UNIVERSITY OF LJUBLJANA
FACULTY OF MATHEMATICS AND PHYSICS

Mathematics – 3rd cycle

Philipp Georg Haselwarter

Effective Metatheory for Type Theory

PhD Thesis

Advisor: prof. dr. **Andrej Bauer**

Ljubljana, 2021

UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 3. stopnja

Philipp Georg Haselwarter

Učinkovalna meta-teorija za teorijo tipov

Doktorska disertacija

Mentor: prof. dr. **Andrej Bauer**

Ljubljana, 2021

Abstract

In this dissertation, I propose *finitary type theories* as a definition of a wide class of type theories in the style of Martin-Löf, and I design a programming language for deriving judgements in finitary type theories.

State of the art computer implementations of type theory rely on a computational interpretation of type theory, either via decidability results or via realisability. Such results are not readily available for all type theories studied in the literature, which renders their implementation challenging.

The implementation of a flexible proof assistant supporting user-specified type theories requires a general definition outlining the structure of a type theory. I give a mathematically precise definition of a class of *finitary type theories*, that covers familiar examples, including Extensional Type Theory, the Calculus of Constructions, and Homotopy Type Theory. I first focus on the mathematical development of finitary type theories, before turning to their implementation in proof assistants.

The definition proceeds in stages, starting with raw syntax, raw rules, and raw type theories, then delineating *finitary* rules and type theories, and finally specifying *standard* type theories. Once these definitions are accomplished, general meta-theoretic results in the form of a uniqueness of typing theorem and a cut elimination theorem are proved. I reformulate finitary type theories with a suitable treatment of free variables as *context-free type theories*, paving the way to an implementation in a proof assistant. The definition of context-free type theories again proceeds in stages of refinement, and I prove metatheorems for each successive stage. Translation theorems between context-free and finitary type theories relate the two formalisms.

I introduce the Andromeda metalanguage (AML), an effectful programming language that allows convenient manipulation of judgement and rules of user-definable context-free type theories, and supports common proof development techniques. AML leverages algebraic effects and runners to extend proof assistant algorithms with local hypothesis in a modular way. The operational semantics of AML is inspired by bidirectional typing and helps the user harness contextual information, exhibiting a virtuous interaction with effect operations. AML has been implemented in the Andromeda prover, and I describe first experiments in the computer-assisted development of context-free type theories in AML.

2020 Mathematics Subject Classification: 03B38, 03B70, 18C10, 68N15, 68Q10, 68V15, 03F50, 03F07

Keywords: Dependent type theory, algebraic theory, proof assistant, metalanguage, computational effects

Izvleček

V disertaciji definiram *končne teorije tipov* kot širok razred teorij tipov v stilu Martina-Löfa in oblikujem programski jezik za izpeljevanje sodb v splošnih teorijah tipov.

Sodobne računalniške implementacije teorije tipov se zanašajo na njeno računsko interpretacijo bodisi preko rezultatov o odločljivosti ali realizabilnosti. Takšni rezultati pa niso na voljo za vse teorije tipov, ki jih srečamo v literaturi, zato njihova implementacija predstavlja izziv.

Radi bi implementirali fleksibilen dokazovalni pomočnik, ki omogoča, da uporabnik sam določi teorijo tipov. Za to pa potrebujemo splošno definicijo teorije tipov, ki oriše njeno strukturo. V disertaciji podam matematično natančno definicijo razreda *končnih teorij tipov*, ki pokrije znane primere, vključno z ekstenzionalno teorijo tipov, računom konstrukcij in homotopsko teorijo tipov. Najprej se osredotočim na matematični razvoj končnih teorij tipov, preden se posvetim njihovi implementaciji v dokazovalnih pomočnikih.

Definicija je zgrajena po stopnjah. Začnemo s surovo sintakso, surovimi pravili in surovimi teorijami tipov, nato razmejimo *končna* pravila in teorije tipov ter na koncu opredelimo *standardne* teorije tipov. S temi definicijami lahko dokažemo tudi metateoretične rezultate kot sta izrek o enoličnosti tipov in izrek o eliminaciji rezov. Da bi končne teorije tipov lažje implementirali v dokazovalnem pomočniku, jih preoblikujem v kontekstno neodvisne teorije tipov, ki omogočajo pravilno ravnanje s prostimi spremenljivkami. Definicija kontekstno neodvisnih teorij tipov je ponovno zgrajena po stopnjah. Za vsako stopnjo dokažem primerne metaizreke. Formalizma končnih torij tipov in kontekstno neodvisnih teorij tipov sta povezana preko izrekov prevedbe.

Uvedem metajezik Andromeda (AML), učinkovni programski jezik, ki omogoča priročno ravnanje s sodbami in pravili v kontekstno neodvisnih teorijah tipov, ki jih uporabnik lahko sam določi. Jezik tudi podpira običajne tehnike za razvoj dokazov. AML izkoristi algebrajske učinke in poganjalce, da na modularen način z lokalnimi hipotezami razširi algoritme v dokazovalnih pomočnikih. Operacijska semantika jezika AML se naslanja na dvosmerno tipiziranje in pomaga uporabniku unovčiti informacije o kontekstu. S tem pokaže uspešno interakcijo z operacijami učinkov. AML je implementiran v dokazovalnem pomočniku Andromeda. Opišem tudi prve poskuse razvoja kontekstno neodvisnih teorij tipov z računalniško pomočjo v AML.

2020 Mathematics Subject Classification: 03B38, 03B70, 18C10, 68N15, 68Q10, 68V15, 03F50, 03F05, 03F07
Ključne besede: Odvisna teorija tipov, algebrajska teorija, dokazovalni pomočnik, metajezik, računski učinki

Acknowledgements

I thank my adviser Andrej Bauer for his scientific guidance and for his kindness, which has helped create a welcoming research environment in Ljubljana. His inexhaustible enthusiasm for research, the generosity with which he shares his knowledge and ideas, and his ability to build bridges are a continuous inspiration for me. It is only fitting that the Slovenian word for doctoral adviser is *mentor*.

I thank Robert Harper and Matija Pretnar for accepting to be part of my thesis committee. Their insightful comments have, in some places greatly, improved the quality of this manuscript. My collaboration with Peter LeFanu Lumsdaine has taught me a great deal, not least to face hard technical problems with a smile, and I thank him for that. I count myself lucky to have been a part of the foundations of mathematics and theoretical computer science research group, and I will remember our many joint lunches fondly. Thanks to Alex Simpson for his sense of humour and for his ever insightful mathematical remarks. Niels Voorneveld and our honorary members Riccardo Ugolini and Brett Chenoweth have endured my ramblings about type theory over many a coffee, and I am grateful to have undertaken my doctoral studies along their side. Thanks to Julija Vorotnjak for providing many of the aforementioned coffees, and for always offering a kind word and an open ear. Anja Petković Komel eventually joined me with great enthusiasm in my type theoretic investigations and collaborating with her has been a delight. She deserves further thanks for the innumerable times she helped me navigate the Slovene language. Thanks to Žiga Lukšič for frequently hosting type theoretic research in his office, and for his commitment to teaching. Our group has seen a number of academic visitors over the years, and I would like to thank particularly Théo Winterhalter, Antonin Delpeuch, and Pierre-Marie Pédrot, who made me feel at home linguistically.

This thesis would not have been possible without the many teachers and friends that I met along the way. Thank you to my teachers who gave me the tools and the confidence to pursue research, in particular I want to name Markus Gnad, Diether Thumser, Stef Graillat, Laurent Boudin, Emmanuel Chailloux, Paul-André Melliès, and Matthieu Sozeau. I would not have wound up doing this PhD if I had not been so fortunate to meet Eric Castro, Pierre Chemama, Béatrice Carré, Simon Jacquin, Dahmun Goudarzi, Claire, Jakob Vidmar, Frédéric Bour, Rafaël Proust, and Alan Picol. Thank you all.

Meine Eltern und mein Bruder waren immer für mich da, haben mich stets unterstützt, an mich geglaubt und meine Neugierde geschürt. Ihr seid die beste Familie die ich mir wünschen kann. Danke! Sofia, I cannot thank you enough for your support throughout these tumultuous years, and for your bravery in embarking on this adventure with me. You changed my world, thank you.

Software This manuscript uses the `memoir` class (Madsen and Wilson 2021). Inference rules are typeset with the `mathpartir` package (Rémy 2015). GNU Emacs (FSF 2021b) with AUCTeX (FSF 2021a) was used for document preparation.

Funding This dissertation is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Awards No. FA9550-14-1-0096 and No. FA9550-17-1-0326.

Support for travel via short term sciencific missions to Stockholm and Aarhus through COST Action EUTypes CA15123 is gratefully acknowledged.

A part of the results presented in this thesis was obtained during a visit to the Max Planck Institute for Mathematics (MPIM) in Bonn, during the *Workshop on Homotopy Type Theory* in 2016. This visit was supported by the MPIM. Both this support and the hospitality of MPIM are gratefully acknowledged.

A part of the results presented in this thesis was obtained during a visit to the Hausdorff Research Institute for Mathematics (HIM), University of Bonn, during the *Types, Sets and Constructions* trimester in 2018. This visit was supported by the HIM. Both this support and the hospitality of HIM are gratefully acknowledged.

Contents

Abstract **5**

Izvleček **7**

Contents **11**

List of Figures **14**

1 Introduction **19**
- 1.1 Approaches to Type Theory . . . 19
- 1.2 On the mathematical study of type theory . . . 22
- 1.3 Type theory and proof assistants . . . 23
  - 1.3.1 Computer support for new type theories. . . . 24
  - 1.3.2 Requirements for user definable type theories . . . 24
  - 1.3.3 Effects in proof assistants . . . 26
- 1.4 Aim of the thesis . . . 26
- 1.5 Overview of the thesis . . . 27
  - 1.5.1 Chapter 2: Finitary type theories . . . 27
  - 1.5.2 Chapter 3: Context-free type theories . . . 28
  - 1.5.3 Chapter 4: An effectful metalanguage for type theories . . . 29
  - 1.5.4 Chapter 5: Conclusion . . . 30

2 Finitary type theories **33**
- 2.1 Finitary type theories . . . 34
  - 2.1.1 Raw syntax . . . 34
  - 2.1.2 Deductive systems . . . 42
  - 2.1.3 Raw rules and type theories . . . 43
  - 2.1.4 Finitary rules and type theories . . . 50
- 2.2 Metatheorems . . . 52
  - 2.2.1 Metatheorems about raw theories . . . 52
  - 2.2.2 Metatheorems about finitary theories . . . 70
  - 2.2.3 Metatheorems about standard theories . . . 71

3 Context-free type theories **75**

3.1 Context-free finitary type theories . . . 75
3.1.1 Raw syntax of context-free type theories . . . 76
3.1.2 Context-free rules and type theories . . . 82
3.2 Metatheorems about context-free theories . . . 89
3.2.1 Metatheorems about context-free raw theories . . . 89
3.2.2 Metatheorems about context-free finitary theories . . . 100
3.2.3 Metatheorems about context-free standard theories . . . 100
3.2.4 Special metatheorems about context-free theories . . . 102
3.3 A correspondence between theories with and without contexts . . . 104
3.3.1 Translation from cf-theories to tt-theories . . . 104
3.3.2 Translation from tt-theories to cf-theories . . . 109

4 An effectful metalanguage for type theories 119
4.1 AML preliminaries . . . 120
4.1.1 Bidirectional evaluation . . . 120
4.1.2 Operations and runners . . . 126
4.2 AML syntax . . . 129
4.3 AML operational semantics . . . 133
4.3.1 General programming . . . 134
4.3.2 Type theory . . . 137
4.3.3 Toplevel . . . 141
4.4 Standard derived forms . . . 145
4.4.1 Rule application and formation . . . 147
4.4.2 Handling syntactic equality . . . 150
4.4.3 Recovering λCF-Lambda . . . 150
4.5 On soundness & completeness . . . 151
4.6 AML in Andromeda 2 . . . 152

5 Conclusion 155
5.1 Related work . . . 155
5.1.1 Finitary type theories . . . 155
5.1.2 Andromeda metalanguage . . . 157
5.2 Future work . . . 161

A AML implementation of the boundary conversion lemma 163

B Equational LF in Andromeda 2 165
B.1 Equational LF rules . . . 166
B.2 Equational LF examples . . . 172

C Razširjeni povzetek v slovenščini 177
C.1 Poglavje 1: Uvod . . . 177
C.2 Poglavje 2: Končne teorije tipov . . . 178
C.3 Poglavje 3: Kontekstno neodvisne teorije tipov . . . 181

C.4 Poglavje 4: Učinkovni metajezik za teorije tipov 184
C.5 Poglavje 5: Zaključek 189

D Bibliography 191

List of Figures

2.1 The raw syntax of expressions, boundaries and judgements. 37
2.2 Free, bound, and metavariable occurrences 38
2.3 Filling the head of a boundary 40
2.4 Variable, metavariable and abstraction closure rules 48
2.5 Equality closure rules 48
2.6 Well-formed abstracted boundaries 49
2.7 Well-formed metavariable and variable contexts 49
2.8 Admissible substitution rules 59

3.1 The raw syntax of context-free finitary type theories 77
3.2 Context-free variable occurrences and assumption sets 79
3.3 Abstraction and substitution 80
3.4 Context-free filling the head of a boundary 81
3.5 The action of a metavariable instantiation 82
3.6 Context-free free variable, metavariable, and abstraction closure rules . 87
3.7 Context-free closure rules for equality 88
3.8 Well-formed context-free abstracted boundaries 88

4.1 Declarative and algorithmic rules 121
4.2 Bidirectional typing and elaboration rules, with and without contexts . . 122
4.3 Pseudo-AML rules 124
4.4 Syntax of general AML computations 129
4.5 Syntax of type theoretic AML computations 130
4.6 Syntax of AML values and results 131
4.7 Syntax of AML toplevel commands and patterns 132
4.8 Operational semantics of return and operations 134
4.9 Operational semantics of let binding 135
4.10 Operational semantics of runners 136
4.11 Operational semantics of case matching and function application 137
4.12 Operational semantics of ascription and mode-switch 137
4.13 Operational semantics of TT variables 138
4.14 Operational semantics of metavariables and congruence rules 139
4.15 Operational semantics of TT equality forms 140
4.16 Operational semantics of TT boundaries 141

4.17 Pattern matching 143
4.18 Operational semantics of toplevel commands 144
4.19 Syntax of derived boundary, substitution, and conversion computations . 146
4.20 Induced operational semantics of boundaries and substitution 146
4.21 Syntax and induced operational semantics of derived fresh computation 147
4.22 Syntax of derived abstraction computations 148
4.23 Induced operational semantics of abstraction 149
4.24 Syntax and induced operational semantics of rule application 149
4.25 Syntax and induced operational semantics of derivable rules 149
4.26 Syntax and induced operational semantics of checking lambda 151

C.1 Pravila zaprtja za spremenljivke, metaspremenljivke in abstrakcijo. . . . 179
C.2 Surova sintaksa končnih kontekstno neodvisnih teorij tipov. 182
C.3 Izvleček kontekstno neodvisnih pravil. 182
C.4 Sintaksa splošnih AML izračunov (izvleček). 184
C.5 Sintaksa AML izračunov za teorije tipov (izvleček). 185
C.6 Sintaksa AML vrednosti in rezultatov (izvleček). 185
C.7 Sintaksa AML ukazov na najvišjem nivoju in vzorcev (izvleček). 185
C.8 Operacijska semantika operacij. 186
C.9 Operacijska semantika vezave “let” (izvleček). 186
C.10 Operacijska semantika poganjalcev (izvleček). 186
C.11 Operacijska semantika pripisa meje in menjave načina. 187
C.12 Operacijska semantika spremenljivk teorije tipov. 187
C.13 Operacijska semantika za projekcijo meje v teoriji tipov. 187
C.14 Sintaksa in inducirana operacijska semantika uporabe pravila. 188

Ukiyo-e print illustration showing masses of children playing over, under, and around an enormous elephant. Possibly related to the specimen in (Johnstone 2003). Illustration in “Omacha-e, kodomo-e, harimazechō”, ca. 1875. Source: Wikimedia Commons.

Chapter 1

Introduction

In this thesis we develop a general mathematical theory of dependent type theories and describe their computational treatment in an effectful metatheory. Many examples of dependent type theories such as Martin-Löf type theory (Martin-Löf 1998) or the Calculus of Constructions (Coquand and Huet 1988) have been studied, but there is no generally accepted definition that encompasses a wide range of examples and allows the development of a general metatheory. We propose such a definition in the form of finitary type theories, and prove basic metatheoretical results. We then reformulate finitary type theories in such a way that practical proof development inside such a type theory becomes feasible, arriving at the definition of context free type theories. This allows us to develop a generic metalanguage for standard finitary type theories.

In the remainder of this chapter we will describe what lead us to seek a general definition of type theories and outline the landscape of proof assistants. We will mention the difficulties inherent in the design of such a definition and of a generic proof assistant, and give a brief overview of each chapter of the thesis.

1.1 Approaches to Type Theory

There are three main paths to approach type theory: the logical path, the computational path, and the categorical path. Their history is much too broad a subject to be covered in this introduction, so we will only focus on a few key developments that are of direct relevance to the work presented in this dissertation.

The logical path can be traced back to the foundational crisis at the turn of the twentieth century when Bertrand Russell proposed a *doctrine of types* in Appendix B to his heroic effort to ground mathematics in logic (Russell 1903), and continued to study type theories (Russell 1908) in order to rule out paradoxes that plagued naïve foundational systems. Investigations by the logicians of the 1920s and 1930s, notably Church, lead to the development of the theory of simple types. *Dependent* types first entered the picture with the inception of De Bruijn's Automath in 1967, the first proof assistant that allowed manipulation of derivations as objects in a formal system. From this point onward, the development of type theory and proof assistants has been tightly

interwoven. In the beginning of the 1970s, Martin-Löf proposed his *intuitionistic theory of types* (Martin-Löf 1998) as a foundation for constructive mathematics. Martin-Löf's paradigm was that type theory should serve as the formalism both for *logical reasoning* about and for the *construction* of mathematical objects. In the early 1970s, Milner started working on the LCF proof checker (Milner 1972), starting a line of research of importance to proof assistants at large and to the HOL lineage of provers based on Church's simple type theory in particular. As a side effect of the LCF project, the ML programming language was created (Gordon, Milner, Morris et al. 1978; Milner 1978). Meanwhile, Girard (Girard 1972) and shortly thereafter Reynolds (Reynolds 1974) independently invented System F.

Church connected simple types in 1940 to his lambda calculus (Church 1940). This laid the foundation for the computational interpretation of type theory, as, by then, Church, Kleene, and Turing had shown that the lambda calculus was universal as a model of computation. The rôle of types in connection with computation crystallised with the Curry-Howard correspondence, connecting the computational content of typed lambda calculi to derivations in natural deduction. The slogans "propositions-as-types" and "proofs-as-programs" are associated with this correspondence, and the extension of the correspondence to new computational and logical disciplines has been extraordinarily fruitful, fuelling research in proof theory and programming languages.

