{\Sigma_2}(\mathsf{c}, \vartheta; \gamma)$$

za vsak sintaktični razred c, obliko metaspremenljivk ϑ in obliko spremenljivk γ, podana z

$$f_*\mathsf{a} = \mathsf{a}, \qquad f_* x = x, \qquad f_*\star = \star,$$
$$f_*(\{x\}e) = \{x\}(f_*e),$$
$$f_*(\mathsf{M}(t_1, \ldots, t_m)) = \mathsf{M}(f_*t_1, \ldots, f_*t_m)$$
$$f_*(\mathsf{S}(e_1, \ldots, e_n)) = \langle \mathsf{M}_1 \mapsto f_*e_1, \ldots \mathsf{M}_n \mapsto f_*e_n \rangle_* f(\mathsf{S})$$

kjer so $\mathsf{M}_1, \ldots, \mathsf{M}_n$ metaspremenljivke iz ϑ v $\mathsf{ar}(\mathsf{S}) = (\mathsf{cl}(\mathsf{S}), \vartheta)$.

Kompozitum $g \circ f\colon \Sigma_1 \to \Sigma_3$ sintaktičnih preslikav $f\colon \Sigma_1 \to \Sigma_2$ in $g\colon \Sigma_2 \to \Sigma_3$ je definiran kot

$$(g \circ f)(\mathsf{S}) = g_*(f\ \mathsf{S}).$$

V razdelku 8.3 pokažemo, da sintaktične transformacije tvorijo relativno monado za sintakso. Od tod izhajajo pričakovane lastnosti sintaktičnih transformacij: delovanje ohranja identiteto (lema 8.3.1), delovanje interagira s substitucijami (lema 8.3.2) in z instancijacijami (lema 8.3.3), delovanje ohranja kompozitum (lema 8.3.4) in kompozitum je asociativen (posledica 8.3.5).

3.2. Transformacije teorij tipov

Sintaktične transformacije so dobra podlaga za pojem transformacij med teorijami tipov, vendar ne upoštevajo zelo pomembnega aspekta teorij tipov: deduktivnega sistema in izpeljivosti. Zato definicijo transformacije nadgradimo v transformacije teorij tipov.

Definicija 3.5 *Transformacija teorij tipov* $f\colon \mathcal{T} \to \mathcal{U}$ je sintaktična transformacija $f\colon \Sigma_{\mathcal{T}} \to \Sigma_{\mathcal{U}}$, tako da za vsako pravilo $\Theta \Longrightarrow \mathcal{j}$ v teoriji $\mathcal{T}$ podamo izpeljavo $\mathcal{D}$ od $f_*\Theta \Longrightarrow f_*\mathcal{j}$ v teoriji $\mathcal{U}$.

Tako definirane transformacije ohranjajo izpeljivost, kot veli naslednji izrek.

Izrek 3.6 Naj bo $f\colon \mathcal{T} \to \mathcal{U}$ transformacija teorij tipov in $\Theta; \Gamma \vdash \mathcal{J}$ izpeljiva sodba v teoriji $\mathcal{T}$. Potem je

$$f_*\Theta; f_*\Gamma \vdash f_*\mathcal{J}$$

izpeljiva sodba v teoriji $\mathcal{U}$. Podobno velja tudi za izpeljive meje.

Dokaz se nahaja pod izrekom 9.1.3.

Podobno kot pri sodbeno enakih instanciacijah lahko definiramo sodbeno enake transformacije.

Definicija 3.7 Transformaciji teorij tipov $f : \mathcal{T} \to \mathcal{U}$ in $g : \mathcal{T} \to \mathcal{U}$ sta *sodbeno enaki*, če je za vsako objektno pravilo $R = \Theta \Longrightarrow \mathcal{b}\boxed{e}$ v teoriji $\mathcal{T}$ sodba

$$f_*\Theta; [] \vdash_{\mathcal{U}} (f_*\mathcal{b})\boxed{f_*e \equiv g_*e}$$

izpeljiva.

Delovanje sodbeno enakih transformacij daje sodbe enakosti kot je razvidno iz trditev 9.1.6, 9.1.7, 9.1.8, 9.1.9 in 9.1.10.

Z definicijo transformacij lahko postavimo *kategorijo teorij tipov*, kjer so objekti (surove) teorije tipov in morfizmi transformacije. V razdelku 9.2 pokažemo, da ima ta kategorija začetni objekt in koprodukte.

Ustreznost definicije transformacij teorij tipov pokažejo primeri, ki jih definicija zajema. Poleg identične transformacije in preimenovanj simbolov v razdelku 9.3 pokažemo še, da je primer tudi Curry-Howardova korespondenca [48, 49, 74, 149], ki prevede logiko prvega reda v Martin-Löfovo teorijo tipov in pri tem klasično logiko prevede v intuicionistično logiko. Transformacije nam tudi omogočajo študijo konzervativnih razširitev z definicijo (primer 9.3.5). Uporabnost transformacij pa najbolj pokaže primer dopolnitve.

[149]: Wadler (2015), "Propositions as Types"
[48]: Curry (1934), "Functionality in Combinatory Logic"
[49]: Curry et al. (1958), *Combinatory logic. Vol. I*
[74]: Howard (1980), "The Formulae-as-Types Notion of Construction"

3.3. Izrek o dopolnitvi

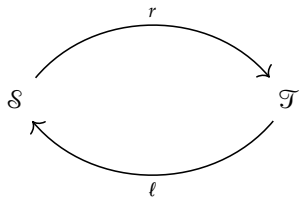
Ko oblikujemo teorijo tipov za uporabo v dokazovalnem pomočniku, se srečamo z dilemo, kako polna naj bo sintaksa. Če so vsi termi polno označeni s tipi, jih je lažje algoritmično procesirati in imajo lepe metateoretične lastnosti. Vendar pa je takšna sintaksa hitro neobvladljiva, zato se v praksi uporablja bolj ekonomična sintaksa, kjer so nekatere oznake tipov izpuščene.

To neskladje se pogosto rešuje tako, da oblikujemo dve teoriji tipov: prva, imenujmo jo $\mathcal{S}$, vsebuje vse oznake tipov in ponavadi prebiva v jedru dokazovalnega pomočnika, druga, ekonomična teorija $\mathcal{T}$, pa je namenjena uporabnikom. Teorijo $\mathcal{T}$ nato sistem prevede v $\mathcal{S}$ z uporabo *dopolnjevalca*[2]. Ta pojav lahko opazimo v praksi, na primer ko dokazovalna pomočnika Agda [5] in Coq [45] določita implicitne argumente.

2: Dopolnjevalec je natančno definiran v definiciji 10.3.2

[5]: (2021), *The Agda proof assistant*
[45]: (2021), *The Coq proof assistant, version 2021.02.2*

Proces dopolnitve lahko opišemo z naslednjim diagramom.



Začnemo s končno teorijo tipov $\mathcal{T}$, ki predstavlja ekonomično verzijo. Verzija teorije s polnimi oznakami tipov je *standardna* teorija tipov $\mathcal{S}$, saj pri standardni teoriji pravila za simbole natančno zapišejo vse

premise. Iz teorije $\mathcal{S}$ v teorijo $\mathcal{T}$ nas povede "pozabljiva" transformacija teorij tipov r, ki ji pravimo *retrogradna transformacija*. Ta transformacija pobriše oznake tipov, a je še vedno konzervativna. Bolj zanimiva pa je preslikava ℓ v obratni smeri, ki ji pravimo *dopolnitvena preslikava* in slika izpeljave iz teorije $\mathcal{T}$ v sodbe v teoriji $\mathcal{S}$ ter deluje kot prerez retrogradne transformacije. Natančna definicija dopolnitvene preslikave se nahaja v definiciji 10.1.2.

Pri danih pogojih je retrogradna transformacija surjektivna na izpeljivih sodbah (posledica 10.1.4), dopolnitvena preslikava pa je določena do sodbene enakosti natančno (posledica 10.1.6). V razdelku 10.1.2 dokažemo, da ima dopolnitev naslednjo univerzalno lastnost.

Izrek 3.8 Naj bo $\mathcal{T}$ končna teorija tipov in $(\mathcal{S}_1, r_1, \ell_1)$ ter $(\mathcal{S}_2, r_2, \ell_2)$ dopolnitvi teorije $\mathcal{T}$. Potem obstaja konzervativna transformacija teorij tipov $f\colon \mathcal{S}_1 \to \mathcal{S}_2$ z dopolnitveno preslikavo $\ell_f\colon \mathcal{S}_2 \to \mathcal{S}_1$, da velja $r_2 \circ f = r_1$. Transformacija f je enolična do sodbene enakosti natančno.

Univerzalna lastnost dopolnitve nam pove, da če dopolnitev obstaja, je enolična (do sodobene enakosti natančno). Naslednji izrek o dopolnitvi pa pove, da vsaka končna teorija tipov ima dopolnitev, ki jo v dokazu izreka (razdelka 10.2.1 in 10.2.2) tudi konstruiramo.