Constable and his group at Cornell developed the NuPRL system (Constable et al. 1986) based on *computational type theory* inspired by ideas of Martin-Löf. The starting point of computational type theory is a language with an operational semantics. A type then specifies the behaviour of a program. The fact that a term has a certain type thus requires further evidence in the form of a typing derivation.

In 1984, Coquand and Huet proposed the *calculus of constructions* (Coquand and Huet 1988), a simple yet expressive type theory adding impredicative quantification in the style of Girard and Reynold's System F to Martin-Lövian type theory. As the typing relation of the calculus of constructions is decidable, whether or not a term has a certain type, or equivalently whether a proposed proof constitute adequate evidence for a theorem, can be mechanically checked by a computer. The calculus of constructions forms the base upon which the Coq proof assistant is built.

The inception of the categorical, or algebraic, point of view is commonly attributed to Lambek, who connected the simply typed lambda calculus to cartesian closed categories in the early 1970s. This enabled the mathematical study of the general denotational semantics of type theories, going beyond proof theoretical accounts. Cartmell's thesis on the study of the categorical semantics of Martin-Löf's type theory (Cartmell 1978) as an instance of a wider framework of *generalised algebraic theories* appeared in 1978.

A variant of Martin-Löf's type theory with identity types satisfying the so called *equality reflection rule* became known as *extensional type theory* (Martin-Löf 1979; Martin-Löf 1982), and Seely proposed locally cartesian closed categories as their natural categorical model (Seely 1984). Much of the attention of categorical semantics focused on extensional type theory until the construction of the groupoid model by Hofmann and Streicher in 1993 (Hofmann and Streicher 1994). However, due to

the reflection of provable equality into judgemental equality that is characteristic of extensional type theory, type checking is not decidable in this setting (Hofmann 1997; Castellan et al. 2017). As a result, almost all computer implementations of type theory restrict their attention to intensional type theory, despite the mathematical naturality of extensional type theory. This created a rift between semanticists and practitioners of type theory. Hofmann famously demonstrated that nothing is lost by working in intensional type theory when derivability of judgements is considered (Hofmann 1997). The practical implications of working with intensional type theory are not fully accounted for in this result, as the convenience of conducting proofs with extensional equality types is lost when replaced by intensional identity types and axioms. Around 2006, the gods of type theory decided that the time was ripe for homotopy type theory, and several researchers independently observed the existence of homotopical models of intensional identity types. The subsequent rise of univalent foundations and homotopy type theory (Voevodsky 2014; Univalent Foundations Program 2013), spearheaded by Voevodsky, has lead to a surge in interest in the study of the semantics of intensional type theory. Meanwhile, the implementation of extensional type theory is still mostly uncharted territory.

In a unexpected turn of events, proponents of homotopy type theory rediscovered the appeal of extensional type theory. Simplicial sets are the traditional natural homotopy category of topological spaces and provided the first known model of univalent type theory (Kapulkin and Lumsdaine 2012; Streicher 2011). An obvious question is thus how to define simplicial types inside homotopy type theory. As it turns out, this and the (at least conjecturally) simpler problem of constructing semi-simplicial types, is surprisingly hard (Mike Shulman 2014; Herbelin 2015). It is an open problem whether either of the two constructions is possible in homotopy type theory.

Voevodsky proposed his *homotopy type system* to "*reflect the structures which exist in the target of the canonical univalent model of the Martin-Lof system*" (Voevodsky 2013), where the model in question refers to simplicial sets. In addition to the structure required for the interpretation of the intensional identity type, simplicial sets also form a model of extensional type theory. This observation is internalised in HTS, which "wraps" univalent type theory in an extensional type theory. Variants of HTS such as 2-level type theory have since been used to define semi-simplicial types (Altenkirch et al. 2016), and provide a method for the construction of similar infinite dimensional structures. HTS contains extensional type theory as a subsystem, and type checking is thus undecidable also in HTS.

In this thesis we study a wide class of type theories, including those that do not allow for decidable type checking. For this study, we will adopt Martin-Löf's style of presenting type theory. Specifically, Martin-Löf presents type theory via four kinds of judgements:

Under hypotheses Γ,

- $\Gamma \vdash A$ type asserts that A is a well-formed type,
- $\Gamma \vdash A \equiv B$ asserts that the types A and B are equal,

- $\Gamma \vdash s : A$ asserts that s is a term of type A,
- $\Gamma \vdash s \equiv t : A$ asserts that the terms s and t of type A are equal.

A type theory is then given as a set of rules, whose inductive reading defines a deductive system for the derivation of judgements.

1.2 On the mathematical study of type theory

The development of a mathematical study of type theory allows us to connect type theory with well-understood mathematical concepts. So far, the focus of research in this area has been the development of semantics for specific type theories, such as the interpretation of extensional type theory in locally cartesian closed categories by Seely (1984), Curien (1993), and Hofmann (1994), or more recently the study of homotopical models of intensional type theory (Voevodsky 2006; Awodey and Warren 2007). The study of classes of categorical models for certain type theories, starting with Cartmell (1978), has been extremely fruitful.

The study of general phenomena in type theory requires a general definition of what constitutes a type theory. Broadly speaking, such a definition should cover familiar examples of type theory and allow us to prove general theorems about type theories. These desiderata can be seen as opposing forces: if a definition is too general, we cannot hope to prove or even state certain theorems, if it is too narrow, important type theories are excluded. A satisfactory treatment of type theory in the spirit of categorical logic has to define what a type theory is, define a class of models, and establish a correspondence between the two.

As the focus of this thesis is the connection between type theories and proof assistants, rather than model theory, there are some additional properties we would like a general definition of type theory to satisfy.

1. It is straightforward to take a type theory from a research paper and formulate it according to the general definition. This means that the general definition should include the following notions:
   - (i) (abstract) raw syntax, including a treatment of substitution and metavariables
   - (ii) (hypothetical) judgement forms
   - (iii) the deductive system induced by common structural rules and specific rules of a particular type theory
2. It can be interpreted in any reasonably strong metatheory, and is not committed to one particular ambient foundation.
3. The implementation of a type theory in a proof assistant can easily be seen to form an instance of the general definition.

Together with Andrej Bauer and Peter LeFanu Lumsdaine, we proposed *general dependent type theories* (Bauer, Haselwarter and Lumsdaine 2020) as a mathematical

framework for the study of type theories. This line of work happened concurrently with the preparation of this thesis but will not be included here. In our opinion, general dependent type theories are well suited to address points (1) and (2). They do not, however, directly lend themselves to the implementation of a proof assistant. To reduce the gap between general dependent type theories and an implementation, we will proceed in two steps, introducing first finitary type theories and then context-free type theories, as outlined in Section 1.5.

1.3 Type theory and proof assistants

The goal of proof assistants, also known as interactive theorem provers, is to allow the computer verification of formal proofs. At the core of a theorem prover is a logical formalism, as, for example, first order logic with set theory, higher order logic, or a version of type theory. Type theory in the tradition of Martin-Löf is a natural choice as logical foundation, because it is built around the notion of dependency, which is pervasive in mathematics. Type theory further allows mathematical objects to be represented directly as types, in contrast, for example, to set theories, which rely on the encoding of mathematical structures in terms of simpler sets. The direct presentation of mathematics in type theory is philosophically appealing from a structuralists point of view (Awodey 2014). Moreover, it has the very concrete advantage that both the computer implementation and the users of the proof assistant can work directly with the primitives of the theory. This allows efficient representations of and computation with mathematical structures, such as finite groups (Gonthier, Asperti et al. 2013).

There are numerous successful proof assistants based on type theory. In particular, NuPRL (Constable et al. 1986) belongs to the school of provers based on realisability, or *computational type theory*. Coq (The Coq development team 2021a), Agda (Norell 2007), and Lean (de Moura et al. 2015) instead rely on the decidability of type checking for the type theories they respectively implement. The realisability approach can handle expressive type theories that include strong logical principles, such as equality reflection. The commitment to one particular realisability interpretation that is inherent in computational type theory, however, means that a proof in NuPRL can only be interpreted in a class of realisability models. The essential requirement for decidable type checking is the decidability of judgemental equality. Systems based on decidable type theories afford the user the convenience that any proofs of judgemental equality can be relegated to the proof assistant. Decidable judgemental equality can be used as powerful computation mechanism through small scale reflection (Gonthier, Mahboubi et al. 2015), but unless it is used expertly, computation can quickly become infeasible. A more severe limitation of this approach is that type checking is simply undecidable for many type theories of interest.

All of the aforementioned proof assistants are tailored closely to the particular type theory they implement. This is in fact a requirement of their respective approaches. Extending a realisability model requires careful work, and crafting algorithms for judgemental equality is a subtle business.

1.3.1 Computer support for new type theories.

As there is great interest in the study and development of new type theories, there is a growing need for extensible proof assistants. Already early type theories proposed by Martin-Löf were intended to be open ended and compatible with the introduction of new types. Most mature proof assistants are indeed open ended in the sense that they allow the definition of new inductive types according to fixed schemata. Deeper changes to their core theory are harder to come by. For example, in Coq one can change the universe system to include the rule $\mathsf{Type} : \mathsf{Type}$, but the addition of this feature required modification of the implementation of Coq. Swapping out one type theory for a different, albeit similar one is infeasible. For instance, the UniMath (Voevodsky et al. n.d.) project developed in Coq aims to work within a minimal fragment of Coq's theory, but it is not possible to properly enforce this restriction.

One approach to work with a different type theories in a proof assistant is provided by syntactic models (Hofmann 1997; S. P. Boulier 2018; S. Boulier et al. 2017; Pédrot and Tabareau 2017). This is a powerful methodology to extend type theories with new logical and computational principles via shallow embeddings. They preserve desirable properties of the host type theory such as decidability of type checking. However, this also means that their construction is challenging. Furthermore, there is no mathematical theory characterising which type theories arise in this way.

Limited extensions of judgemental equality have been considered. Coq Modulo Theory (Barras et al. 2011) is a proposal to extend the judgemental equality of Coq in a safe way that preserves soundness and decidability of type checking. While CoqMT allows to extend the equality checker with an arbitrary decidable first order theory, it can of course not account for full equality reflection. A recent proposal by Abel and Cockx considers the addition of rewrite rules to Agda (Cockx and Abel 2016), but, as the user cannot control when these rules are applied, only a very specific class of well-behaved rewriting rules can be used.

Finally, the development of a proof assistant requires both a high level of domain expertise and many programming hours. It is thus not sustainable to develop a dedicated proof assistant for each novel type theory proposed. Instead, we provide a reusable framework in the hope that it can be of service to other researchers wishing to experiment.

1.3.2 Requirements for user definable type theories

Considering the speed at which new type theories are proposed in research, the need for a proof assistant with flexible support for user definable type theories is increasingly evident. Such a flexible proof assistant would allow the user to decide on the type theory they want to work in and provide good support for proof development techniques commonly found in contemporary theorem provers.

In order to allow user definable type theories in a proof assistant, we first have to delineate precisely a class of type theories that an implementation of a customisable proof assistant should accept. In other words, we need a formal definition of type

theories, for which we can then provide a flexible proof assistant. If users can postulate arbitrary rules for judgemental equality, we cannot expect the resulting theory to have decidable type checking. The proof assistant must thus provide control over equality checking to the user.

In light of these requirements, a flexible proof assistant cannot be based on the computation-based architectures used in systems such as NuPRL, Coq, or Agda. An alternative approach is provided by the LCF/HOL architecture, pioneered by Robin Milner (Gordon, Milner and Wadsworth 1979). In an LCF/HOL prover, the logic is implemented in a minimalistic kernel, while proof automation and computation are delegated to a metalanguage (Gordon 2000). The interface to the kernel is abstract, and the type safety of the metalanguage guarantees that arbitrary computation can be used without risking the soundness of the system. Instead of recording entire proof trees, only the concluding judgement is stored.

The LCF/HOL architecture is appropriate for handling unspecified type theories because it does not require them to be equipped with good computational properties. Moreover, the commitment to a computational metalanguage allows the user to program the system to develop domain-specific judgemental equality algorithms for their type theory of choosing. For these reasons, in this thesis, we will develop a flexible proof assistant adopting the LCF/HOL approach.

Handling judgemental equality is not the only obstacle to the implementation of user definable type theories. As previously mentioned, extensional type theory is one of the theories we want to be able to represent in our system. It serves as a formidable source of counterexamples to the intuitions of implementers of intensional type theories. Besides the aforementioned undecidability of equality, extensional type theory also fails to satisfy strengthening (Harper, Honsell et al. 1993), as the equality reflection rule induces a strong form of proof irrelevance whereby an inhabitant of the equality type, potentially containing variables, can be eliminated without record in the resulting judgement. It is worth pointing out that while extensional type theory is an important example that we want to cover, it is not the only setting in which such issues of proof irrelevance arise (Awodey and Bauer 2004), and that conversely the techniques we use to solve the aforementioned issues still accommodate type theories that do not display such "pathologies".

A consequence of the failure of strengthening is that every judgement we derive has to contain sufficient information to deduce in which context it is valid. Explicit representations of contexts as lists, as are commonly found in pen-and-paper presentations of type theory, are ill-suited for implementation for reasons of efficiency, and because the linear order of hypothesis is overly rigid when deriving judgements through forward reasoning. For instance, the judgements $x{:}A, y{:}B \vdash s : C$ and $y{:}B, x{:}A \vdash t : D$ cannot readily be combined as their contexts are different. Unstructured representations such as a map of names to types is ill-suited because the proof-irrelevance incurred by equality reflection also invalidates exchange (Bauer, Gilbert et al. 2018). Therefore, a record of the dependencies between variables has to be kept. In the experimental implementation of extensional type theory in Andromeda 1 (Bauer, Gilbert et al. 2018), contexts were represented as directed acyclic graphs, similarly to the way certain proof

assistants implement a flexible universe management system (Huet 1988). This may constitute a feasible implementation strategy but we found it unsatisfactory from a type theoretic point of view.

We adopt an alternative solution by annotating variables with their types, and removing contexts all together. The dependency graph is then implicit in the recursive annotations of variables. To deal with examples such as the equality reflection rule, the structure of equality judgements has to be changed, such that each equation records the assumptions that were used in its derivation. This idea will form the basis of context-free type theories (Chapter 3).

1.3.3 Effects in proof assistants

Proof engines for type theories are naturally effectful: the current theory signature is a monotone state, unification can change global state, control effects are employed in the form of backtracking, etc. MTac (Ziliani, Dreyer et al. 2013) and Lean represent these effects as monads. In Coq, the tactic engine (Spiwack 2010) and Ltac2 (Pédrot 2019) select a few "blessed" effects that are primitive to the language and can be used in direct style. This allows for a natural use of effectful computations as native part of the language, such as references in ML, rather than as derived concept, such as monads in e.g. Haskell, that force the user to manually sequence effectful computations (Danvy 1992). Dually, the direct-style approach can be viewed as working in an ambient tactic- or proof-monad (Kirchner and Muñoz 2010). For instance, the effects available in the proof monad of Ltac2 are input-output, setting of mutable variables, failure, backtracking, and accessing the current proof state, i.e. the current goals and hypotheses. Manipulating the behaviour of effectful programs, for example steering the unification engine, requires changes to the implementation of the underlying algorithms. In recent years, there has been an emphasis on allowing the user to customise the behaviour of proof assistants, for example via unification hints (Asperti et al. 2009) and canonical structures (Mahboubi and Tassi 2013; Ziliani and Sozeau 2015).

We thus embrace effects as a reality in proof development rather than trying to confine them to a monad. Proof development for general type theories is unchartered territory, and we cannot foresee which techniques and effects will be the most useful.

We introduce effects into our language via algebraic effect operations and runners (Ahman and Bauer 2019). Algebraic effects allow users to define their own effects (Bauer and Pretnar 2015), and runners allow to locally modify the behaviour of effect operations in a principled way. This allows the implementation of well-known techniques while leaving space for experimentation, and opens the possibility for users to customise the behaviour of tactics.

1.4 Aim of the thesis

The aim of this thesis is to develop an effective metatheory for type theory. This is achieved by the proposing *finitary type theories* as a definition of a wide class of type

theories in the style of Martin-Löf, reformulating them as *context-free type theories* in a style suitable for implementation, and designing an effectful programming language for deriving judgements in context-free type theories.

1.5 Overview of the thesis

We give a brief overview of the contents of each chapter, and highlight novel contributions.

1.5.1 Chapter 2: Finitary type theories

Chapter 2 proposes *finitary type theories* as an elementary definition of a wide class of type theories in the style of Martin-Löf. A type theory should verify certain meta-theoretical properties: the constituent parts of any derivable judgement should be well-formed, substitution rules should be admissible, and each term should have a unique type.

The definition of finitary type theories proceeds in stages. Each of the stages refines the notion of *rule* and *type theory* by specifying conditions of well-formedness. We start with the raw syntax (§2.1.1) of expressions and formal metavariables, out of which contexts, substitutions, and judgements are formed. Next we introduce *raw rules* (§2.1.3), a formal notion of what is commonly called "schematic inference rule". We introduce the *structural rules* (Figs. 2.4 to 2.6) that are shared by all type theories, and define *congruence rules* (Def. 2.1.13). These rules are then collected into raw type theories (Def. 2.1.16). The definition of raw rules ensures the well-typedness of each constituent part of a raw rule, by requiring the derivability of the presuppositions of a rule. In order to rule out circularities in the derivations of well-typedness, and to provide an induction principle for finitary type theories, we introduce *finitary rules* and *type theories* (§2.1.4). Finally, *standard* type theories are introduced (Def. 2.1.20) to enforce that each symbol is associated to a unique rule. We prove metatheorems about raw (§2.2.1), finitary (§2.2.2), and standard type theories (§2.2.3).

Contributions. A mathematically precise definition of type theories is proposed. All constructions carried out in this chapter are constructive, and we aimed to use only elementary notions that should be interpretable in a range of different foundational formalisms. To summarise, we

- define a notion of arity and signature suitable for the binding structures commonly found in type theory,
- define a general notion of raw syntax,
- give a formal treatment of metavariables,
- introduce a useful decomposition of judgements into *heads* and *boundaries*,
- define rules in a matching common type theoretic practise,
- explicate properties that make type theories *finitary* and *standard*,

- prove the following metatheorems:
  - admissibility of substitution and equality substitution (Theorem 2.2.8),
  - admissibility of instantiation of metavariables (Theorem 2.2.13) and equality instantiation (Theorem 2.2.17),
  - derivability of presuppositions (Theorem 2.2.18),
  - admissibility of "economic" rules (Propositions 2.2.19 and 2.2.20)
  - inversion principles (Theorem 2.2.22),
  - uniqueness of typing (Theorem 2.2.24).

1.5.2 Chapter 3: Context-free type theories

The goal of this chapter is the development of a context-free presentation of finitary type theories that can serve as foundation of the implementation of a proof assistant. The definition of finitary type theories in Chapter 2 is well-suited for the metatheoretic study of type theory, but does not address the implementation issues discussed in Section 1.3.2. In particular, in keeping with traditional accounts of type theory, contexts are explicitly represented as lists.