Izrek 3.9 Vsaka končna teorija tipov ima dopolnitev.

V razdelku 10.3 preučimo nekaj algoritmičnih lastnosti dopolnitve in kako se le-ta povezuje s preverjanjem tipov ter enakosti.

Izrek 3.10 Končna teorija $\mathcal{T}$ ima dopolnjevalec, če in samo če ima $\mathcal{T}$ odločljivo preverjanje tipov in enakosti.

3.4. Prispevki

Glavna dva prispevka sta definicija transformacije teorij tipov in dokaz izreka o dopolnitvi (izrek 10.2.1). Vse konstrukcije in dokazi so konstruktivni.

Definicijo transformacij gradimo postopoma:

- Opišemo splošno shemo relativnih monad za sintakso (razdelek 7.1).
- Definiramo pojem preimenovanja simbolov (definicija 8.1.1).
- Definiramo sintaktične transformacije (definicija 8.2.1) in dokažemo, da tvorijo relativno monado nad kategorijo signatur (razdelek 8.3).
- Definiramo transformacije teorij tipov (definicija 9.1.2) in dokažemo, da ohranjajo izpeljivost (izrek 9.1.3).
- Dokažemo nekaj meta-teoretičnih lastnosti sodbeno enakih transformacij (trditev 9.1.6, posledica 9.1.10).
- Definiramo kategorijo teorij tipov in transformacij ter pokažemo, da ima začetni objekt (trditev 9.2.2) in koprodukte (trditev 9.2.3).
- Pokažemo, da je Curry-Howardova korespondenca primer transformacije (primer 9.3.1, dodatek A).

- Z uporabo meta-teoretičnih lastnosti transformacij med teorijami tipov dokažemo, da je razširitev z definicijo konzervativna (primer 9.3.5).

Podamo matematično definicijo *dopolnitve* (definicija 10.1.3) in dokažemo, da ima univerzalno lastnost (izrek 10.1.7). Formuliramo in dokažemo izrek o dopolnitvi (izrek 10.2.1, razdelka 10.2.1 in 10.2.2). Analiziramo algoritmične lastnosti dopolnitve:

- Definiramo *dopolnjevalec*, algoritem za dopolnitev (definicija 10.3.2).
- Povežemo odločljivo preverjanje tipov in enakosti z odločljivim preverjanjem (samo) enakosti v standardnih teorijah tipov (trditev 10.3.5).
- Povežemo izračunljivo dopolnitev z odločljivim preverjanjem tipov in enakosti (izrek 10.3.9).
- Povežemo preverjanje tipov in enakosti končne teorije tipov s preverjanjem v dopolnitvi (izrek 10.3.10, posledica 10.3.11).

4. Algoritem za preverjanje enakosti

Algoritmi za preverjanje enakosti so ključne komponente dokazovalnih pomočnikov, ki temeljijo na teorijah tipov [2, 5, 45, 58, 108, 133]. Uporabnika osvobodijo bremena dokazovanja trivialnih sodbenih enakosti in zagotavljajo algoritme za računanje z normalizacijo. Nekateri sistemi gredo korak dalje [39, 50] in dopuščajo uporabnikom, da razširijo vgrajene algoritme za preverjanje enakosti.

[45]: (2021), *The Coq proof assistant, version 2021.02.2*
[5]: (2021), *The Agda proof assistant*
[108]: Moura et al. (2015), "The Lean Theorem Prover (System Description)"
[133]: Sozeau et al. (2019), "Coq Coq correct! Verification of Type Checking and Erasure for Coq, in Coq"
[58]: Gilbert et al. (2019), "Definitional proof-irrelevance without K"
[2]: Abel et al. (2017), "Decidability of Conversion for Type Theory in Type Theory"

Situacija je manj ugodna v dokazovalnem pomočniku, ki podpira splošne teorije tipov, kot jih Andromeda 2 [9, 20]. Tam v splošnem ni nujno, da je na voljo algoritem za preverjanje enakosti. Kljub temu pa bi moral dokazovalni pomočnik zagotoviti podporo za algoritme za preverjanje enakosti, ki so preprosti za uporabo in dobro delujejo v pogostih primerih. S tem namenom smo razvili in implementirali splošen algoritem za preverjanje enakosti, ki zadošča izreku skladnosti.