In *context-free type theories*, the syntax of expressions (§3.1.1) is modified so that each free variable is annotated with its type a^A rather than being assigned a type by a context. As the variables occurring in the type annotation A are also annotated, the dependency between variables is recorded. Judgements in context-free type theories thus do not carry an explicit context. Metavariables are treated analogously. To account for the possibility of proof-irrelevant rules like equality reflection, where not all of the variables used to derive the premises are recorded in the conclusion, we augment type and term equality judgements with *assumption sets* (§3.1.1.5). Intuitively, in a judgement $\vdash A \equiv B$ by α, the assumption set α contains the (annotated) variables that were used in the derivation of the equation but may not be amongst the free variables of A and B. The conversion rule of type theory allows the use of a judgemental equality to construct a term judgement. To ensure that assumption sets on equations are not lost as a result of conversion, we include *conversion terms* (Fig. 3.1).

Following the development of finitary type theories, we introduce raw context-free rules and type theories (§3.1.2). We proceed to define *context-free finitary rules* and type theories whose well-formedness is derivable with respect to a well-founded order (Def. 3.1.13), and *standard* theories (Def. 3.1.14).

Subsequently, we prove metatheorems about context-free raw (§3.2.1), finitary (§3.2.2), and standard type theories (§3.2.3). In Section 3.2.4, we prove that raw context-free type theories satisfy strengthening (Theorem 3.2.16). The constructions underlying these metatheorems are defined on judgements rather than derivations, and can thus be implemented *effectively* in a proof assistant for context-free type theories without storing derivation trees. Finally, we establish a correspondence between type theories with and without contexts by constructing translations back and forth (§3.3).

Contributions. A context-free definition of type theory is proposed. We show that context-free type theories satisfy useful effective metatheorems. We provide a precise connection to finitary type theories. To summarise, we

- define a syntax with annotated variables and metavariables,
- define context-free judgements with assumption sets,
- define rules appropriate structural rules (Figs. 3.6 to 3.8),
- explicate properties that make context-free type theories *finitary* and *standard*,
- prove the following metatheorems:
  - admissibility of substitution (Theorems 3.2.4 and 3.2.7),
  - derivability of presuppositions (Theorem 3.2.5),
  - admissibility of instantiation of metavariables (Proposition 3.2.8),
  - admissibility of "economic" rules (Propositions 3.2.9 and 3.2.10)
  - fine-grained inversion principles (Theorem 3.2.14),
  - uniqueness of typing (Theorem 3.2.15),
  - admissibility of strengthening (Theorem 3.2.16),
- we give a translation of finitary context-free type theories to finitary type theories with context (Theorem 3.3.5),
- we give a translation of standard type theories with context to standard context-free type theories (Theorem 3.3.10).

1.5.3 Chapter 4: An effectful metalanguage for type theories

In Chapter 4, we present the Andromeda metalanguage (AML), an effectful programming language that allows convenient manipulation of judgement and rules of context-free type theories, and supports common proof development techniques.

The definition of context-free type theories is well suited as the kernel of a proof assistant. AML combines type theoretic primitives for the construction of judgements, boundaries, and rules, with general purpose programming constructs in the form of functions, algebraic effect operations, and runners. The operational semantics of AML is inspired by bidirectional typing. We explain how to refine a declarative definition of type theory into a bidirectional one (§4.1.1). We then introduce *bidirectional evaluation*, which we will generalise to context-free type theories. The effectful character of AML arises from its use of *operations* and *runners*, which we explain in Section 4.1.2. We define the formal syntax of AML computations, values, and toplevel commands (§4.2). AML is equipped with an operational semantics (§4.3) which combines bidirectional evaluation with operations and runners. The use of effects is a central aspect of AML, and operations display a virtuous interaction with bidirectional evaluation (§4.3.1). In Section 4.3.2, we describe the dynamic behaviour of type theoretic primitives, as well as the computational interpretation of our effective of metatheorems for context-free type theories, and the mechanism that allows users to define their own type theories in AML (§4.3.3).

To showcase the expressiveness of AML and provide a more user friendly surface language, we define a number of *derived forms* (§4.4). We present an AML program

implementing Lemma 3.2.17, which allows the user to work transparently with context-free type theories, ignoring conversion terms (§4.4.2). The corresponding code can be found in Appendix A. We outline the formal relation between AML and context-free type theory (§4.5), and briefly present the implementation of AML in the Andromeda 2 prover (§4.6), and give a definition of Harper's *Equational Logical framework* (Harper 2021) in Andromeda 2 (§B).

Contributions. AML is an effectful programming language for context-free type theories, supporting proof development in user-specified type theories in a convenient high level language. To summarise, we

- embed context-free type theories in a ML-style language,
- extend bidirectional typing to bidirectional evaluation,
- demonstrate how to employ runners for proof development with local hypotheses,
- provide a mechanism for user definable rules and derived rules,
- implement effective versions of metatheorems of standard context-free type theories,
- implement AML in the Andromeda 2 prover.

1.5.4 Chapter 5: Conclusion

We give an overview over related work and outline directions for future work in Chapter 5. In Section 5.1.1 we discuss the relation of finitary type theories to other recently proposed general definitions of type theories. We compare the AML approach to existing effectful metalanguages, user-extensible proof assistants, and previous work of our own (§5.1.2). In Section 5.2, we suggest the next steps in the metatheoretic study of finitary and context-free type theories and sketch interesting extensions. Finally, we propose theoretical and practical questions as well as possible extensions for AML.

Ceci n’est pas un coq.
Source: Biodiversity Heritage Library.

Chapter 2

Finitary type theories

We present a general definition of a class of dependent type theories which we call *finitary type theories*. Our definition broadly follows the development of general type theories (Bauer, Haselwarter and Lumsdaine 2020), but is specialized to serve as a formalism for implementation of a proof assistant. Nevertheless, we expect the notion of finitary type theories to be applicable in other situations and without reference to any particular implementation.

Our definition captures dependent type theories of Martin-Löf style, i.e. theories that strictly separate terms and types, have four judgement forms (for terms, types, type equations, and typed term equations), and hypothetical judgements standing in intuitionistic contexts. Among examples are the 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. Counter-examples can be found just as easily: in cubical type theory the interval type is special, cohesive and linear type theories have non-intuitionistic contexts, polymorphic λ-calculi quantify over all types, pure type systems organize the judgement forms in their own way, and so on.

The rest of this chapter proceeds as follows. In Section 2.1 we give an account of dependent type theories that is close to how they are traditionally presented. First, the raw abstract syntax of expressions and judgements is presented (Section 2.1.1) and used to define *raw type theories* (Section 2.1.3). These suffice to define the notions of derivation and deductive system, but may be quite unwieldy from a type-theoretic viewpoint. We thus provide more restrictive but well-behaved notions of *finitary* and *standard type theories* (Section 2.1.4).

In Section 2.2 we show that the definitions are sensible and desirable by establishing good structural properties of type theories, such as admissibility of substitutions (Theorem 2.2.8) and instantiations (Theorems 2.2.13 and 2.2.17), presuppositivity (Theorem 2.2.18), an inversion principle (Theorem 2.2.22), and uniqueness of typing (Theorem 2.2.24).

2.1 Finitary type theories

Our treatment of type theories follows in essence the definition of general type theories carried out in (Bauer, Haselwarter and Lumsdaine 2020), but is tailored to support algorithmic derivation checking in three respects: we limit ourselves to finitary symbols and rules, construe metavariables as a separate syntactic class rather than extensions of signatures by fresh symbols, and take binding of variables to be a primitive operation on its own.

2.1.1 Raw syntax

In this section we describe the *raw* syntax of fintary type theories, also known as pre-syntax. We operate at the level of *abstract* syntax, i.e. we construe syntactic entities as syntax trees generated by grammatical rules in inductive fashion. Of course, we still display such trees *concretely* as string of symbols, a custom that should not detract from the abstract view.

Raw expressions are formed without any typing discipline, but they have to be syntactically well-formed in the sense that free and bound variables must be well-scoped and that all symbols must be applied in accordance with the given signature. We shall explain the details of these conditions after a short word on notation.

We write $[X_1, \ldots, X_n]$ for a finite sequence and $f = \langle X_1 {\mapsto} Y_1, \ldots, X_n {\mapsto} Y_n \rangle$ for a sequence of pairs (X_i, Y_i) that represents a map taking each X_i to Y_i. An alternative notation is $\langle X_1{:}Y_1, \ldots, X_n{:}Y_n \rangle$, and we may elide the parenthess $[\cdots]$ and $\langle \cdots \rangle$. The ***domain*** of such f is the set $|f| = \{X_1, \ldots, X_n\}$, and it is understood that all X_i are different from one another. Given $X \notin |f|$, the ***extension*** $\langle f, X {\mapsto} Y \rangle$ of f by $X \mapsto Y$ is the map

$$\langle f, X{\mapsto}Y \rangle : Z \mapsto \begin{cases} Y & \text{if } Z = X, \\ f(Z) & \text{if } Z \in |f|. \end{cases}$$

Given a list $\ell = [\ell_1, \ldots, \ell_n]$, we write $\ell_{(i)} = [\ell_1, \ldots, \ell_{i-1}]$ for its i-th initial segment. We use the same notation in other situations, for example $f_{(i)} = \langle X_1 \mapsto Y_1, \ldots, X_{i-1} \mapsto Y_{i-1} \rangle$ for f as above.

2.1.1.1 Variables and substitution

We distinguish notationally between the disjoint sets of *free variables* $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$ and *bound variables* $x, y, z, \ldots$, each of which are presumed to be available in unlimited supply. The free variables are scoped by variable contexts, while the bound ones are always captured by abstractions.

The strict separation of free and bound variables is fashioned after *locally nameless syntax* (McKinna and Pollack 1993; Charguéraud 2012), a common implementation technique of variable binding in which free variables are represented as names and the bound ones as de Bruijn indices (de Bruijn 1972). In Section 3.1 the separation between free and bound variables will be even more pronounced, as only the former ones are annotated with types.

We write $e[s/x]$ for the substitution of an expression s for a bound variable x in expression e and $e[\vec{s}/\vec{x}]$ for the (parallel) substitution of $s_1, \dots, s_n$ for $x_1, \dots, x_n$, with the usual proviso about avoiding the capture of bound variables. In Section 2.2.1, when we prove admissibility of substitution, we shall also substitute expressions for free variables, which of course is written as $e[s/\mathsf{a}]$. Elsewhere we avoid such substitutions and only ever replace free variables by bound ones, in which case we write $e[x/\mathsf{a}]$. This typically happens when an expression with a free variable is used as part of a binder, such as the codomain of a Π-type or the body of a lambda. We take care to always keep bound variables well-scoped under binders.

2.1.1.2 Arities and signatures

The raw expressions of a finitary type theory are formed using ***symbols*** and ***metavariables***, which constitute two separate syntactic classes. Each symbol and metavariable has an associated arity, as follows.

The ***symbol arity*** $(c, [(c_1, n_1), \dots, (c_k, n_k)])$ of a symbol S tells us that

1. the syntactic class of S is $c \in \{\mathsf{Ty}, \mathsf{Tm}\}$,
2. S accepts k arguments,
3. the i-th argument must have syntactic class $c_i \in \{\mathsf{Ty}, \mathsf{Tm}, \mathsf{EqTy}, \mathsf{EqTm}\}$ and binds n_i variables.

The syntactic classes Ty and Tm stand for type and term expressions, and EqTy and EqTm for type and term equations, respectively. For the time being the latter two are mere formalities, as the only expression of these syntactic classes are the dummy values $\star_{\mathsf{Ty}}$ and $\star_{\mathsf{Tm}}$. However, in Section 3.1 we will introduce genuine expressions of syntactic classes EqTy and EqTm.

The information about symbol arities is collected in a ***signature*** Σ, which maps each symbol to its arity. When discussing syntax, it is understood that such a signature has been given, even if we do not mention it explicitly.

Example 2.1.1. 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 ***metavariable arity*** associated to a metavariable M is a pair (c, n), where the syntactic class $c \in \{\mathsf{Ty}, \mathsf{Tm}, \mathsf{EqTy}, \mathsf{EqTm}\}$ indicates whether M is respectively a type, term, type equality, or term equality metavariable, and n is the number of term arguments it accepts. The metavariables of syntactic classes Ty and Tm are the ***object*** metavariables, and can be used to form expressions. The metavariable of syntactic classes EqTy and EqTm are the ***equality*** metavariables, and do not participate in formation of expressions. We introduce them to streamline several definitions, and to

have a way of referring to equational premises in Section 3.1. The information about metavariable arities is collected in a metavariable context, cf. Section 2.1.1.4.

A metavariable M of arity (c, n) could be construed as a symbol of arity

$$(c, \underbrace{[(\mathsf{Tm}, 0), \ldots, (\mathsf{Tm}, 0)]}_{n}).$$

This apporach is taken in (Bauer, Haselwarter and Lumsdaine 2020), but we keep metavariables and symbols separate because they play different roles, especially in context-free type theories in Section 3.1.

2.1.1.3 Raw expressions

The raw syntactic constituents of a finitary type theory, with respect to a given signature Σ, are outlined in Fig. 2.1. In this section we discuss the top part of the figure, which involves the syntax of term and type expressions, and arguments.

A ***type expression***, or just a ***type***, is formed by an application $\mathsf{S}(e_1, \ldots, e_n)$ of a type symbol to arguments, or an application $\mathsf{M}(t_1, \ldots, t_n)$ of a type metavariable to term expressions. A ***term expression***, or just a ***term***, is a free variable a, a bound variable x, an application $\mathsf{S}(e_1, \ldots, e_n)$ of a term symbol to arguments, or an application $\mathsf{M}(t_1, \ldots, t_n)$ of a term metavariable to term expressions.

An ***argument*** is a type or a term expression, the dummy argument $\star_{\mathsf{Ty}}$ of syntactic class EqTy, or the dummy argument $\star_{\mathsf{Tm}}$ of syntactic class EqTm. We write just $\star$ when it is clear which of the two should be used. Another kind of argument is an ***abstraction*** $\{x\}e$, which binds x in e. An iterated abstraction $\{x_1\}\{x_2\} \cdots \{x_n\}e$ is abbreviated as $\{\vec{x}\}e$. Note that abstraction is a primitive syntactic operation, and that it provides no typing information about x.

Example 2.1.2. In our notation a dependent product is written as $\Pi(A, \{x\}B)$, and a fully annotated function as $\lambda(A, \{x\}B, \{x\}e)$. The fact that x ranges over A is not part of the raw syntax and will be specified later by an inference rule.

In all cases, in order for an expression to be well-formed, the arities of symbols and metavariables must be respected. If S has arity $(c, [(c_1, n_1), \ldots, (c_k, n_k)])$, then it must be applied to k arguments $e_1, \ldots, e_k$, where each e_i is of the form $\{x_1\} \cdots \{x_{n_i}\}e'_i$ with e'_i a non-abstracted argument of syntactic class c_i. Similarly, a metavariable M of arity (n, c) must be applied to n term expressions. When a symbol S takes no arguments, we write the corresponding expression as S rather than $\mathsf{S}()$, and similarly for metavariables.

As is usual, expressions which differ only in the choice of names of bound variables are considered syntactically equal, e.g., $\{x\}\mathsf{S}(\mathsf{a}, x)$ and $\{y\}\mathsf{S}(\mathsf{a}, y)$ are syntactically equal and we may write $(\{x\}\mathsf{S}(\mathsf{a}, x)) = (\{y\}\mathsf{S}(\mathsf{a}, y))$.

For future reference we define in Fig. 2.2 the sets of free variable, bound variable, and metavariable occurrences, where we write set comprehension as $\{\!\!\{ \cdots \}\!\!\}$ in order to distinguish it from abstraction. A syntactic entity is said to be ***closed*** if no free variables occur in it.

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 argument
	$\mid \star_{\mathsf{Tm}}$	dummy argument
	$\mid \{x\}e$	abstracted argument (x bound in e)
Judgement thesis j	$::= A \ \mathsf{type}$	A is a type
	$\mid t : A$	t has type T
	$\mid A \equiv B \ \mathsf{by} \ \star_{\mathsf{Ty}}$	A and B are equal types
	$\mid s \equiv t : A \ \mathsf{by} \ \star_{\mathsf{Tm}}$	s and t are equal terms at A
Abstracted judgement $\mathcal{J}$	$::= j$	judgement thesis
	$\mid \{x{:}A\} \ \mathcal{J}$	abstracted judgement (x bound in $\mathcal{J}$)
Boundary thesis $\mathscr{b}$	$::= \Box \ \mathsf{type}$	a type
	$\mid \Box : A$	a term of type A
	$\mid A \equiv B \ \mathsf{by} \ \Box$	type equation boundary
	$\mid s \equiv t : B \ \mathsf{by} \ \Box$	term equation boundary
Abstracted boundary $\mathcal{B}$	$::= \mathscr{b}$	boundary thesis
	$\mid \{x{:}A\} \ \mathcal{B}$	abstracted boundary (x bound in $\mathcal{B}$)
Variable context Γ	$::= [\mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n]$	
Metavariable context 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 2.1: The raw syntax of expressions, boundaries and judgements.