[50]: *The Dedukti logical framework*
[39]: Cockx et al. (2016), "Sprinkles of extensionality for your vanilla type theory"

Algoritem deluje za standardne teorije tipov (definicija 4.4.5) in je zasnovan po vzoru algoritmov za preverjanje enakosti [3, 136], ki imajo dve fazi: najprej nastopi tipsko vodena faza, kjer uporabljamo pravila ekstenzionalnosti, nato pa nastopi faza normalizacije, ki uporablja pravila za izračun (β-pravila). Na običajnih primerih (Martin-Löfova teorija tipov, λ-račun, sistem F) se algoritem vede enako kot standardni algoritmi za preverjanje enakosti.

[136]: Stone et al. (2006), "Extensional equivalence and singleton types"
[3]: Abel et al. (2012), "On Irrelevance and Algorithmic Equality in Predicative Type Theory"

[9]: Bauer et al. *The Andromeda proof assistant*
[20]: Bauer et al. (2018), "Design and Implementation of the Andromeda Proof Assistant"

4.1. Vzorci in objektno obrnljiva pravila

Algoritem za preverjanje enakosti izpelje enačbo z uporabo (instanciacijo) pravil teorije tipov. Da bi algoritem lahko določil, ali je pravilo primerno za uporabo in kako se končna enačba ujema z vzorcem, določimo pogoje, pri katerih pravila lahko uporabljamo.

Definicija 4.1 Pravilo $\Xi \Longrightarrow \mathcal{j}$ je *deterministično*, če za vsako sodbo

$\Theta;\Gamma \vdash \mathcal{J}'$ obstaja največ ena instanciacija I metakonteksta Ξ nad $\Theta;\Gamma$, da velja $I_*\mathcal{J} = \mathcal{J}'$.

V algoritmu bomo uporabljali samo deterministična pravila. Nadalje si pomagamo z vzorci.

Definicija 4.2 *Vzorci* so izrazi v katerih se vsaka metaspremenljivka pojavi največ enkrat bodisi brez argumentov $\mathsf{M}()$, ali kot argument oblike $\{\vec{x}\}\mathsf{M}(\vec{x})$, kjer so $\vec{x}$ edine vezane spremenljivke. Vzorci so opisani s slovnico na spodnji sliki.

$$
\begin{array}{rll}
\text{Vzorec tip } P & ::= \mathsf{M}() \mid \mathsf{S}(q_1, \ldots, q_n) & \text{če } \mathsf{mv}(q_i) \cap \mathsf{mv}(q_j) = \emptyset \text{ za } i \neq j \\
\text{Vzorec term } p & ::= \mathsf{S}(q_1, \ldots, q_n) & \text{če } \mathsf{mv}(q_i) \cap \mathsf{mv}(q_j) = \emptyset \text{ za } i \neq j \\
\text{Vzorec argument } q & ::= \{\vec{x}\}\mathsf{M}(\vec{x}) \mid P \mid p &
\end{array}
$$

Vzorci nam dajejo sintaktični kriterij za želene pogoje na pravilih. Če je $\Xi \Longrightarrow \mathcal{J}\boxed{p}$ pravilo, kjer je p vzorec, ki zajame vse objektne metaspremenljivke iz Ξ, potem je pravilo deterministično.

Notacija $\mathsf{mv}(e)$ označuje množico metaspremenljivk, ki se pojavijo v izrazu e.

Bistveni pogoj za uporabo pravil v algoritmu za preverjanje enakosti je objektna obrnljivost. Izpeljivo pravilo $\Xi \Longrightarrow \mathcal{J}$ je *objektno obrnljivo*, če velja naslednje: če je I instanciacija metakonteksta Ξ nad $\Theta;[]$ in velja $\vdash \Theta\ \mathsf{mctx}$ ter $|\Xi| \cap |\Theta| = \emptyset$, ter je $\Theta;[] \vdash I_*\mathcal{J}$ izpeljiva sodba, potem je I izpeljiva instanciacija do metaspremenljivk za enačbe natančno.

Natančna definicija objektne obrnljivosti je v definiciji 13.2.2.

Poleg vzorcev potrebujemo še dodaten sintaktični pogoj za objektno obrnljivost: izpeljivost, ki je naravna za spremenljivke. Pravimo, da je izpeljiva objektna sodba *naravna za spremenljivke*, če ima izpeljavo, v kateri nobeni uporabi pravil TT-Meta in TT-Var v naslednjem koraku ne sledi pravilo za pretvorbo TT-Conv-Tm, razen če se pravilo za (meta)spremenljivko uporabi v pod-izpeljavi za sodbo enakosti.

Formalno je pogoj naravnosti za spremenljivke podan v definiciji 13.2.6.

Tako dobimo zadosten sintaktični pogoj za objektno obrnljivost.

Trditev 4.3 V standardni teoriji tipov naj bo $\Xi \Longrightarrow \mathcal{J}\boxed{p}$ izpeljivo končno objektno pravilo, ki je naravno za spremenljivke. Če je p vzorec, ki vsebuje vse objektne spremenljivke iz Ξ, potem je pravilo objektno obrnljivo.

4.2. Pravila za izračun in ekstenzionalnost

Algoritem za preverjanje enakosti uporablja dve vrsti pravil za enačbe. Začnemo s pravili, ki usmerjajo normalizacijo.

Definicija 4.4 Izpeljivo končno pravilo $\Theta \Longrightarrow A \equiv B$ je *pravilo za izračun za tip*, če je pravilo $\Theta \Longrightarrow A\ \mathsf{type}$ deterministično in objektno obrnljivo.

Definicija 4.5 Izpeljivo končno pravilo $\Theta \Longrightarrow u \equiv v : A$ je *pravilo za izračun za term*, če je u uporaba simbola za term in je pravilo $\Theta \Longrightarrow u : \tau_{\Theta;[]}(u)$ deterministično in objektno obrnljivo.

Pogoj objektne obrnljivosti v pravilih za izračun lahko preverjamo sintaktično z vzorci in naravnostjo za spremenljivke, kot je razvidno iz trditve 14.1.3.

Primer 4.6 Primer pravila za izračun je β-pravilo za uporabo funkcij

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \{x{:}\mathsf{A}\}\ \mathsf{B}\ \mathsf{type} \qquad \vdash \{x{:}\mathsf{A}\}\ \mathsf{s} : \mathsf{B}(x) \qquad \vdash \mathsf{t} : \mathsf{A}}{\vdash \mathsf{apply}(\mathsf{A}, \{x\}\mathsf{B}(x), \lambda(\mathsf{A}, \{x\}\mathsf{B}(x), \{x\}\mathsf{s}(x)), \mathsf{t}) \equiv \mathsf{s}(\mathsf{t}) : \mathsf{B}(\mathsf{t})},$$

ki ga zaradi tehničnih pogojev lineariziramo v obliko

$$\frac{\begin{array}{c}\vdash \mathsf{A}_1\ \mathsf{type} \qquad \vdash \{x{:}\mathsf{A}_1\}\ \mathsf{B}_1\ \mathsf{type} \\ \vdash \mathsf{A}_2\ \mathsf{type} \qquad \vdash \{x{:}\mathsf{A}_2\}\ \mathsf{B}_2\ \mathsf{type} \\ \vdash \{x{:}\mathsf{A}_2\}\ \mathsf{s} : \mathsf{B}_2(x) \qquad \vdash \mathsf{t} : \mathsf{A}_1 \\ \vdash \mathsf{A}_1 \equiv \mathsf{A}_2 \qquad \vdash \{x{:}\mathsf{A}_1\}\mathsf{B}_1(x) \equiv \mathsf{B}_2(x)\end{array}}{\vdash \mathsf{apply}(\mathsf{A}_1, \{x\}\mathsf{B}_1(x), \lambda(\mathsf{A}_2, \{x\}\mathsf{B}_2(x), \{x\}\mathsf{s}(x)), \mathsf{t}) \equiv \mathsf{s}(\mathsf{t}) : \mathsf{B}_1(\mathsf{t})}.$$

Drugo vrsto pravil algoritem uporablja, da reducira enačbo na možne enačbe.

Definicija 4.7 *Pravilo za ekstenzionalnost* je izpeljivo končno pravilo oblike

$$\Theta, \mathsf{s}{:}(\square : A), \mathsf{t}{:}(\square : A), \Phi \Longrightarrow \mathsf{s} \equiv \mathsf{t} : A$$

tako da Φ vsebuje samo metaspremenljivke za enačbe in je pravilo $\Theta \Longrightarrow A\ \mathsf{type}$ deterministično in objektno obrnljivo.