Free variables:

$$\mathsf{fv}(\mathsf{a}) = \{\!|\mathsf{a}|\!\} \qquad \mathsf{fv}(x) = \{\!||\!\} \qquad \mathsf{fv}(\{x\}e) = \mathsf{fv}(e)$$

$$\mathsf{fv}(\mathsf{S}(e_1 \ldots e_n)) = \mathsf{fv}(e_1) \cup \cdots \cup \mathsf{fv}(e_n)$$

$$\mathsf{fv}(\mathsf{M}(t_1 \ldots t_n)) = \mathsf{fv}(t_1) \cup \cdots \cup \mathsf{fv}(t_n)$$

$$\mathsf{fv}(A\ \mathsf{type}) = \mathsf{fv}(A) \qquad \mathsf{fv}(t : A) = \mathsf{fv}(t) \cup \mathsf{fv}(A)$$

$$\mathsf{fv}(A \equiv B\ \mathsf{by}\ \star) = \mathsf{fv}(A) \cup \mathsf{fv}(B)$$

$$\mathsf{fv}(s \equiv t : A\ \mathsf{by}\ \star) = \mathsf{fv}(s) \cup \mathsf{fv}(t) \cup \mathsf{fv}(A)$$

$$\mathsf{fv}(\{x : A\}\ \mathcal{J}) = \mathsf{fv}(A) \cup \mathsf{fv}(\mathcal{J})$$

$$\mathsf{fv}(\Box\ \mathsf{type}) = \{\!||\!\} \qquad \mathsf{fv}(\Box : A) = \mathsf{fv}(A)$$

$$\mathsf{fv}(A \equiv B\ \mathsf{by}\ \Box) = \mathsf{fv}(A) \cup \mathsf{fv}(B)$$

$$\mathsf{fv}(s \equiv t : A\ \mathsf{by}\ \Box) = \mathsf{fv}(s) \cup \mathsf{fv}(t) \cup \mathsf{fv}(A)$$

$$\mathsf{fv}(\{x : A\}\ \mathcal{B}) = \mathsf{fv}(A) \cup \mathsf{fv}(\mathcal{B})$$

Bound variables:

$$\mathsf{bv}(\mathsf{a}) = \{\!||\!\} \qquad \mathsf{bv}(x) = \{\!|x|\!\} \qquad \mathsf{bv}(\{x\}e) = \mathsf{bv}(e) \setminus \{\!|x|\!\}$$

$$\mathsf{bv}(\mathsf{S}(e_1 \ldots e_n)) = \mathsf{bv}(e_1) \cup \cdots \cup \mathsf{bv}(e_n)$$

$$\mathsf{bv}(\mathsf{M}(t_1 \ldots t_n)) = \mathsf{bv}(t_1) \cup \cdots \cup \mathsf{bv}(t_n)$$

Metavariables:

$$\mathsf{mv}(\mathsf{a}) = \{\!||\!\} \qquad \mathsf{mv}(x) = \{\!||\!\} \qquad \mathsf{mv}(\{x\}e) = \mathsf{mv}(e)$$

$$\mathsf{mv}(\mathsf{S}(e_1 \ldots e_n)) = \mathsf{mv}(e_1) \cup \cdots \cup \mathsf{mv}(e_n)$$

$$\mathsf{mv}(\mathsf{M}(t_1 \ldots t_n)) = \{\!|\mathsf{M}|\!\} \cup \mathsf{mv}(t_1) \cup \cdots \cup \mathsf{mv}(t_n)$$

$$\mathsf{mv}(A\ \mathsf{type}) = \mathsf{mv}(A) \qquad \mathsf{mv}(t : A) = \mathsf{mv}(t) \cup \mathsf{mv}(A)$$

$$\mathsf{mv}(A \equiv B\ \mathsf{by}\ \star) = \mathsf{mv}(A) \cup \mathsf{mv}(B)$$

$$\mathsf{mv}(s \equiv t : A\ \mathsf{by}\ \star) = \mathsf{mv}(s) \cup \mathsf{mv}(t) \cup \mathsf{mv}(A)$$

$$\mathsf{mv}(\{x{:}A\}\ \mathcal{J}) = \mathsf{mv}(A) \cup \mathsf{mv}(\mathcal{J})$$

$$\mathsf{mv}(\Box\ \mathsf{type}) = \{\!||\!\} \qquad \mathsf{mv}(\Box : A) = \mathsf{mv}(A)$$

$$\mathsf{mv}(A \equiv B\ \mathsf{by}\ \Box) = \mathsf{mv}(A) \cup \mathsf{mv}(B)$$

$$\mathsf{mv}(s \equiv t : A\ \mathsf{by}\ \Box) = \mathsf{mv}(s) \cup \mathsf{mv}(t) \cup \mathsf{mv}(A)$$

$$\mathsf{mv}(\{x : A\}\ \mathcal{B}) = \mathsf{mv}(A) \cup \mathsf{mv}(\mathcal{B})$$

Figure 2.2: Free, bound, and metavariable occurrences

2.1.1.4 Judgements and boundaries

The bottom part of Fig. 2.1 displays the syntax of judgements and boundaries, which we discuss next.

There are four ***judgement forms***: "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. We may shorten the equational forms to "$A \equiv B$" and "$s \equiv t : A$" in this chapter, as the only possible choice for by is $\star$.

Less familiar, but equally fundamental, is the notion of a ***boundary***. Whereas a judgement is an assertion, a boundary is a *question* to be answered, a *promise* to be fulfilled, or a *goal* to be accomplished: "$\Box$ type" asks that a type be constructed; "$\Box : A$" that the type A be inhabited; and "$A \equiv B$ by $\Box$" and "$s \equiv t : A$ by $\Box$" 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. Thus in general an abstracted judgement has the form

$$\{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\ j,$$

where j is a ***judgement thesis***, i.e. an expression taking one of the four (non-abstracted) judgement forms. We may abbreviate such an abstraction as $\{\vec{x}{:}\vec{A}\}\ j$. Analogously, an ***abstracted boundary*** has the form

$$\{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\ \mathscr{b},$$

where $\mathscr{b}$ is a ***boundary thesis***, i.e. it takes one of the four (non-abstracted) boundary forms. The reason for introducing abstracted judgements and boundaries will be explained shortly.

An abstracted boundary has the associated metavariable arity

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

where $c \in \{\mathsf{Ty}, \mathsf{Tm}, \mathsf{EqTy}, \mathsf{EqTm}\}$ is the syntactic class of $\mathscr{b}$. Similarly, the associated metavariable arity of an argument is

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

where $c \in \{\mathsf{Ty}, \mathsf{Tm}\}$ is the syntactic class of the (non-abstracted) expression e.

The placeholder $\Box$ in a boundary $\mathcal{B}$ may be ***filled*** with an argument e, called the ***head***, to give a judgement $\mathcal{B}\boxed{e}$, provided that the arities of $\mathcal{B}$ and e match. Because equations are proof irrelevant, their placeholders can be filled uniquely with (suitably abstracted) dummy value $\star$. Filling is summarized in Fig. 2.3, where we also include notation for filling an object boundary with an equation that results in the corresponding equation. The figure rigorously explicates the dummy values, but we usually omit them. Filling may be inverted: given an abstracted judgement $\mathcal{J}$ there is a unique abstracted boundary $\mathcal{B}$ and a unique argument e such that $\mathcal{J} = \mathcal{B}\boxed{e}$.

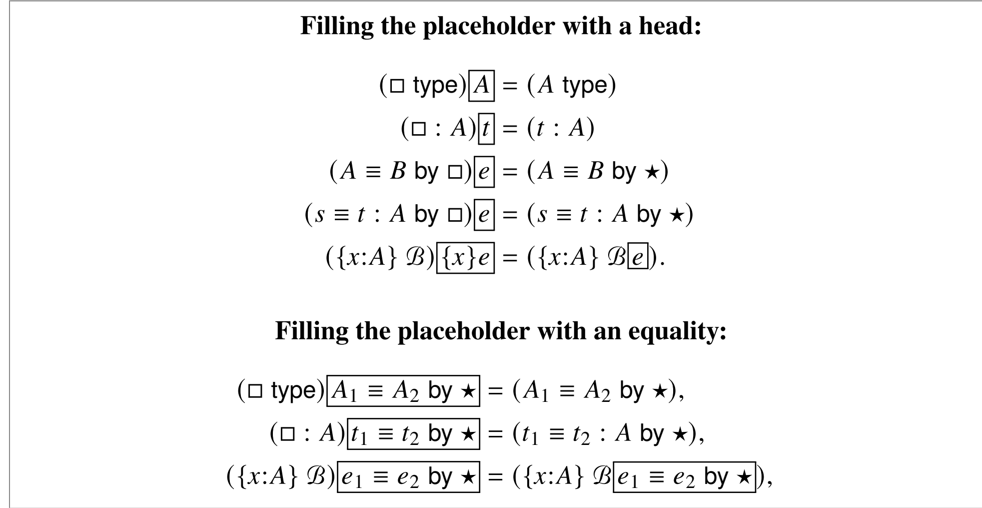


Figure 2.3: Filling the head of a boundary

Example 2.1.3. 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

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

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

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

Names of bound variables are immaterial, we would still get the same judgement if we filled the left-hand boundary with $\{u\}\{v\}u$ or $\{y\}\{x\}y$, but not with $\{x\}\{y\}y$.

Information about available metavariables is collected by a ***metavariable context***, which is a finite list $\Theta = [\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n]$, also construed as a map, assigning to each metavariable M_i a boundary $\mathcal{B}_i$. In Section 2.1.3, the assigned boundaries will assign the typing of metavariable, while at the level of raw syntax they determine metavariable arities. That is, Θ assigns the metavariable arity $\mathsf{ar}(\mathcal{B}_i)$ to M_i.

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

The metavariable context Θ is syntactically well formed when each $\mathcal{B}_i$ is a syntactically well-formed boundary in the signature $\langle \Sigma, \Theta_{(i)} \rangle$. In addition each $\mathcal{B}_i$ must be closed, i.e. contain no free variables.

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

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

It differs from traditional notion of a judgement in a non-essential way, which nevertheless requires an explanation. First, the context of a hypothetical judgement

$$\Theta; \mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_m{:}A_m \vdash \{x_1{:}B_1\} \cdots \{x_m{:}B_m\}\ j$$

provides information about metavariables, not just the free variables. Second, the variables are split between the 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. It is useful to think of the former as the *global* hypotheses that interact with other judgements, and the latter as *local* to the judgement. We could of course delegate the metavariable context to be part of the signature as is done in (Bauer, Haselwarter and Lumsdaine 2020), and revert to the more familiar form

$$\mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_n{:}A_n, x_1{:}B_1, \ldots, x_m{:}B_m \vdash j$$

by joining the variable context and the abstraction, but we would still have to carry the metavariable information in the signature, and would lose the ability to explicitly mark the split between the global and the local parts. The split will be especially important in Section 3.1, where the context will be removed, but the abstraction kept.

Hypothetical boundaries are formed in the same fashion, as

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

The intended meaning is that $\mathcal{B}$ is a well-typed boundary in context $\Theta; \Gamma$.

2.1.1.5 Metavariable instantiations

Metavariables are slots that can be instantiated with arguments. Suppose $\Theta = \langle \mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_k{:}\mathcal{B}_k \rangle$ is a metavariable context over a signature Σ. An ***instantiation of*** Θ ***over*** a context $\Xi; \Gamma$ is a seqence $I = \langle \mathsf{M}_1 {\mapsto} e_1, \ldots, \mathsf{M}_k {\mapsto} e_k \rangle$, representing a map that takes each M_i to an argument e_i over $\Theta; \Gamma$ such that $\mathsf{ar}(\mathcal{B}_i) = \mathsf{ar}(e_i)$.

An instantiation $I = \langle \mathsf{M}_1 {\mapsto} e_1, \ldots, \mathsf{M}_k {\mapsto} e_k \rangle$ of Θ may be *restricted* to an instantiation $I_{(i)} = \langle \mathsf{M}_1 {\mapsto} e_1, \ldots, \mathsf{M}_{i-1} {\mapsto} e_{i-1} \rangle$ of $\Theta_{(i)}$.

An instantiation I of Θ over $\Xi;\Gamma$ acts on an expression e over $\Theta;\Gamma$ to give an expression $I_* e$ over $\Xi;\Gamma$ in which the metavariables are replaced by the corresponding expressions, as follows:

$$I_* x = x, \qquad I_* \mathsf{a} = \mathsf{a}, \qquad I_* \star = \star, \qquad I_*(\{x\}e) = \{x\}(I_* e),$$
$$I_*(\mathsf{S}(e_1, \ldots, e_n)) = \mathsf{S}(I_* e_1, \ldots, I_* e_n),$$
$$I_*(\mathsf{M}_i(t_1, \ldots, t_n)) = e_i[(I_* t_1)/x_1, \ldots, (I_* t_{n_i})/x_{n_i}].$$

Abstracted judgements and boundaries may be instantiated too:

$$I_*(A \;\mathsf{type}) = (I_* A \;\mathsf{type}), \qquad I_*(t : A) = (I_* t : I_* A),$$
$$I_*(A \equiv B \;\mathsf{by}\; \star) = (I_* A \equiv I_* B \;\mathsf{by}\; \star), \qquad I_*(\{x{:}A\}\; \mathcal{J}) = \{x{:}I_* A\}\; I_* \mathcal{J},$$

and by imagining that $I_* \Box = \Box$, the reader can tell how to instantiate a boundary. Finally, a hypothetical judgement $\Theta;\Gamma \vdash \mathcal{J}$ may be instantiated to $\Xi; I_*\Gamma \vdash I_*\mathcal{J}$, and similarly for a hypothetical boundary.

2.1.2 Deductive systems

We briefly recall the notions of a deductive system, derivability, and a derivation tree; see for example (Aczel 1977) for an introduction. A ***(finitary) closure rule*** on a set S is a pair $([p_1, \ldots, p_n], q)$, also displayed as

$$\frac{p_1 \;\cdots\; p_n}{q},$$

where $\{p_1, \ldots, p_n\} \subseteq S$ are the ***premises*** and $q \in S$ is the ***conclusion***. Let $\mathsf{Clos}(S)$ be the set of all closure rules on S.

A ***deductive system*** (also called a *closure system*) on a set S is a family of closure rules $C : R \to \mathsf{Clos}(S)$, indexed by a set R of rule names. A set $D \subseteq S$ is said to be ***deductively closed for*** C when, for all $i \in R$, if $C_i = ([p_1, \ldots, p_n], q)$ and $\{p_1, \ldots, p_n\} \subseteq D$, then $q \in D$. The ***associated closure operator*** is the map $\mathcal{P}S \to \mathcal{P}S$ which takes $D \subseteq S$ to the least deductively closed supserset $\overline{D}$ of D, which exists by Tarski's fixed-point theorem (Tarski 1955). We say that $q \in S$ is ***derivable from hypotheses*** $H \subseteq S$ when $q \in \overline{H}$, and that it is ***derivable in*** C when $q \in \overline{\emptyset}$.

A closure rule $([p_1, \ldots, p_n], q)$ is ***admissible for*** C when $q \in \overline{\{p_1, \ldots, p_n\}}$. That is, adjoining an admissible closure rule to a closure system has no effect on its associated closure operator.

Derivability is witnessed by well-founded trees, which are constructed as follows. For each $q \in S$ let $\mathsf{Der}_C(q)$ be generated inductively by the clause (where der is a formal tag):

- for every $i \in R$, if $C_i = ([p_1, \ldots, p_n], q)$ and $t_j \in \mathsf{Der}_C(p_j)$ for all $j = 1, \ldots, n$, then $\mathsf{der}_i(t_1, \ldots, t_n) \in \mathsf{Der}_C(q)$.

The elements of $\mathsf{Der}_C(q)$ are ***derivation trees*** with ***conclusion*** q. Indeed, we may view $\mathsf{der}_i(t_1, \ldots, t_n)$ as tree with the root labeled by i and the subtrees $t_1, \ldots, t_n$. A leaf is a tree of the form $\mathsf{der}_j()$, which arises when the corresponding closure rule C_j has no premises.

Proposition 2.1.4. *Given a closure system C on S, an element $q \in S$ is derivable in C if, and only if, there exists a derivation tree over C whose conclusion is q.*

Proof. The claim is that $T = \{q \in S \mid \exists t \in \mathsf{Der}_C(q) . \top\}$ coincides with $\overline{C}$. The inclusion $\overline{C} \subseteq T$ holds because T is deductively closed. The reverse inclusion $T \subseteq \overline{C}$ is established by induction on derivation trees. □

We remark that allowing infinitary closure rules brings with it the need for the axiom of choice, for it is unclear how to prove that T is deductively closed without the aid of choice.

It is evident that derivability and derivation trees are monotone in all arguments: if $S \subseteq S'$, $R \subseteq R'$, and the closure system $C' : R' \to \mathsf{Clos}(S')$ restricts to $C : R \to \mathsf{Clos}(S)$, then any $q \in S$ derivable in C is also derivable in C' as an element of S'. Moreover, any derivation tree in $\mathsf{Der}_C(q)$ may be construed as a derivation tree in $\mathsf{Der}_{C'}(q)$.

Henceforth we shall consider solely deductive systems on the set of hypothetical judgements and boundaries. Because we shall vary the deductive system, it is useful to write $\Theta; \Gamma \vdash_C \mathcal{J}$ when $(\Theta; \Gamma \vdash \mathcal{J}) \in \overline{C}$, and similarly for $\Theta; \Gamma \vdash_C \mathcal{B}$.

2.1.3 Raw rules and type theories

A type theory in its basic form is a collection of closure rules. Some closure rules are specified directly, but many are presented by *inference rules* – templates whose instantiations yield the closure rules. We deal with the *raw* syntactic structure of such rules first.

Definition 2.1.5. A ***raw rule*** over a signature Σ is a hypothetical judgement over Σ of the form $\Theta; [\,] \vdash j$. We notate such a raw rule as

$$\Theta \Longrightarrow j.$$

The elements of Θ are the ***premises*** and j is the ***conclusion***. We say that the rule is an ***object rule*** when j is a type or a term judgement, and an ***equality rule*** when j is an equality judgement.

Defining inference rules as hypothetical judgements with empty contexts and empty abstractions permits in many situations uniform treatment of rules and judgements. Note that the premises and the conclusion may not contain any free variables, and that the conclusion must be non-abstracted. Neither condition impedes expressivity of raw rules, because free variables and abstractions may be promoted to premises.

Example 2.1.6. To help the readers' intuition, let us see how Definition 2.1.5 captures a traditional inference rule, such as product formation

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

The use of A and B in the premises reveals that their arities are $(\mathsf{Ty}, 0)$, and $(\mathsf{Ty}, 1)$, respectively. In fact, the premises assign boundaries to metavariables, namely each metavariable, applied generically, is the head of its boundary. If we pull out the metavariables from the heads of premises, the assignment becomes explicit:

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

This is just a different way of writing the raw rule

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

Example 2.1.7. We may translate raw rules back to their traditional form by filling the heads with metavariables applied generically. For example, the reader may readily verify that the raw rule

$$\mathsf{A}{:}(\Box\ \mathsf{type}),\ \mathsf{s}{:}(\Box : \mathsf{A}),\ \mathsf{t}{:}(\Box : \mathsf{A}),\ \mathsf{p}{:}(\Box : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})) \Longrightarrow \mathsf{s} \equiv \mathsf{t} : \mathsf{A}\ \mathsf{by}\ \star$$

corresponds to the *equality reflection* rule of extensional type theory that is traditionally written as

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A} \qquad \vdash \mathsf{p} : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A}}\ \textsc{Eq-Reflect}$$

For everyone's benefit, we shall display raw rules in traditional form, but use Definition 2.1.5 when formalities demand so.

It may be mystifying that there is no variable context Γ in a raw rule, for is it not the case that rules may be applied in arbitrary contexts? Indeed, *closure* rules have contexts, but raw rules do not because they are just templates. The context appears once we instantiate the template, as follows.

Definition 2.1.8. An ***instantiation*** of a raw rule $R = (\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n \Longrightarrow \mathscr{b}\boxed{e})$ over a signature Σ and context $\Theta; \Gamma$ is 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 associated ***closure rule*** I_*R is $([p_1, \ldots, p_n, q], r)$ where p_i is $\Theta; \Gamma \vdash (I_{(i)*}\mathcal{B}_i)\boxed{e_i}$, q is $\Theta; \Gamma \vdash I_*\mathscr{b}$, and r is $\Theta; \Gamma \vdash I_*(\mathscr{b}\boxed{e})$.

We included among the premises the well-formedness of the instantiated boundary $\Theta; \Gamma \vdash I_*\mathscr{b}$, so that the conclusion is well-formed. We need the premise as an induction hypothesis in the proof of Theorem 2.2.18. In Section 2.2.2 we shall formulate well-formedness conditions that allow us to drop the boundary premise.

Of special interest are the rules that give type-theoretic meaning to primitive symbols. To define them, we need the boundary analogue of raw rules.