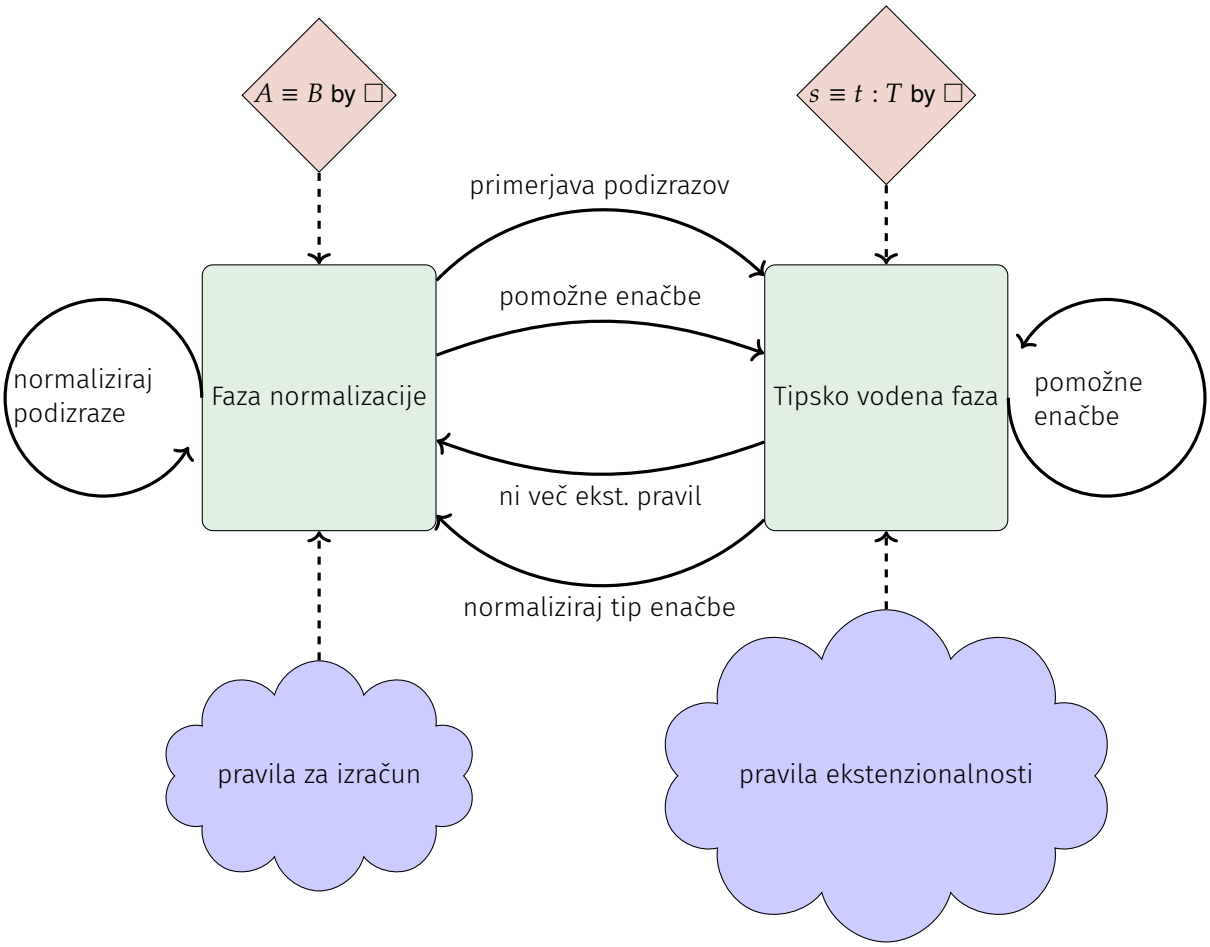
Podobno kot pri pravilih za izračun lahko pravila za ekstenzionalnost preverjamo s sintaktičnim kriterijem (trditev 14.2.2).

Primer 4.8 Pravila za ekstenzionalnost ponavadi pravijo, da so elementi tipa enaki, kadar so enaki njihovi deli. Na primer pravilo za ekstenzionalnost za produkte je

$$\frac{\begin{array}{c}\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{B}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \times \mathsf{B} \qquad \vdash \mathsf{t} : \mathsf{A} \times \mathsf{B} \\ \vdash \mathsf{fst}(\mathsf{A}, \mathsf{B}, \mathsf{s}) \equiv \mathsf{fst}(\mathsf{A}, \mathsf{B}, \mathsf{t}) : \mathsf{A} \qquad \vdash \mathsf{snd}(\mathsf{A}, \mathsf{B}, \mathsf{s}) \equiv \mathsf{snd}(\mathsf{A}, \mathsf{B}, \mathsf{t}) : \mathsf{B}\end{array}}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A} \times \mathsf{B}}.$$

4.3. Opis algoritma

Algoritem za preverjanje enakosti je natančno podan v poglavju 15. Algoritem sestavljata dve fazi: tipsko vodena faza, kjer uporabljamo pravila ekstenzionalnosti, ter faza normalizacije, ki temelji na pravilih za izračun. Algoritem lahko na grobo orišemo z naslednjim diagramom.