Definition 2.1.9. A ***raw rule-boundary*** over a signature Σ is a hypothetical boundary over Σ of the form $\Theta; [\,] \vdash \mathscr{b}$. We notate such a raw rule-boundary as

$$\Theta \Longrightarrow \mathscr{b}.$$

The elements of Θ are the ***premises*** and $\mathscr{b}$ is the ***conclusion boundary***. We say that the rule-boundary is an ***object rule-boundary*** when $\mathscr{b}$ is a type or a term boundary, and an ***equality rule-boundary*** when $\mathscr{b}$ is an equality boundary.

Here is how a rule-boundary generates a rule associated to a symbol.

Definition 2.1.10. Given a raw object rule-boundary

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

over Σ, the ***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 $\mathscr{b}$. The ***associated symbol rule*** for $\mathsf{S} \notin |\Sigma|$ is the raw rule

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

over the extended signature $\langle \Sigma, \mathsf{S} {\mapsto} (c, [\mathsf{ar}(\mathscr{B}_1), \ldots, \mathsf{ar}(\mathscr{B}_n)]) \rangle$, where $\widehat{\mathsf{M}}$ is the ***generic application*** of the metavariable M with associated boundary $\mathscr{B}$, defined as:

1. $\widehat{M} = \{x_1\} \cdots \{x_k\} \mathsf{M}(x_1, \ldots, x_k)$ if $\mathsf{ar}(\mathscr{B}) = (c, k)$ and $c \in \{\mathsf{Ty}, \mathsf{Tm}\}$,
2. $\widehat{M} = \{x_1\} \cdots \{x_k\} \star$ if $\mathsf{ar}(\mathscr{B}) = (c, k)$ and $c \in \{\mathsf{EqTy}, \mathsf{EqTm}\}$.

A raw rule is said to be a ***symbol rule*** if it is the associated symbol rule for some symbol S.

The above definition separates the rule-boundary from the head of the conclusion because the latter can be calculated from the former. It would be less economical to define a symbol rule directly as a raw rule, as we would still have to verify that the supplied head is the expected one. In examples we shall continue to display symbol rules in their traditional form.

Example 2.1.11. According to Definition 2.1.10, the symbol rule for Π is generated by the rule-boundary

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

Indeed, the associated symbol rule for Π is

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

We allow equational premises in object rules. For example,

$$\textsc{Refl'}\ \frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A} \qquad \vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A}}{\vdash \mathsf{refl}(\mathsf{A}, \mathsf{s}, \mathsf{t}, \star) : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})}$$

is a valid symbol rule, assuming Id and refl have their usual arities.

We also record the analogous construction of an equality rule from a given equality rule-boundary.

Definition 2.1.12. Given an equality rule-boundary

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

the ***associated equality rule*** is

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

We next formulate the rules that all type theories share, starting with the most nitty-gritty ones, the congruence rules.

Definition 2.1.13. The ***congruence rules*** associated with a raw object rule R

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

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

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

In case of a term equation at type B, the congruence rule has the additional premise $\Theta; \Gamma \vdash I_* B \equiv J_* B$, which ensures that the right-hand side of the conclusion $J_* e$ has type $I_* B$. Having the equation avaliable as a premise allows us to use it in the inductive proof of Theorem 2.2.18. In Section 2.2.2 we show that the rule without the premises is admissible under suitable conditions.

Example 2.1.14. The congruence rule associated with the product formation rule from Example 2.1.6 is

$$\frac{\begin{array}{ll} \Theta; \Gamma \vdash A_1 \;\mathsf{type} & \Theta; \Gamma \vdash \{x{:}A_1\}\; B_1 \;\mathsf{type} \\ \Theta; \Gamma \vdash A_2 \;\mathsf{type} & \Theta; \Gamma \vdash \{x{:}A_2\}\; B_2 \;\mathsf{type} \\ \Theta; \Gamma \vdash A_1 \equiv A_2 & \Theta; \Gamma \vdash \{x{:}A_1\}\; B_1 \equiv B_2 \end{array}}{\Theta; \Gamma \vdash \Pi(A_1, \{x\}B_1) \equiv \Pi(A_2, \{x\}B_2)} \tag{2.1}$$

Next we have formation and congruence rules for the metavariables. As metavariables are like symbols whose arguments are terms, it is not suprising that their rules are quite similar to symbol rules.

Definition 2.1.15. Given a context $\Theta;\Gamma$ over Σ with $\Theta = [\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n]$, and $\mathcal{B}_k = (\{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathscr{b})$, the ***metavariable rules*** for M_k are the closure rules of the form

$$\frac{\begin{array}{l}\textsc{TT-Meta}\\ \Theta(\mathsf{M}_k) = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathscr{b} \\ \Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m \\ \Theta;\Gamma \vdash \mathscr{b}[\vec{t}/\vec{x}]\end{array}}{\Theta;\Gamma \vdash (\mathscr{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_k(\vec{t})}}$$

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

$$\frac{\begin{array}{ll}\textsc{TT-Meta-Congr} & \\ \Theta(\mathsf{M}_k) = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathscr{b} & \\ \Theta;\Gamma \vdash s_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta;\Gamma \vdash s_j \equiv t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta;\Gamma \vdash C[\vec{s}/\vec{x}] \equiv C[\vec{t}/\vec{x}] & \text{if } \mathscr{b} = (\Box : C)\end{array}}{\Theta;\Gamma \vdash (\mathscr{b}[\vec{s}/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}) \equiv \mathsf{M}_k(\vec{t})}}$$

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

We are finally ready to give a definition of type theory which is sufficient for explaining derivability.

Definition 2.1.16. A ***raw type theory*** T over a signature Σ is a family of raw rules over Σ, called the ***specific rules*** of T. The ***associated deductive system*** of T consists of:

1. the ***structural rules*** over Σ:
   a) the *variable*, *metavariable*, *metavariable congruence*, and *abstraction* closure rules (Fig. 2.4),
   b) the *equality* closure rules, (Fig. 2.5),
   c) the *boundary* closure rules (Fig. 2.6);
2. the instantiations of the specific rules of T (Definition 2.1.8);
3. for each specific object rule of T, the instantiations of the associated congruence rule (Definition 2.1.13).

We write $\Gamma \vdash_T \mathcal{J}$ when $\Gamma \vdash \mathcal{J}$ is derivable with respect to the deductive system associated to T, and similarly for $\Gamma \vdash_T \mathcal{B}$.

TT-Var
$$\frac{\mathsf{a} \in |\Gamma|}{\Theta; \Gamma \vdash \mathsf{a} : \Gamma(\mathsf{a})}$$

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

TT-Meta-Congr
$$\frac{\begin{array}{ll}\Theta(\mathsf{M}_k) = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathcal{b} & \\ \Theta; \Gamma \vdash s_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta; \Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta; \Gamma \vdash s_j \equiv t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta; \Gamma \vdash C[\vec{s}/\vec{x}] \equiv C[\vec{t}/\vec{x}] & \text{if } \mathcal{b} = (\Box : C)\end{array}}{\Theta; \Gamma \vdash (\mathcal{b}[\vec{s}/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}) \equiv \mathsf{M}_k(\vec{t})}}$$

TT-Abstr
$$\frac{\Theta; \Gamma \vdash A \ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma| \qquad \Theta; \Gamma, \mathsf{a}{:}A \vdash \mathcal{J}[\mathsf{a}/x]}{\Theta; \Gamma \vdash \{x{:}A\}\ \mathcal{J}}$$

Figure 2.4: Variable, metavariable and abstraction closure rules

TT-EqTy-Refl
$$\frac{\Theta; \Gamma \vdash A \ \mathsf{type}}{\Theta; \Gamma \vdash A \equiv A}$$

TT-EqTy-Sym
$$\frac{\Theta; \Gamma \vdash A \equiv B}{\Theta; \Gamma \vdash B \equiv A}$$

TT-EqTy-Trans
$$\frac{\Theta; \Gamma \vdash A \equiv B \qquad \Theta; \Gamma \vdash B \equiv C}{\Theta; \Gamma \vdash A \equiv C}$$

TT-EqTm-Refl
$$\frac{\Theta; \Gamma \vdash t : A}{\Theta; \Gamma \vdash t \equiv t : A}$$

TT-EqTm-Sym
$$\frac{\Theta; \Gamma \vdash s \equiv t : A}{\Theta; \Gamma \vdash t \equiv s : A}$$

TT-EqTm-Trans
$$\frac{\Theta; \Gamma \vdash s \equiv t : A \qquad \Theta; \Gamma \vdash t \equiv u : A}{\Theta; \Gamma \vdash s \equiv u : A}$$

TT-Conv-Tm
$$\frac{\Theta; \Gamma \vdash t : A \qquad \Theta; \Gamma \vdash A \equiv B}{\Theta; \Gamma \vdash t : B}$$

TT-Conv-EqTm
$$\frac{\Theta; \Gamma \vdash s \equiv t : A \qquad \Theta; \Gamma \vdash A \equiv B}{\Theta; \Gamma \vdash s \equiv t : B}$$

Figure 2.5: Equality closure rules

TT-Bdry-Ty
$$\frac{}{\Theta;\Gamma \vdash \Box\ \mathsf{type}}$$

TT-Bdry-Tm
$$\frac{\Theta;\Gamma \vdash A\ \mathsf{type}}{\Theta;\Gamma \vdash \Box : A}$$

TT-Bdry-EqTy
$$\frac{\Theta;\Gamma \vdash A\ \mathsf{type} \qquad \Theta;\Gamma \vdash B\ \mathsf{type}}{\Theta;\Gamma \vdash A \equiv B\ \mathsf{by}\ \Box}$$

TT-Bdry-EqTm
$$\frac{\Theta;\Gamma \vdash A\ \mathsf{type} \qquad \Theta;\Gamma \vdash s : A \qquad \Theta;\Gamma \vdash t : A}{\Theta;\Gamma \vdash s \equiv t : A\ \mathsf{by}\ \Box}$$

TT-Bdry-Abstr
$$\frac{\Theta;\Gamma \vdash A\ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma| \qquad \Gamma, \mathsf{a}{:}A \vdash \mathscr{B}[\mathsf{a}/x]}{\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B}}$$

Figure 2.6: Well-formed abstracted boundaries

Ext-Empty
$$\frac{}{\vdash [\,]\ \mathsf{mctx}}$$

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

Ctx-Empty
$$\frac{}{\Theta \vdash [\,]\ \mathsf{vctx}}$$

Ctx-Extend
$$\frac{\Theta \vdash \Gamma\ \mathsf{vctx} \qquad \Theta;\Gamma \vdash A\ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma|}{\Theta \vdash (\Gamma, \mathsf{a}{:}A)\ \mathsf{vctx}}$$

Figure 2.7: Well-formed metavariable and variable contexts

Several remarks are in order regarding the above definition and the rules in Figs. 2.4 to 2.6:

1. It is assumed throughout that all the entities involved are syntactically valid, i.e. that arities are respected and variables are well-scoped.

2. The metavariable rules TT-Meta and TT-Meta-Congr are exactly as in Definition 2.1.15.

3. The rules TT-Var, TT-Meta, and TT-Abstr contain *side-conditions*, such as $\mathsf{a} \in |\Gamma|$ and $\Theta(\mathsf{M}) = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathscr{b}$. For purely aesthetic reasons, these are written where premises ought to stand. For example, the correct way to read TT-Abstr is: "For all Θ, Γ, A, a, $\mathscr{J}$, if $\mathsf{a} \notin |\Gamma|$, then there is a closure rule with premises $\Theta;\Gamma \vdash A\ \mathsf{type}$ and $\Theta;\Gamma, \mathsf{a}{:}A \vdash \mathscr{J}[\mathsf{a}/x]$, and the conclusion $\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{J}$."

4. The structural rules impose no well-typedness conditions on contexts. Instead, Fig. 2.7 provides two auxiliary judgement forms, "$\vdash \Theta\ \mathsf{mctx}$" and "$\Theta \vdash \Gamma\ \mathsf{vctx}$", stating that Θ is a well-typed metavariable context, and Γ a well-typed variable

context over Θ, respectively. These will be used as necessary. Note that imposing the additional premise $\Theta; \Gamma \vdash \Gamma(\mathsf{a})\ \mathsf{type}$ in TT-VAR would *not* ensure well-formednes of Γ, as not all variables need be accessed in a derivation. Requiring that TT-META check the boundary of the metavariable is similarly ineffective.

5. We shall show in Section 2.2.1 that substitution rules (Fig. 2.8) are admissible.

This may be a good moment to record the difference between derivability and admissibility.

Definition 2.1.17. Consider a raw theory T and a raw rule R, both over a signature Σ:

1. R is ***derivable*** in T when it has a derivation in T.

2. R is ***admissible*** in T when, for every instantiation I of R, the conclusion of $I_* R$ is derivable in T from the premises of $I_* R$.

2.1.4 Finitary rules and type theories

Raw rules are *syntactically* well-behaved: the premises and the conclusion are syntactically well-formed entities, and all metavariables, free variable and bound variables well-scoped. Nevertheless, a raw rule may be ill-formed for type-theoretic reasons, a deficiency rectified by the next definition.

Recall that a ***well-founded order*** on a set I is an irreflexive and transitive relation $\prec$ satisfying, for each $S \subseteq I$,

$$(\forall i \in I \,.\, (\forall j \prec i \,.\, j \in S) \Rightarrow i \in S) \Rightarrow S = I.$$

The logical reading of the above condition is an induction principle: in order to show $\forall x \in I \,.\, \phi(x)$ one has to prove, for any $i \in I$, that $\phi(i)$ holds assuming that $\phi(j)$ does for all $j \prec i$.

Definition 2.1.18. Given a raw theory T over a signature Σ, a raw rule $\Theta \Longrightarrow \mathcal{B}\boxed{e}$ over Σ is ***finitary*** over T when $\vdash_T \Theta\ \mathsf{mctx}$ and $\Theta; [\,] \vdash_T \mathcal{B}$. Similarly, a raw rule-boundary $\Theta \Longrightarrow \mathcal{B}$ is finitary when $\vdash_T \Theta\ \mathsf{mctx}$ and $\Theta; [\,] \vdash_T \mathcal{B}$.

A ***finitary type theory*** is a raw type theory $(R_i)_{i \in I}$ for which there exists a well-founded order $(I, \prec)$ such that each R_i is finitary over $(R_j)_{j \prec i}$.

Example 2.1.19. We take stock by considering several examples of rules. The rule

$$\textsc{Unique-Ty}\quad \frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{B}\ \mathsf{type} \qquad \vdash \mathsf{t} : \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{t} : \mathsf{B}\ \mathsf{type}}{\vdash \mathsf{A} \equiv \mathsf{B}}$$

is not raw because it introduces the metavariable t twice. Assuming Π has arity $(\mathsf{Ty}, [(\mathsf{Ty}, 0), (\mathsf{Ty}, 1)])$, consider the rules

$$\textsc{Ty-}\Pi\textsc{-Short}\quad \frac{\vdash \{x{:}\mathsf{A}\}\ \mathsf{B}(x)\ \mathsf{type}}{\vdash \Pi(\mathsf{A}, \{x\}\mathsf{B}(x))} \qquad \textsc{Ty-}\Pi\textsc{-Long}\quad \frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \{x{:}\mathsf{A}\}\ \mathsf{B}(x)\ \mathsf{type}}{\vdash \Pi(\mathsf{A}, \{x\}\mathsf{B}(x))}$$

The rule Ty-Π-Short is not raw because it fails to introduce the metavariable A, while Ty-Π-Long is finitary over any theory. The rule

$$\textsc{Succ-Congr-Typo}\quad \frac{\vdash \mathsf{m} : \mathsf{nat} \qquad \vdash \mathsf{n} : \mathsf{bool} \qquad \vdash \mathsf{m} \equiv \mathsf{n} : \mathsf{nat}}{\vdash \mathsf{succ}(\mathsf{m}) \equiv \mathsf{succ}(\mathsf{n}) : \mathsf{nat}}$$

is raw when the symbols bool, nat, and succ respectively have arities $(\mathsf{Ty}, [])$, $(\mathsf{Ty}, [])$, and $(\mathsf{Tm}, [(\mathsf{Tm}, 0)])$. Whether it is also finitary depends on a theory. For instance, given the raw rules

$$\textsc{Ty-Bool}\quad \frac{}{\vdash \mathsf{bool}\ \mathsf{type}} \qquad \textsc{Ty-Nat}\quad \frac{}{\vdash \mathsf{nat}\ \mathsf{type}} \qquad \textsc{Tm-Succ}\quad \frac{\vdash \mathsf{n} : \mathsf{nat}}{\vdash \mathsf{succ}(\mathsf{n}) : \mathsf{nat}} \qquad \textsc{Bool-Eq-Nat}\quad \frac{}{\vdash \mathsf{bool} \equiv \mathsf{nat}}$$

the rule Succ-Congr-Typo is not finitary over the first three rules, but is finitary over all four of them. As a last example, given the symbol Id with arity $(\mathsf{Ty}, [(\mathsf{Ty}, 0), (\mathsf{Tm}, 0), (\mathsf{Tm}, 0)])$, the rules

$$\textsc{Ty-Id}\quad \frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A}}{\vdash \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})\ \mathsf{type}} \qquad \textsc{Ty-Id-Typo}\quad \frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A}}{\vdash \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{s})\ \mathsf{type}}$$

$$\textsc{Eq-Reflect}\quad \frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A} \qquad \vdash \mathsf{p} : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A}}$$

are all raw, both Ty-Id and Ty-Id-Typo are finitary over an empty theory, while Eq-Reflect is finitary over a theory containing Ty-Id. The rule Ty-Id is a symbol rule, but Ty-Id-Typo does not.

Could we have folded Definition 2.1.5 of raw rules and Definition 2.1.18 of finitary rules into a single definition? Not easily, as that would generate a loop: finitary rules refer to theories and derivability, which refer to closure rules, which are generated from raw rules. Without a doubt something is to be learned by transforming the cyclic dependency to an inductive definition, but we do not attempt to do so here.

A finitary type theory is fairly well behaved from a type-theoretic point of view, but can still suffer from unusual finitary rules, such as Ty-Id-Typo from Example 2.1.19, which looks like a spelling mistake. We thus impose a further restriction by requiring that every rule be either a symbol rule or an equality rule.

Definition 2.1.20. A finitary type theory is ***standard*** if its specific object rules are symbol rules, and each symbol has precisely one associated rule.

A standard type theory and its signature may be built iteratively as follows:

1. The empty theory is standard over the empty signature.
2. Given a standard type theory T over Σ, and a rule-boundary

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

   finitary for T:

   - If $\mathscr{b}$ is an object boundary, and $\mathsf{S} \notin |\Sigma|$, then T extended with the associated symbol rule

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

     is standard over the extended signature $\langle \Sigma, \mathsf{S} {\mapsto} \alpha \rangle$, where α is the symbol arity associated with the rule-boundary.
   - If $\mathscr{b}$ is an equation boundary, then T extended with the equality rule

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

     is standard over Σ.

A more elaborate well-founded induction may be employed when a theory features infinitely many rules, such as an infinite succession of universes.