Algoritem sprejme kot vhodni podatek izpeljivo mejo in bodisi vrne izpeljivo enačbo, ki ima dano mejo, ali pa ne uspe najti izpeljave. Če je dana meja za enakost tipov, algoritem takoj vstopi v fazo normalizacije, sicer pa najprej nastopi tipsko vodena faza.

Normalizacija prepiše izraz $\mathsf{S}(e_1, \ldots, e_n)$ tako, da normalizira nekatere izmed argumentov $e_1, \ldots, e_n$, ki jim pravimo *glavni argumenti* (principal arguments) in uporabi pravila za izračun ter ponovi postopek. Parametrizirana je z naslednjimi podatki:

1. standardno teorijo tipov $\mathcal{T}$,
2. družino $\mathscr{C}$ pravil za izračun v teoriji $\mathcal{T}$,
3. za vsak simbol S, ki vzame k argumentov, množico $\wp(\mathsf{S}) \subseteq \{1, \ldots, k\}$ njegovih *glavnih argumentov*.

Normalizacija ima tri neodvisne različice:

$\Theta; \Gamma \vdash \mathscr{B}\boxed{e \triangleright e'}$	normaliziraj argument e v e',
$\Theta; \Gamma \vdash \mathscr{b}\boxed{\mathsf{S}(\vec{e}) \triangleright_{\mathrm{p}} \mathsf{S}(\vec{e}')}$	normaliziraj glavne argumente od S,
$\Theta; \Gamma \vdash \mathscr{b}\boxed{e \triangleright_{\mathrm{c}} e'}$	uporabi pravilo za izračun in prepiši e v e'.

V posebnem primeru

$$\Theta; \Gamma \vdash (A \triangleright A') \ \mathsf{type} \qquad \text{in} \qquad \Theta; \Gamma \vdash t \triangleright t' : B$$

izražata dejstvi, da se tip A normalizira v A' in term t v t'. Postopek normalizacije je podan na sliki 15.1.

Preverjanje enakosti poteka v naslednjih medsebojno rekurzivnih fazah:

$\Theta;\Gamma \vdash \mathcal{B}\boxed{e \sim e'}$	e in e' sta enaka argumenta
$\Theta;\Gamma \vdash s \sim_{\mathrm{e}} t : A$	s in t sta ekstenzionalno enaka
$\Theta;\Gamma \vdash s \sim_{\mathrm{n}} t : A$	normalizirana terma s in t sta enaka
$\Theta;\Gamma \vdash A \sim_{\mathrm{n}} B$	normalizirana tipa A in B sta enaka

Prva je splošna primerjava argumentov e in e' pri objektni meji $\mathcal{B}$, druga je *tipsko vodena faza* in tretja je *faza normalizacije*, ki primerja normalizirane izraze. Induktivna specifikacija teh faz se nahaja na sliki 15.2. Faze so parametrizirane s standardno teorijo tipov $\mathcal{T}$, z družino pravil ekstenzionalnosti $\mathcal{E}$ nad $\mathcal{T}$, družino pravil za izračun $\mathcal{C}$ nad $\mathcal{T}$, in specifikacijo glavnih argumentov $\wp$.

Algoritem zadošča izrekoma skladnosti, ki sta dokazana v razdelku 15.3.

Izrek 4.9 (Skladnost normalizacije) Naj bosta v standardni teoriji tipov podani množica pravil za izračun $\mathcal{C}$ in specifikacija glavnih argumentov $\wp$. Za objektni meji $\mathcal{B}$ in $\mathcal{b}$ velja:

1. Če velja $\Theta;\Gamma \vdash \mathcal{B}\boxed{e}$ in $\Theta;\Gamma \vdash \mathcal{B}\boxed{e \triangleright e'}$, potem velja $\Theta;\Gamma \vdash \mathcal{B}\boxed{e \equiv e'}$ in $\Theta;\Gamma \vdash \mathcal{B}\boxed{e'}$.
2. Če velja $\Theta;\Gamma \vdash \mathcal{b}\boxed{e}$ in $\Theta;\Gamma \vdash \mathcal{b}\boxed{e \triangleright_{\mathrm{p}} e'}$, potem velja $\Theta;\Gamma \vdash \mathcal{b}\boxed{e \equiv e'}$ in $\Theta;\Gamma \vdash \mathcal{b}\boxed{e'}$.
3. Če velja $\Theta;\Gamma \vdash \mathcal{b}\boxed{e}$ in $\Theta;\Gamma \vdash \mathcal{b}\boxed{e \triangleright_{\mathrm{c}} e'}$, potem velja $\Theta;\Gamma \vdash \mathcal{b}\boxed{e \equiv e'}$ in $\Theta;\Gamma \vdash \mathcal{b}\boxed{e'}$.