2.2 Metatheorems

We put our definitions to the test by proving metatheorems which stipulate desirable structural properties of type theories. The theorems are all rather standard and expected, and so are their proofs. Nevertheless, we prove them to verify that our definition of type theories is sensible, and to provide general-purpose metatheorems that apply in a wide range of situations.

2.2.1 Metatheorems about raw theories

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 contexts, judgements, and boundaries.

Proposition 2.2.1 (Renaming)**.** *If a raw type theory derives a judgement or a boundary, then it also derives its renaming.*

Proof. Let ρ be a renaming of a derivable judgement $\Theta; \Gamma \vdash \mathcal{J}$. We show that $\rho_*\Theta; \rho_*\Gamma \vdash \rho_*\mathcal{J}$ is derivable by induction on the derivation. The case of boundaries is similar.

Most cases only require a direct application of the induction hypotheses to the premises. The only somewhat interesting case is TT-Abstr,

$$\frac{\Theta; \Gamma \vdash A \ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma| \qquad \Theta; \Gamma, \mathsf{a}{:}A \vdash \mathcal{J}[\mathsf{a}/x]}{\Theta; \Gamma \vdash \{x{:}A\}\ \mathcal{J}}$$

As $\mathsf{a} \notin |\Gamma|$, and thus $\mathsf{a} \notin |\rho|$, we may extend ρ to a renaming $\rho' = \langle \rho, \mathsf{a} \mapsto \mathsf{b} \rangle$, where b is such that $\mathsf{b} \notin |\rho_*\Gamma|$. By induction hypothesis for the first premise, $\rho_*\Theta; \rho_*\Gamma \vdash \rho_*A\ \mathsf{type}$ is derivable. We apply the induction hypothesis for the second premise to ρ' and obtain $\rho'_*\Theta; \rho'_*(\Gamma, \mathsf{a}{:}A) \vdash \rho'_*(\mathcal{J}[\mathsf{a}/x])$, which equals $\rho_*\Theta; \rho_*\Gamma, \mathsf{b}{:}\rho_*A \vdash (\rho_*\mathcal{J})[\mathsf{b}/x]$. Thus, we may conclude by TT-Abstr,

$$\frac{\rho_*\Theta; \rho_*\Gamma \vdash (\rho_*A)\ \mathsf{type} \qquad \mathsf{b} \notin |\rho_*\Gamma| \qquad \rho_*\Theta; \rho_*\Gamma, \mathsf{b}{:}\rho_*A \vdash (\rho_*\mathcal{J})[\mathsf{b}/x]}{\rho_*\Theta; \rho_*\Gamma \vdash \{x{:}\rho_*A\}\ \rho_*\mathcal{J}}$$ □

Proposition 2.2.2 (Weakening)**.** *For a raw type theory:*

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

An analogous statement holds for boundaries.

Proof. Once again we proceed by induction on the derivation of the judgement in a straightforward manner, where the case TT-Abstr relies on renaming (Proposition 2.2.1) to ensure that a remains fresh in the subderivations. □

In several places we shall require well-formedness of contexts, a useful consequence of which we record first.

Proposition 2.2.3. *If a raw type theory derives* $\vdash \Theta\ \mathsf{mctx}$ *then it derives* $\Theta; [\,] \vdash \Theta(\mathsf{M})$ *for every* $\mathsf{M} \in |\Theta|$*; and if it derives* $\Theta \vdash \Gamma\ \mathsf{vctx}$*, then it derives* $\Theta; \Gamma \vdash \Gamma(\mathsf{a})\ \mathsf{type}$ *for every* $\mathsf{a} \in |\Gamma|$.

Proof. By induction on the derivation of $\vdash \Theta\ \mathsf{mctx}$ and $\Theta \vdash \Gamma\ \mathsf{vctx}$, respectively, followed by weakening. □

2.2.1.1 Admissibility of substitution

In this section we prove that in a raw type theory substitution is admissible, and that substitution preserves judgemental equality.

Lemma 2.2.4. *If a raw type theory derives* $\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash \mathcal{J}$ *and* $\Theta; \Gamma \vdash t : A$ *then it derives* $\Theta; \Gamma, \Delta[t/\mathsf{a}] \vdash \mathcal{J}[t/\mathsf{a}]$.

Proof. We proceed by induction on the derivation of the judgement. The induction is mutual with the corresponding statement for boundaries, Lemma 2.2.5.

Case TT-VAR: If the derivation ends with the variable rule for a then we apply weakening to $\Theta; \Gamma \vdash t : A$ to get $\Theta; \Gamma, \Delta[t/\mathsf{a}] \vdash t : A$. For other variables, we apply the variable rule for the same variable.

Case TT-ABSTR: Consider a derivation which ends with an abstraction

$$\frac{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash B \ \mathsf{type} \qquad \mathsf{b} \notin |\Gamma, \mathsf{a}{:}A, \Delta| \qquad \Theta; \Gamma, \mathsf{a}{:}A, \Delta, \mathsf{b}{:}B \vdash \mathcal{J}[\mathsf{b}/x]}{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash \{x{:}B\}\ \mathcal{J}}$$

The induction hypotheses for the premises yield

$$\Theta; \Gamma, \Delta[t/\mathsf{a}] \vdash B[t/\mathsf{a}] \ \mathsf{type} \quad \text{and} \quad \Theta; \Gamma, \Delta[t/\mathsf{a}], \mathsf{b}{:}B[t/\mathsf{a}] \vdash (\mathcal{J}[\mathsf{b}/x])[t/\mathsf{a}].$$

Note that $(\mathcal{J}[\mathsf{b}/x])[t/\mathsf{a}] = (\mathcal{J}[t/\mathsf{a}])[\mathsf{b}/x]$, because $x \notin \mathsf{bv}(t)$ as t is closed, and $\mathsf{a} \neq \mathsf{b}$ by assumption. Hence abstracting b in the second premise yields

$$\Theta; \Gamma, \Delta[t/\mathsf{a}] \vdash \{x{:}B[t/\mathsf{a}]\}\ \mathcal{J}[t/\mathsf{a}],$$

as desired.

Case TT-META *and* TT-META-CONGR: We only consider the congruence rules, as the metavariable rule is treated similarly. Consider a derivation which ends with the congruence rule for a metavariable M whose boundary is $\Theta(\mathsf{M}) = \{\vec{x}{:}\vec{B}\}\ \mathscr{b}$:

$$\frac{\begin{array}{c} \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash s_j : B_j[\vec{s}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m \\ \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash t_j : B_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m \\ \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash s_j \equiv t_j : B_j[\vec{s}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m \\ \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash C[\vec{s}/\vec{x}] \equiv C[\vec{t}/\vec{x}] \quad \text{if } \mathscr{b} = \Box : C \end{array}}{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash (\mathscr{b}[\vec{s}/\vec{x}])\boxed{\mathsf{M}(\vec{s}) \equiv \mathsf{M}(\vec{t})}}$$

We apply the induction hypotheses to the premises, and conclude by TT-META-CONGR for M, applied to $\vec{s}[\mathsf{a}/x]$ and $\vec{t}[\mathsf{a}/x]$, taking into account that in general $(e[u/x])[v/\mathsf{a}] = (e[v/\mathsf{a}])[u[v/\mathsf{a}]/x]$.

Case of a specific rule: Consider a derivation ending with the application of a raw rule $R = (\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow j)$ with $j = \mathscr{b}\boxed{e}$, instantiated by $I = \langle \mathsf{M}_1 {\mapsto} e_1, \ldots, \mathsf{M}_n {\mapsto} e_n \rangle$,

$$\frac{\begin{array}{c} \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i} \quad \text{for } i = 1, \ldots, n \\ \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash I_*\mathscr{b} \end{array}}{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash I_* j}$$

The induction hypotheses for the premises yield, for $i = 1, \ldots, n$,

$$\Theta; \Gamma, \Delta[t/\mathsf{a}] \vdash ((I_{(i)*}\mathscr{B}_i)\boxed{e_i})[t/\mathsf{a}],$$

which equals

$$\Theta;\Gamma,\Delta[t/\mathsf{a}] \vdash (I[t/\mathsf{a}]_{(i)*}\mathscr{B}_i)\boxed{e_i[t/\mathsf{a}]}.$$

By Lemma 2.2.5, we further obtain $\Theta;\Gamma,\Delta \vdash (I[t/\mathsf{a}])_*\mathscr{b}$. Now apply R instantiated at $I[t/a] = \langle \mathsf{M}_1{\mapsto}e_1[t/\mathsf{a}], \ldots, \mathsf{M}_n{\mapsto}e_n[t/\mathsf{a}]\rangle$ to derive $\Theta;\Gamma,\Delta[t/\mathsf{a}] \vdash I[t/a]_*j$, which equals $\Theta;\Gamma,\Delta[t/\mathsf{a}] \vdash (I_*j)[t/\mathsf{a}]$.

Case of a congruence rule: Apply the induction hypotheses to the premises and conclude by the same rule.

Cases TT-EqTy-Refl, TT-EqTy-Sym, TT-EqTy-Trans, TT-EqTm-Refl, TT-EqTm-Sym, TT-EqTm-Trans, TT-Conv-Tm, TT-Conv-EqTm: These cases are dispensed with, once again, by straightforward applications of the induction hypotheses. □

Lemma 2.2.5. *If a raw type theory derives* $\Theta;\Gamma,\mathsf{a}{:}A,\Delta \vdash \mathscr{B}$ *and* $\Theta;\Gamma \vdash t : A$ *then it derives* $\Theta;\Gamma,\Delta[t/\mathsf{a}] \vdash \mathscr{B}[t/\mathsf{a}]$.

Proof. The base cases immediately reduce to the previous lemma. The case of TT-Bdry-Abstr is similar to the case of TT-Abstr in the previous lemma. □

Lemma 2.2.6. *In a raw type theory the following rules are admissible:*

$$\frac{\Theta;\Gamma \vdash \{x{:}A\}\ \mathcal{J} \qquad \Theta;\Gamma \vdash t : A}{\Theta;\Gamma \vdash \mathcal{J}[t/x]}\ \text{TT-Subst} \qquad \frac{\Theta;\Gamma \vdash \{x{:}A\}\ \mathscr{B} \qquad \Theta;\Gamma \vdash t : A}{\Theta;\Gamma \vdash \mathscr{B}[t/x]}\ \text{TT-Bdry-Subst}$$

$$\frac{\Theta;\Gamma \vdash \{x{:}A\}\ \mathcal{J} \qquad \Theta;\Gamma \vdash B\ \mathsf{type} \qquad \Theta;\Gamma \vdash A \equiv B}{\Theta;\Gamma \vdash \{x{:}B\}\ \mathcal{J}}\ \text{TT-Conv-Abstr}$$

Proof. Suppose the premises of TT-Subst are derivable. By inversion the first premise is derived by an application of TT-Abstr, therefore for some $\mathsf{a} \notin |\Gamma|$, we can derive $\Theta;\Gamma,\mathsf{a}{:}A \vdash \mathcal{J}[\mathsf{a}/x]$. Lemma 2.2.4 yields $\Theta;\Gamma \vdash (\mathcal{J}[\mathsf{a}/x])[t/\mathsf{a}]$, which is equal to the conclusion of TT-Subst.

The rule TT-Bdry-Subst follows from Lemma 2.2.5.

Next, assuming the premises of TT-Conv-Abstr are derivable, its conclusion is derived as

$$\dfrac{\Theta;\Gamma \vdash B\ \mathsf{type} \qquad \dfrac{\dfrac{\Theta;\Gamma \vdash \{x{:}A\}\ \mathcal{J}}{\Theta;\Gamma,\mathsf{a}{:}B \vdash \{x{:}A\}\ \mathcal{J}} \qquad \dfrac{\dfrac{}{\Theta;\Gamma,\mathsf{a}{:}B \vdash \mathsf{a}{:}B} \qquad \dfrac{\dfrac{\Theta;\Gamma \vdash A \equiv B}{\Theta;\Gamma \vdash B \equiv A}}{\Theta;\Gamma,\mathsf{a}{:}B \vdash B \equiv A}}{\Theta;\Gamma,\mathsf{a}{:}B \vdash \mathsf{a}{:}A}}{\Theta;\Gamma,\mathsf{a}{:}B \vdash \mathcal{J}[\mathsf{a}/x]}\ \text{TT-Subst}}{\Theta;\Gamma \vdash \{x{:}B\}\ \mathcal{J}}$$

□

The next lemma claims that substitution preserves equality, but is a bit finicky to state. Given terms s and t, and an object judgement $\mathcal{J}$, define $\mathcal{J}[(s \equiv t)/\mathsf{a}]$ by

$$\begin{aligned} (A \ \mathsf{type})[(s \equiv t)/\mathsf{a}] &= (A[s/\mathsf{a}] \equiv A[t/\mathsf{a}]) \\ (u : A)[(s \equiv t)/\mathsf{a}] &= (u[s/\mathsf{a}] \equiv u[t/\mathsf{a}] : A[s/\mathsf{a}]) \\ (\{x{:}A\}\ \mathcal{J})[(s \equiv t)/\mathsf{a}] &= (\{x{:}A[s/\mathsf{a}]\}\ \mathcal{J}[(s \equiv t)/\mathsf{a}]). \end{aligned}$$

That is, $\mathcal{J}[(s \equiv t)/\mathsf{a}]$ descends into abstractions by substituting s for a in the types, and distributes types and terms over the equation $s \equiv t$.

Lemma 2.2.7. *If a raw type theory derives*

$$\Theta; \Gamma \vdash s : A, \tag{2.2}$$

$$\Theta; \Gamma \vdash t : A, \tag{2.3}$$

$$\Theta; \Gamma \vdash s \equiv t : A. \tag{2.4}$$

$$\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash \mathcal{J}, \tag{2.5}$$

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash B[s/\mathsf{a}] \equiv B[t/\mathsf{a}] \quad \text{for all } \mathsf{b} \in |\Delta| \text{ with } \Delta(\mathsf{b}) = B, \tag{2.6}$$

then it derives

1. $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash \mathcal{J}[s/\mathsf{a}]$,
2. $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash \mathcal{J}[t/\mathsf{a}]$, *and*
3. $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash \mathcal{J}[(s \equiv t)/\mathsf{a}]$ *if $\mathcal{J}$ is an object judgement.*

Proof. We proceed by induction on the derivation of (2.5).

Case TT-VAR: For a variable $\mathsf{b} \in |\Gamma|$, (1) and (2) follow by the same variable rule, while (3) follows by reflexivity for b and the same variable rule.

For the variable a, the desired judgements are precisely the assumptions (2.2), (2.3), and (2.4) weakened to $\Gamma, \Delta[s/\mathsf{a}]$.

For a variable $\mathsf{b} \in |\Delta|$ with $B = \Delta(\mathsf{b})$, the same variable rule derives $\Theta; \Delta[s/\mathsf{a}] \vdash \mathsf{b} : B[s/\mathsf{a}]$ to satisfy (1), while (2) requires an additional conversion along

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash B[s/\mathsf{a}] \equiv B[t/\mathsf{a}] \tag{2.7}$$

which is just (2.6). To show (3), namely $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash \mathsf{b} \equiv \mathsf{b} : B[s/\mathsf{a}]$, we use TT-EQTM-REFL and the variable rule.

Case TT-ABSTR: Consider a derivation ending with an abstraction

$$\frac{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash B \ \mathsf{type} \qquad \mathsf{b} \notin |\Gamma, \mathsf{a}{:}A, \Delta| \qquad \Theta; \Gamma, \mathsf{a}{:}A, \Delta, \mathsf{b}{:}B \vdash \mathcal{J}[\mathsf{b}/x]}{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash \{x{:}B\}\ \mathcal{J}}$$

The induction hypothesis (1) applied to the first premise yields

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash B[s/\mathsf{a}] \ \mathsf{type}, \tag{2.8}$$

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash B[s/\mathsf{a}] \equiv B[t/\mathsf{a}]. \tag{2.9}$$

Equation (2.9) ensures that the extended variable context $\Delta, \mathsf{b}{:}B$ satisfies (2.6), hence we may use the induction hypothesis (1) for the last premise to show

$$\Theta; \Gamma, \Delta[s/\mathsf{a}], \mathsf{b}{:}B[s/\mathsf{a}] \vdash \mathcal{J}[\mathsf{b}/x][s/\mathsf{a}],$$

which equals

$$\Theta; \Gamma, \Delta[s/\mathsf{a}], \mathsf{b}{:}B[s/\mathsf{a}] \vdash \mathcal{J}[s/\mathsf{a}][\mathsf{b}/x]. \tag{2.10}$$

We can thus use the abstraction rule with (2.8) and (2.10) to derive $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash \{x{:}B[s/\mathsf{a}]\}\ \mathcal{J}[s/\mathsf{a}]$, as required.

The derivation of $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash \{x{:}B[t/\mathsf{a}]\}\ \mathcal{J}[t/\mathsf{a}]$ is more interesting. We first apply induction hypothesis (2) to the last premise and get

$$\Theta; \Gamma, \Delta[s/\mathsf{a}], \mathsf{b}{:}B[s/\mathsf{a}] \vdash \mathcal{J}[\mathsf{b}/x][t/\mathsf{a}].$$

Abstraction now gets us to $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash \{x{:}B[s/\mathsf{a}]\}\ \mathcal{J}[t/\mathsf{a}]$, after which we apply TT-Conv-Abstr from Lemma 2.2.6 to replace $B[s/\mathsf{a}]$ with $B[t/\mathsf{a}]$ using (2.9).

Lastly, we use the induction hypothesis (3) for the last premise to derive

$$\Theta; \Gamma, \Delta[s/\mathsf{a}], \mathsf{b}{:}B[s/\mathsf{a}] \vdash (\mathcal{J}[\mathsf{b}/x])[(s \equiv t)/\mathsf{a}],$$

which equals

$$\Theta; \Gamma, \Delta[s/\mathsf{a}], \mathsf{b}{:}B[s/\mathsf{a}] \vdash (\mathcal{J}[(s \equiv t)/\mathsf{a}])[\mathsf{b}/x]. \tag{2.11}$$

We may thus apply abstraction to (2.8) and (2.11) to derive

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash \{x{:}B[s/\mathsf{a}]\}\ \mathcal{J}[(s \equiv t)/\mathsf{a}],$$

as desired.

Case TT-Meta: Suppose (2.5) concludes with the metavariable rule for M, where $\Theta(\mathsf{M}) = \mathcal{B} = (\{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\ \mathcal{b})$:

$$\frac{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash u_i : A_i[\vec{u}_{(i)}/\vec{x}_{(i)}] \quad \text{for } i = 1, \ldots, n \qquad \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash \mathcal{b}[\vec{u}/\vec{x}]}{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash ((\mathcal{b}[\vec{u}/\vec{x}])\boxed{\mathsf{M}(\vec{u})})} \tag{2.12}$$

Judgements (1) and (2) are derived by the metavariable rule for M, applied to the corresponding induction hypotheses for the premises of (2.12). We address (3) in case $\mathcal{b} = (\Box : B)$, and leave the simpler case $\mathcal{b} = (\Box\ \mathsf{type})$ to the reader. We thus seek a derivation of

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash (\mathsf{M}(\vec{u}) : B[\vec{u}/\vec{x}])[(s \equiv t)/\mathsf{a}]$$

which equals

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash \mathsf{M}(\vec{u}[s/\mathsf{a}]) \equiv \mathsf{M}(\vec{u}[t/\mathsf{a}]) : B[\vec{u}[s/\mathsf{a}]/\vec{x}].$$

This is just the conclusion of the congruence rule TT-Meta-Congr for M, suitably applied so that its term and term equation premises are precisely the induction

hypotheses (1,2,3) for the term premises of (2.12), and its type equation premise is obtained by application of the induction hypothesis (3) to the last premise of (2.12).