Izrek 4.10 (Skladnost preverjanja enakosti) Naj bodo v standardni teoriji tipov podani družini $\mathcal{C}$ in $\mathcal{E}$ pravil za izračun in ekstenzionalnost ter specifikacija glavnih argumentov $\wp$. Za objektno mejo $\mathcal{B}$ velja:

1. $\Theta;\Gamma \vdash \mathcal{B}\boxed{e \equiv e'}$ velja, če velja

   $$\Theta;\Gamma \vdash \mathcal{B}\boxed{e}, \qquad \Theta;\Gamma \vdash \mathcal{B}\boxed{e'}, \quad \text{in} \quad \Theta;\Gamma \vdash \mathcal{B}\boxed{e \sim e'}.$$

2. $\Theta;\Gamma \vdash u \equiv v : A$ velja, če velja

   $$\Theta;\Gamma \vdash u : A, \qquad \Theta;\Gamma \vdash v : A, \quad \text{in} \quad \Theta;\Gamma \vdash u \sim_{\mathrm{e}} v : A.$$

3. $\Theta;\Gamma \vdash A \equiv B$ velja, če velja

   $$\Theta;\Gamma \vdash A\ \mathsf{type}, \qquad \Theta;\Gamma \vdash B\ \mathsf{type}, \quad \text{in} \quad \Theta;\Gamma \vdash A \sim_{\mathrm{n}} B.$$

4. $\Theta;\Gamma \vdash u \equiv v : A$ velja, če velja

   $$\Theta;\Gamma \vdash u : A, \qquad \Theta;\Gamma \vdash v : A, \quad \text{in} \quad \Theta;\Gamma \vdash u \sim_{\mathrm{n}} v : A.$$

4.4. Implementacija v Andromedi 2

Algoritem je implementiran v dokazovalnem pomočniku Andromeda 2 [9, 20, 23], kjer uporabnik lahko definira in uporablja katero koli standardno teorijo tipov. Andromeda 2 je dokazovalni pomočnik stila LCF, tj. meta-nivojski programski jezik z abstraktnim tipom za sodbe in meje ter z izpeljivimi pravili, ki jih nadzoruje zaupanja vredno jedro, sestavljeno iz približno 4200 vrstic kode v jeziku OCaml. Jedro implementira različico *teorije tipov brez kontekstov*, ki je definirana v [69].

[20]: Bauer et al. (2018), "Design and Implementation of the Andromeda Proof Assistant"
[9]: Bauer et al. *The Andromeda proof assistant*
[23]: Bauer et al. (2020), "Equality Checking for General Type Theories in Andromeda 2"

[69]: Haselwarter et al. (2021), *Finitary type theories with and without contexts*

Implementacijo algoritma za preverjanje enakosti sestavlja približno 1400 vrstic programske kode v jeziku OCaml. Koda ni del jedra Andromede, vendar pa jedro validira vsak korak v sklepanju. Uporabnik mora zgolj podati pravila za enakost, ki jih želi uporabljati, nato pa jih algoritem avtomatično klasificira kot pravila za izračun ali pravila ekstenzionalnosti, zavrne neustrezna pravila in izbere glavne argumente. V razdelku 16.1 je opisanih nekaj primerov uporabe implementiranega algoritma.

4.5. Prispevki

Podamo *splošen algoritem za preverjanje enakosti* (poglavje 15), ki deluje za standardne teorije tipov:

- Podamo definicijo pravil za izračun (definiciji 14.1.1 in 14.1.2) ter pravil ekstenzionalnosti (definicija 14.2.1) preko pogoja objektne obrnljivosti.
- Podamo zadosten sintaktični kriterij za prepoznavanje in uporabo pravil za izračun in ekstenzionalnost.
- Dokažemo, da algoritem zadošča izreku o skladnosti (razdelek 15.3).
- Algoritem implementiramo v dokazovalnem pomočniku Andromeda 2.
- Pokažemo vrsto primerov uporabe algoritma v dokazovalnem pomočniku Andromeda 2 (razdelek 16.1).