Case TT-META-CONGR: If (2.5) ends with a congruence rule for an object metavariable M then both (1) and (2) follow by the same congruence rule, applied to the respective induction hypotheses for the premises.

Case of a specific rule: Suppose (2.5) ends with an application of the raw rule $R = (\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n \Longrightarrow j)$ instantiated with $I = \langle \mathsf{M}_1 {\mapsto} e_1, \ldots, \mathsf{M}_n {\mapsto} e_n \rangle$:

$$\frac{\begin{array}{ll} \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash (I_{(i)*}\mathscr{B}_i)\boxed{e_i} & \text{for } i = 1, \ldots, n \\ \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash I_* \mathscr{b} & \text{where } j = \mathscr{b}\boxed{e} \end{array}}{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash I_* j} \tag{2.13}$$

We would like to derive

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash (I_* j)[s/\mathsf{a}], \tag{2.14}$$

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash (I_* j)[t/\mathsf{a}], \tag{2.15}$$

and in case j is an object judgement, also

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash (I_* j)[(s \equiv t)/\mathsf{a}]. \tag{2.16}$$

We derive (2.14) by $(I[s/\mathsf{a}])_* R$ where $I[s/\mathsf{a}] = \langle \mathsf{M}_1 {\mapsto} e_1[s/\mathsf{a}], \ldots, \mathsf{M}_n {\mapsto} e_n[s/\mathsf{a}] \rangle$, as its premises are induction hypotheses. Similarly, (2.15) is derived by $(I[t/\mathsf{a}])_* R$. We consider (2.16) in case $j = (u : B)$ and leave the simpler case $j = (B\ \mathsf{type})$ to the reader. We thus need to derive

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash (I_* u)[s/\mathsf{a}] \equiv (I_* u)[t/\mathsf{a}] : (I_* B)[s/\mathsf{a}], \tag{2.17}$$

which we do by applying the congruence rule, where $J = I[s/\mathsf{a}]$ and $K = I[t/\mathsf{a}]$,

$$\frac{\begin{array}{ll} \Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash (J_{(i)*}\mathscr{B}_i)\boxed{e_i[s/\mathsf{a}]} & \text{for } i = 1, \ldots, n \\ \Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash (K_{(i)*}\mathscr{B}_i)\boxed{e_i[t/\mathsf{a}]} & \text{for } i = 1, \ldots, n \\ \Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash (J_{(i)*}\mathscr{B}_i)\boxed{e_i[s/\mathsf{a}] \equiv e_i[t/\mathsf{a}]} & \text{for object boundary } \mathscr{B}_i \\ \Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash J_* B \equiv K_* B & \end{array}}{\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash J_* u \equiv K_* u : J_* B}$$

The first three rows of premises are just the the induction hypotheses for the first row of premises of (2.13), and the last one is (3) for the last premise of (2.13).

Case of a congruence rule: Both (1) and (2) are derived by applying the induction hypotheses to the premises and using the congruence rule.

Case TT-CONV-TM: Consider a derivation ending with a conversion

$$\frac{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash u : B \qquad \Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash B \equiv C}{\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash u : C}$$

The judgements $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash u[s/\mathsf{a}] : C[s/\mathsf{a}]$ and $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash u[t/\mathsf{a}] : C[t/\mathsf{a}]$ immediately follow from the induction hypothesis and conversion. To derive $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash (u : C)[(s \equiv t)/\mathsf{a}]$, note that the induction hypothesis (3) for the first premise yields

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash u[s/\mathsf{a}] \equiv u[t/\mathsf{a}] : B[s/\mathsf{a}],$$

and (1) applied to the second premise

$$\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash B[s/\mathsf{a}] \equiv C[s/\mathsf{a}].$$

Thus by equality conversion we conclude $\Theta; \Gamma, \Delta[s/\mathsf{a}] \vdash u[s/\mathsf{a}] \equiv u[t/\mathsf{a}] : C[s/\mathsf{a}]$.

Cases TT-EqTy-Refl, TT-EqTy-Sym, TT-EqTy-Trans, TT-EqTm-Refl, TT-EqTm-Sym, TT-EqTm-Trans, TT-Conv-EqTm: These cases are dispensed with by straightforward applications of the induction hypotheses. □

TT-Subst
$$\frac{\Theta; \Gamma \vdash \{x{:}A\}\ \mathcal{J} \qquad \Theta; \Gamma \vdash t : A}{\Theta; \Gamma \vdash \mathcal{J}[t/x]}$$

TT-Bdry-Subst
$$\frac{\Theta; \Gamma \vdash \{x{:}A\}\ \mathcal{B} \qquad \Theta; \Gamma \vdash t : A}{\Theta; \Gamma \vdash \mathcal{B}[t/x]}$$

TT-Subst-EqTy
$$\frac{\begin{array}{c}\Theta; \Gamma \vdash \{x{:}A\}\{\vec{y}{:}\vec{B}\}\ C\ \mathsf{type} \\ \Theta; \Gamma \vdash s : A \qquad \Theta; \Gamma \vdash t : A \qquad \Theta; \Gamma \vdash s \equiv t : A\end{array}}{\Theta; \Gamma \vdash \{\vec{y}{:}\vec{B}[s/x]\}\ C[s/x] \equiv C[t/x]}$$

TT-Subst-EqTm
$$\frac{\begin{array}{c}\Theta; \Gamma \vdash \{x{:}A\}\{\vec{y}{:}\vec{B}\}\ u : C \\ \Theta; \Gamma \vdash s : A \qquad \Theta; \Gamma \vdash t : A \qquad \Theta; \Gamma \vdash s \equiv t : A\end{array}}{\Theta; \Gamma \vdash \{\vec{y}{:}\vec{B}[s/x]\}\ u[s/x] \equiv u[t/x] : C[s/x]}$$

TT-Conv-Abstr
$$\frac{\Theta; \Gamma \vdash \{x{:}A\}\ \mathcal{J} \qquad \Theta; \Gamma \vdash B\ \mathsf{type} \qquad \Theta; \Gamma \vdash A \equiv B}{\Theta; \Gamma \vdash \{x{:}B\}\ \mathcal{J}}$$

Figure 2.8: Admissible substitution rules

Theorem 2.2.8 (Admissibility of substitution)**.** *In a raw type theory, the closure rules from Fig. 2.8 are admissible.*

Proof. We already established admissibility of TT-Subst, TT-Bdry-Subst, and TT-Conv-Abstr in Lemma 2.2.6. Both TT-Subst-EqTy and TT-Subst-EqTm are seen to be admissible the same way: invert the abstraction and apply Lemma 2.2.7 to derive the desired conclusion. □

We provide two more lemmas that allow us to combine substitutions and judgmental equalities more flexibly.

Lemma 2.2.9. *Suppose a raw type theory derives*

$$\Theta; \Gamma \vdash s : A, \qquad \Theta; \Gamma \vdash t : A, \quad \textit{and} \quad \Theta; \Gamma \vdash s \equiv t : A.$$

1. *If it derives*

$$\Theta; \Gamma \vdash \{x{:}A\}\{\vec{y}{:}\vec{B}\}\ C \equiv D \quad \textit{and} \quad \Theta; \Gamma \vdash \{x{:}A\}\{\vec{y}{:}\vec{B}\}\ D\ \mathsf{type}$$

then it derives $\Theta; \Gamma \vdash \{\vec{y}{:}\vec{B}[s/x]\}\ C[s/x] \equiv D[t/x]$.

2. *If it derives*

$$\Theta; \Gamma \vdash \{x{:}A\}\{\vec{y}{:}\vec{B}\}\ u \equiv v : C \quad \textit{and} \quad \Theta; \Gamma \vdash \{x{:}A\}\{\vec{y}{:}\vec{B}\}\ v : C$$

then it derives $\Theta; \Gamma \vdash \{\vec{y}{:}\vec{B}[s/x]\}\ u[s/x] \equiv v[t/x] : C[s/x]$.

Proof. We spell out the proof of the first claim only. By substituting s for x in the first assumption we obtain

$$\Theta; \Gamma \vdash \{\vec{y}{:}\vec{B}[s/x]\}\ C[s/x] \equiv D[s/x],$$

and by applying TT-Subst-EqTy to the second assumption

$$\Theta; \Gamma \vdash \{\vec{y}{:}\vec{B}[s/x]\}\ D[s/x] \equiv D[t/x].$$

These two may be combined to give the desired judgement by unpacking the abstraction, applying transitivity, and packing up the abstraction. □

Lemma 2.2.10. *Suppose a raw type theory derives* $\vdash \Theta$ mctx *and, for* $i = 1, \ldots, n$*,*

$$\Theta; \Gamma \vdash s_i : A_i[\vec{s}_{(i)}/\vec{x}_{(i)}]$$
$$\Theta; \Gamma \vdash t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}]$$
$$\Theta; \Gamma \vdash s_i \equiv t_i : A_i[\vec{s}_{(i)}/\vec{x}_{(i)}].$$

If it derives an object judgement $\Theta; \Gamma \vdash \{\vec{x}{:}\vec{A}\}\ \mathcal{B}\boxed{e}$ *then it derives*

$$\Theta; \Gamma \vdash (\mathcal{B}[\vec{s}/\vec{x}])\boxed{e[\vec{s}/\vec{x}] \equiv e[\vec{t}/\vec{x}]}.$$

Proof. First, by inversion on the derivation of $\Theta; \Gamma \vdash \{\vec{x}{:}\vec{A}\}\ \mathcal{B}\boxed{e}$ we see that, for $i = 1, \ldots, n$,

$$\Theta; \Gamma \vdash \{\vec{x}_{(i)}{:}\vec{A}_{(i)}\}\ A_i\ \mathsf{type}.$$

Next, we claim that, for all $j = 1, \ldots, i-1$,

$$\Theta; \Gamma \vdash \{x_j{:}A_j[\vec{s}_{(j)}/\vec{x}_{(j)}]\} \cdots \{x_{i-1}{:}A_{i-1}[\vec{s}_{(j)}/\vec{x}_{(j)}]\}$$
$$A_i[\vec{s}_{(j)}/\vec{x}_{(j)}] \equiv A_i[\vec{t}_{(j)}/\vec{x}_{(j)}].$$

Indeed, when $j = 1$ the statement reduces to reflexivity, while an application of Lemma 2.2.9 lets us pass from j to $j+1$. When $j = i$ we obtain

$$\Theta; \Gamma \vdash A_i[\vec{s}_{(i)}/\vec{x}_{(i)}] \equiv A_i[\vec{t}_{(i)}/\vec{x}_{(i)}],$$

and this can be used to show by conversion that $\Theta; \Gamma \vdash t_i : A_i[\vec{s}_{(i)}/\vec{x}_{(i)}]$. Now the goal can be derived by repeated applications of Lemma 2.2.9. □

2.2.1.2 Admissibility of instantiations

We next turn to admissibility of instantiations, i.e. preservation of derivability under instantiation of metavariables by heads of derivable judgements.

Definition 2.2.11. An instantiation $I = \langle \mathsf{M}_1 {\mapsto} e_1, \ldots, \mathsf{M}_n {\mapsto} e_n \rangle$ of a metavariable context $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ over $\Theta; \Gamma$ is ***derivable*** when $\Theta; \Gamma \vdash (I_{(k)*}\mathscr{B}_k)\boxed{e_k}$ is derivable for $k = 1, \ldots, n$.

Lemma 2.2.12. *In a raw type theory, let I be a derivable instantiation of Ξ over context $\Theta; \Gamma$. If $\Xi; \Gamma, \Delta \vdash \mathscr{J}$ is derivable then so is $\Theta; \Gamma, I_*\Delta \vdash I_*\mathscr{J}$, and similarly for boundaries.*

Proof. We proceed by structural induction on the derivation of $\Xi; \Gamma, \Delta \vdash \mathscr{J}$, only devoting attention to the metavariable and abstraction rules, as all the other cases are straightforward.

Case TT-Meta: Consider an application of a metavariable rule for M with $\Xi(\mathsf{M}) = (\{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathscr{b})$ and $I(\mathsf{M}) = \{\vec{x}\}e$:

$$\frac{\begin{array}{ll}\Xi; \Gamma, \Delta \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Xi; \Gamma, \Delta \vdash \mathscr{b}[\vec{t}/\vec{x}] & \end{array}}{\Xi; \Gamma, \Delta \vdash (\mathscr{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}(\vec{t})}}$$

We need to derive

$$\Theta; \Gamma, I_*\Delta \vdash ((I_*\mathscr{b})[I_*\vec{t}/\vec{x}])\boxed{e[I_*\vec{t}/\vec{x}]}. \tag{2.18}$$

By induction hypothesis, for each $j = 1, \ldots, m$,

$$\Theta; \Gamma, I_*\Delta \vdash I_*t_j : (I_*A_j)[I_*\vec{t}_{(j)}/\vec{x}_{(j)}],$$

while derivability of I at M and weakening by $I_*\Delta$ yield

$$\Theta; \Gamma, I_*\Delta \vdash \{\vec{x}{:}I_*\vec{A}\}\ (I_*\mathscr{b})\boxed{e}. \tag{2.19}$$

We now derive (2.18) by repeatedly using TT-Subst to substitute I_*t_i's for x_i's in (2.19).

Case TT-Meta-Congr: Consider an application of a metavariable congruence rule for M with $\Xi(\mathsf{M}) = (\{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathscr{b})$ and $I(\mathsf{M}) = \{\vec{x}\}e$:

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

We need to derive

$$\Theta; \Gamma, I_* \Delta \vdash ((I_* \mathscr{b})[I_* \vec{s}/x]) \boxed{e[I_* \vec{s}/\vec{x}] \equiv e[I_* \vec{t}/\vec{x}]}.$$

Derivability of I yields

$$\Theta; \Gamma, I_* \Delta \vdash \{\vec{x}{:}I_* \vec{A}\}\ (I_* \mathscr{b}) \boxed{e}. \tag{2.20}$$

We may apply Lemma 2.2.10 to (2.20) with terms $I_* \vec{s}$ and $I_* \vec{t}$. The preconditions of the lemma are met by the induction hypotheses for the premises.

Case TT-ABSTR: Suppose the derivation ends with an abstraction

$$\frac{\Xi; \Gamma, \Delta \vdash A \ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma, \Delta| \qquad \Xi; \Gamma, \Delta, \mathsf{a}{:}A \vdash \mathcal{J}[\mathsf{a}/x]}{\Xi; \Gamma, \Delta \vdash \{x{:}A\}\ \mathcal{J}}$$

The induction hypotheses for the premises state

$$\Theta; \Gamma, I_* \Delta \vdash I_* A \ \mathsf{type} \qquad \text{and} \qquad \Theta; \Gamma, I_* \Delta, \mathsf{a}{:}I_* A \vdash I_*(\mathcal{J}[\mathsf{a}/x]).$$

Because $I_*(\mathcal{J}[\mathsf{a}/x]) = (I_* \mathcal{J})[\mathsf{a}/x]$ we may abstract a to derive

$$\Theta; \Gamma, I_* \Delta \vdash \{x{:}I_* A\}\ I_* \mathcal{J}.$$

□

Theorem 2.2.13 (Admissibility of instantiation)**.** *In a raw type theory, let I be a derivable instantiation of Ξ over context $\Theta; \Gamma$. If $\Xi; \Gamma \vdash \mathcal{J}$ is derivable then so is $\Theta; \Gamma \vdash I_* \mathcal{J}$, and similarly for boundaries.*

Proof. Apply Lemma 2.2.12 with empty Δ. □

We next show that, under favorable conditions, instantiating by judgementally equal instantiations leads to judgemental equality. To make the claim precise, define the notation $(I \equiv J)_* \mathcal{J}$ by

$$\begin{aligned}
(I \equiv J)_*(A \ \mathsf{type}) &= (I_* A \equiv J_* A \ \mathsf{by} \ \star),\\
(I \equiv J)_*(t : A) &= (I_* t \equiv J_* t : I_* A \ \mathsf{by} \ \star),\\
(I \equiv J)_*(\{x{:}A\}\ \mathcal{J}) &= (\{x{:}I_* A\}\ (I \equiv J)_* \mathcal{J})
\end{aligned}$$

and say that instantiations

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

of $\Xi = [\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n]$ over $\Theta; \Gamma$ are ***judgementally equal*** when, for $k = 1, \ldots, n$, if $\mathcal{B}_k$ is an object boundary then $\Theta; \Gamma \vdash (I_{(k)*} \mathcal{B}_k) \boxed{e_k \equiv f_k}$ is derivable.

Lemma 2.2.14. *In a raw type theory, consider derivable instantiations I and J of $\Xi = [\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n]$ over $\Theta; \Gamma$ which are judgementally equal. Suppose*

that $\vdash \Xi$ mctx *and* $\Theta \vdash \Gamma$ vctx, *and that* $\Theta; \Gamma, \Delta \vdash (I_{(i)*}\mathcal{B}_i)\boxed{J(\mathsf{M}_i)}$ *is derivable for* $i = 1, \ldots, n$, *and additionally that, for all* $\mathsf{a} \in |\Delta|$ *with* $\Delta(\mathsf{a}) = A$, *so are*

$$\begin{aligned} &\Theta; \Gamma, I_*\Delta \vdash I_*A \text{ type}, \\ &\Theta; \Gamma, I_*\Delta \vdash J_*A \text{ type}, \\ &\Theta; \Gamma, I_*\Delta \vdash I_*A \equiv J_*A \end{aligned}$$

If $\Xi; \Gamma, \Delta \vdash \mathcal{J}$ *is derivable then so are*

$$\Theta; \Gamma, I_*\Delta \vdash I_*\mathcal{J}, \tag{2.21}$$

$$\Theta; \Gamma, I_*\Delta \vdash J_*\mathcal{J}, \tag{2.22}$$

$$\Theta; \Gamma, I_*\Delta \vdash (I \equiv J)_*\mathcal{J} \quad \textit{if } \mathcal{J} \textit{ is an object judgement.} \tag{2.23}$$

Proof. Note that (2.21) already follows from Theorem 2.2.13, so we do not bother to reprove it, but we include the statement because we use it repeatedly. We proceed by structural induction on the derivations of $\vdash \Xi$ mctx and $\Xi; \Gamma, \Delta \vdash \mathcal{J}$.

Case TT-Var: Consider a derivation ending with the variable rule

$$\frac{}{\Xi; \Gamma, \Delta \vdash \mathsf{a}_i : A_i.}$$

We derive (2.22) by the variable rule, and when $\mathsf{a}_i \in |\Delta|$ a subsequent conversion along $\Theta; \Gamma, I_*\Delta \vdash I_*A_i \equiv J_*A_i$. The judgement (2.23) holds by TT-EqTm-Refl.

Case TT-Abstr: Consider a derivation ending with an abstraction

$$\frac{\Xi; \Gamma, \Delta \vdash B \text{ type} \qquad \mathsf{b} \notin |\Gamma, \Delta| \qquad \Xi; \Gamma, \Delta, \mathsf{b}{:}B \vdash \mathcal{J}[\mathsf{b}/y]}{\Xi; \Gamma, \Delta \vdash \{y{:}B\}\ \mathcal{J}}$$

The induction hypothesis for the first premise yields

$$\Theta; \Gamma, I_*\Delta \vdash I_*B \text{ type}, \tag{2.24}$$

$$\Theta; \Gamma, I_*\Delta \vdash J_*B \text{ type}, \tag{2.25}$$

$$\Theta; \Gamma, I_*\Delta \vdash I_*B \equiv J_*B. \tag{2.26}$$

The extended variable context $\Gamma, \Delta, \mathsf{b}{:}B$ satisfies the preconditions of the induction hypotheses for the second premise, therefore

$$\Theta; \Gamma, I_*\Delta, \mathsf{b}{:}I_*B \vdash (I_*\mathcal{J})[\mathsf{b}/y], \tag{2.27}$$

$$\Theta; \Gamma, I_*\Delta, \mathsf{b}{:}I_*B \vdash (J_*\mathcal{J})[\mathsf{b}/y], \tag{2.28}$$

$$\Theta; \Gamma, I_*\Delta, \mathsf{b}{:}I_*B \vdash ((I \equiv J)_*\mathcal{J})[\mathsf{b}/y], \tag{2.29}$$

where (2.29) is present only when $\mathcal{J}$ is an object judgement. Now (2.23) follows by abstraction from (2.24) and (2.29). To derive (2.22), we first abstract (2.28) to get

$$\Theta; \Gamma, I_*\Delta \vdash \{y{:}I_*B\}\ J_*\mathcal{J}$$

and then apply TT-CONV-ABSTR to convert it along (2.26) to derive the desired

$$\Theta; \Gamma, I_* \Delta \vdash \{y{:}J_* B\}\ J_* \mathcal{J}.$$

Case of a specific rule: Consider a specific rule

$$R = (\mathsf{N}_1{:}\mathcal{B}'_1, \ldots, \mathsf{N}_m{:}\mathcal{B}'_m \Longrightarrow \mathscr{b}\boxed{e})$$

and an instantiation $K = \langle \mathsf{N}_1{\mapsto} g_1, \ldots, \mathsf{N}_m{\mapsto} g_m \rangle$. Suppose the derivation ends with the instantiation $K_* R$:

$$\frac{\Xi; \Gamma, \Delta \vdash (K_{(i)*}\mathcal{B}'_i)\boxed{g_i} \quad \text{for } i = 1, \ldots, m \qquad \Xi; \Gamma, \Delta \vdash K_* \mathscr{b}}{\Xi; \Gamma, \Delta \vdash K_*(\mathscr{b}\boxed{e})} \tag{2.30}$$

We derive (2.22) by $(J_* K)_* R$ where $J_* K = \langle \mathsf{N}_1{\mapsto} J_* g_1, \ldots, \mathsf{N}_m{\mapsto} J_* g_m \rangle$. The resulting premises for $i = 1, \ldots, m$ are precisely the induction hypotheses (2.22) for the premises of (2.30). The last premise, $\Theta; \Gamma, I_* \Delta \vdash (J_* K)_* \mathscr{b}$, follows by case analysis of $\mathscr{b}$ and the same induction hypothesis (2.22). To establish (2.23), we must derive

$$\Theta; \Gamma, I_* \Delta \vdash ((I_* K)_* \mathscr{b})\boxed{(I_* K)_* e \equiv (J_* K)_* e}.$$

We do so by an application of the congruence rule associated with R, instantiated with $I_* K$ and $J_* K$. The resulting closure rule has four sets of premises, all of which are derivable:

- both copies of premises of R are derivable because they are the induction hypotheses (2.21) and (2.22) for the premises of (2.30),
- the additional equational premises are derivable because they are the induction hypotheses (2.23) for the premises of (2.30).

Case of a congruence rule: Similar to the case of a specific rule. Given a congruence rule with instantiations L and K, (2.22) follows from the same congruence rule with instantiations $J_* L$ and $J_* K$. The premises hold by induction hypothesis (2.22).

Case TT-META: Consider a derivation ending with an application of the metavariable rule for M_i, where $\vec{x} = (x_1, \ldots, x_m)$, $\vec{t} = (t_1, \ldots, t_m)$, $J(\mathsf{M}_i) = \{\vec{x}\}e$, and $\mathcal{B}_i = \{\vec{x}{:}\vec{A}\}\ \mathscr{b}$,

$$\frac{\Xi; \Gamma, \Delta \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m \qquad \Xi; \Gamma, \Delta \vdash \mathscr{b}[\vec{t}/\vec{x}]}{\Xi; \Gamma, \Delta \vdash (\mathscr{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_i(\vec{t})}} \tag{2.31}$$

Because J is derivable we know that $\Theta; \Gamma \vdash \{\vec{x}{:}J_* \vec{A}\}\ (J_* \mathscr{b})\boxed{e}$. For (2.22), we derive

$$\Theta; \Gamma, I_* \Delta \vdash ((J_* \mathscr{b})[J_* \vec{t}/\vec{x}])\boxed{e[J_* \vec{t}/\vec{x}]}$$

by substituting $J_*\vec{t}$ for $\vec{x}$ by repeated applications of TT-Subst, which generate premises, for $j = 1, \ldots, m$,

$$\Theta; \Gamma, I_*\Delta \vdash J_* t_j : (J_* A_j)[J_*\vec{t}_{(j)}/\vec{x}_{(j)}].$$

These are precisely the induction hypotheses for the premises of (2.31).

It remains to show (2.23). Writing $I(\mathsf{M}_i)$ as $\{x\}e'$, we must establish

$$\Theta; \Gamma, I_*\Delta \vdash ((I_*\mathscr{b})[I_*\vec{t}/\vec{x}])\boxed{e'[I_*\vec{t}/\vec{x}] \equiv e[J_*\vec{t}/\vec{x}]}.$$

Because I and J are judgementally equal, we know that

$$\Theta; \Gamma \vdash \{\vec{x} : I_*\vec{A}\}\ (I_*\mathscr{b})\boxed{e' \equiv e}.$$

By substituting $I_*\vec{t}$ for $\vec{x}$ by repeated use of TT-Subst, we derive

$$\Theta; \Gamma, I_*\Delta \vdash ((I_*\mathscr{b})[I_*\vec{t}/\vec{x}])\boxed{e'[I_*\vec{t}/\vec{x}] \equiv e[I_*\vec{t}/\vec{x}]}, \tag{2.32}$$

where the substitutions generate obligations, for $j = 1, \ldots, m$,

$$\Theta; \Gamma, I_*\Delta \vdash I_* t_j : (I_* A_j)[I_*\vec{t}_{(j)}/\vec{x}_{(j)}].$$

These are precisely the induction hypotheses for the term premises of (2.31). By transitivity it suffices to derive

$$\Theta; \Gamma, I_*\Delta \vdash ((I_*\mathscr{b})[I_*\vec{t}/\vec{x}])\boxed{e[I_*\vec{t}/\vec{x}] \equiv e[J_*\vec{t}/\vec{x}]}. \tag{2.33}$$

The induction hypotheses for the premises of (2.31) for $j = 1, \ldots, m$ are

$$\Theta; \Gamma, I_*\Delta \vdash I_* t_j : (I_* A_j)[I_*\vec{t}_{(j)}/\vec{x}_{(j)}] \tag{2.34}$$

$$\Theta; \Gamma, I_*\Delta \vdash J_* t_j : (J_* A_j)[J_*\vec{t}_{(j)}/\vec{x}_{(j)}] \tag{2.35}$$

$$\Theta; \Gamma, I_*\Delta \vdash I_* t_j \equiv J_* t_j : (I_* A_j)[I_*\vec{t}_{(j)}/\vec{x}_{(j)}]. \tag{2.36}$$

We would like to apply Lemma 2.2.10 to these judgements to derive (2.33), but the type of the terms $J_* t_j$ in (2.35) does not match the type of the corresponding terms $I_* t_j$. We rectify the situation by successively deriving the equality of the types involved and converting, as follows.

By assumption $\vdash \Xi\ \mathsf{mctx}$ holds and hence $\Xi_{(i)}; [\,] \vdash \{x_1{:}A_1\} \cdots \{x_{j-1}{:}A_{j-1}\}\ A_j\ \mathsf{type}$ for $j = 1, \ldots, m$. Note that the preceding judgement is derivable in a smaller metavariable context, and we can thus appeal to the induction hypothesis to derive

$$\Theta; \Gamma, I_*\Delta \vdash \{x_1{:}I_* A_1\} \cdots \{x_{j-1}{:}I_* A_{j-1}\}\ I_* A_j \equiv J_* A_j.$$

We apply Lemma 2.2.10 together with (2.34,2.35,2.36) to obtain

$$\Theta; \Gamma, I_*\Delta \vdash (I_* A_j)[I_*\vec{t}_{(j)}/\vec{x}_{(j)}] \equiv (J_* A_j)[J_*\vec{t}_{(j)}/\vec{x}_{(j)}].$$

We now appeal to TT-Conv-Tm to derive

$$\Theta; \Gamma, I_*\Delta \vdash J_* t_j : (I_* A_j)[I_*\vec{t}_{(j)}/\vec{x}_{(j)}]. \tag{2.37}$$

Finally we derive (2.33) by applying Lemma 2.2.10 to (2.34,2.37,2.36) and to the judgement $\Theta;\Gamma \vdash \{\vec{x}{:}I_*\vec{A}\}(I_*\mathscr{b})\boxed{e}$, which equals $\Theta;\Gamma \vdash (I_*\mathscr{B}_i)\boxed{J(\mathsf{M}_i)}$ and so is derivable by assumption.

Case TT-Meta-Congr: Consider a derivation ending with an application of the congruence rule for M_i, where $\vec{x} = (x_1, \ldots, x_m)$, $\vec{s} = (s_1, \ldots, s_m)$, $\vec{t} = (t_1, \ldots, t_m)$, $J(\mathsf{M}_i) = \{\vec{x}\}e$, and $\mathscr{B}_i = \{\vec{x}{:}\vec{A}\}\ \mathscr{b}$,

$$\frac{\begin{array}{ll} \Xi;\Gamma,\Delta \vdash s_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\ldots,m \\ \Xi;\Gamma,\Delta \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\ldots,m \\ \Xi;\Gamma,\Delta \vdash s_j \equiv t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\ldots,m \\ \Xi;\Gamma,\Delta \vdash C[\vec{s}/\vec{x}] \equiv C[\vec{t}/\vec{x}] & \text{if } \mathscr{b} = (\Box : C) \end{array}}{\Xi;\Gamma,\Delta \vdash (\mathscr{b}[\vec{s}/\vec{x}])\boxed{\mathsf{M}_i(\vec{s}) \equiv \mathsf{M}_i(\vec{t})}} \tag{2.38}$$

Because J is derivable we know that $\Theta;\Gamma \vdash \{\vec{x}{:}J_*\vec{A}\}\ (J_*\mathscr{b})\boxed{e}$, therefore by weakening also

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

The desired judgement

$$\Theta;\Gamma, I_*\Delta \vdash ((J_*\mathscr{b})[J_*\vec{s}/\vec{x}])\boxed{e[J_*\vec{s}/\vec{x}] \equiv e[J_*\vec{t}/\vec{x}]}$$

may be derived by repeated applications of TT-Subst-EqTm, provided that, for $j = 1, \ldots, m$,

$$\begin{aligned} &\Theta;\Gamma, I_*\Delta \vdash J_*s_j : (J_*A_j)[J_*\vec{s}_{(j)}/\vec{x}_{(j)}], \\ &\Theta;\Gamma, I_*\Delta \vdash J_*t_j : (J_*A_j)[J_*\vec{t}_{(j)}/\vec{x}_{(j)}], \\ &\Theta;\Gamma, I_*\Delta \vdash J_*s_j \equiv J_*t_j : (J_*A_j)[J_*\vec{s}_{(j)}/\vec{x}_{(j)}]. \end{aligned}$$

These are precisely induction hypotheses for (2.38).

Cases TT-EqTy-Refl, TT-EqTy-Sym, TT-EqTy-Trans, TT-EqTm-Refl, TT-EqTm-Sym, TT-EqTm-Trans, TT-Conv-Tm, *and* TT-Conv-EqTm : The remaining cases are all equality rules. Each is established by an appeal to the induction hypotheses for the premises, followed by an application of the same rule. □

Lemma 2.2.14 imposes conditions on the instantiations and the context which can be reduced to the more familiar assumption of well-typedness of the context, using Lemma 2.2.14 itself, as follows.

Lemma 2.2.15. *In a raw type theory, consider* $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ *such that* $\vdash \Xi\ \mathsf{mctx}$*, and derivable instantiations*

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

of Ξ *over* $\Theta;\Gamma$ *which are judgementally equal. Suppose further that* $\Theta \vdash \Gamma\ \mathsf{vctx}$ *and* $\Theta;\Gamma \vdash (I_{(i)*}\mathcal{B}_i)\boxed{f_i}$ *for* $i = 1,\ldots,n$*. If* $\Theta \vdash (\Gamma,\Delta)\ \mathsf{vctx}$*, then for all* $\mathsf{a} \in |\Delta|$ *with* $\Delta(\mathsf{a}) = A$*:*

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

Proof. We proceed by induction on the length of Δ. The base case is trivial. For the induction step, suppose $\Theta \vdash (\Gamma,\Delta,\mathsf{b}{:}B)\ \mathsf{vctx}$. For $\mathsf{a} \in |\Delta|$ we apply the induction hypothesis to Δ and weaken by $\mathsf{b}{:}I_*B$. To deal with b, we apply Lemma 2.2.14 to $\Theta;\Gamma,\Delta \vdash B\ \mathsf{type}$, which holds by inversion, and weaken by $\mathsf{b}{:}I_*B$ to derive the desired

$$\Theta;\Gamma, I_*\Delta, \mathsf{b}{:}I_*B \vdash I_*B\ \mathsf{type},$$
$$\Theta;\Gamma, I_*\Delta, \mathsf{b}{:}I_*B \vdash J_*B\ \mathsf{type},$$
$$\Theta;\Gamma, I_*\Delta, \mathsf{b}{:}I_*B \vdash I_*B \equiv J_*B.$$ ◻

Lemma 2.2.16. *In a raw type theory, consider* $\Xi = [\mathsf{M}_1{:}\mathcal{B}_1,\ldots,\mathsf{M}_n{:}\mathcal{B}_n]$ *such that* $\vdash \Xi\ \mathsf{mctx}$*, and derivable instantiations*

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

of Ξ *over* $\Theta;\Gamma$ *which are judgementally equal. Suppose that* $\Theta \vdash \Gamma\ \mathsf{vctx}$*. Then* $\Theta;\Gamma \vdash (I_{(i)*}\mathcal{B}_i)\boxed{f_i}$ *is derivable for* $i = 1,\ldots,n$*.*

Proof. We proceed by induction on n. The base case is trivial. To prove the induction step for $n > 0$, suppose the statement holds for $\Xi_{(n)}$, $I_{(n)}$ and $J_{(n)}$, and that $\mathcal{B}_n = \{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\ \mathscr{b}$. By inversion on $\vdash \Xi\ \mathsf{mctx}$ and weakening we derive $\Xi_{(n)};\Gamma \vdash \mathcal{B}_n$. Then by inverting the abstractions of $\mathcal{B}_n$ we obtain variables $\vec{a} = (\mathsf{a}_1,\ldots,\mathsf{a}_m)$ such that, with $A'_i = A_i[\vec{a}_{(i)}/\vec{x}_{(i)}]$ and $\Delta = [\mathsf{a}_1{:}A'_1,\ldots,\mathsf{a}_m{:}A'_m]$,

$$\Xi_{(n)} \vdash (\Gamma,\Delta)\ \mathsf{vctx}, \qquad \text{and} \qquad \Xi_{(n)};\Gamma,\Delta \vdash \mathscr{b}[\vec{a}/\vec{x}].$$

We apply Lemma 2.2.15 to $\Xi_{(n)}$, $I_{(n)}$, $J_{(n)}$, and Δ to derive, for $i = 1,\ldots,m$,

$$\Theta;\Gamma, I_*\Delta \vdash I_*A'_i\ \mathsf{type},$$
$$\Theta;\Gamma, I_*\Delta \vdash J_*A'_i\ \mathsf{type},$$
$$\Theta;\Gamma, I_*\Delta \vdash I_*A'_i \equiv J_*A'_i,$$
$$\Theta;\Gamma, I_*\Delta \vdash \mathsf{a}_i : J_*A'_i. \tag{2.39}$$

where (2.39) follows by conversion from the judgement above it. Next, we use (2.39) to substitute a_i for x_i in $\Theta;\Gamma, I_*\Delta \vdash \{\vec{x}{:}J_*\vec{A}\}\ (J_*\mathscr{b})\boxed{f_n}$, which results in

$$\Theta;\Gamma, I_*\Delta \vdash ((J_*\mathscr{b})[\vec{a}/\vec{x}])\boxed{f_n[\vec{a}/\vec{x}]}. \tag{2.40}$$

If we can reduce (2.40) to

$$\Theta; \Gamma, I_* \Delta \vdash ((I_* \mathcal{b})[\vec{a}/\vec{x}])\boxed{f_n[\vec{a}/\vec{x}]}, \tag{2.41}$$

we will be able to derive the desired judgement

$$\Theta; \Gamma, I_* \Delta \vdash \{\vec{x}{:}I_* \vec{A}\}\ (I_* \mathcal{b})\boxed{f_n}$$

by abstracting $\mathsf{a}_1, \dots, \mathsf{a}_n$ in (2.41). There are four cases, depending on what $\mathcal{b}$ is.
Case $\mathcal{b} = (\Box\ \mathsf{type})$: (2.40) and (2.41) are the same.
Case $\mathcal{b} = (\Box : B)$: We convert (2.40) along

$$\Theta; \Gamma, I_* \Delta \vdash (J_* B)[\vec{a}/\vec{x}] \equiv (I_* B)[\vec{a}/\vec{x}],$$

which holds by Lemma 2.2.14 applied to $\Xi_{(n)}; \Gamma, \Delta \vdash B[\vec{a}/\vec{x}]\ \mathsf{type}$ with $\Xi_{(n)}$, $I_{(n)}$, and $J_{(n)}$.
Case $\mathcal{b} = (B \equiv C\ \mathsf{by}\ \Box)$: Here (2.40) and (2.41) are respectively

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

The latter follows from the former if we can also derive

$$\Theta; \Gamma, I_* \Delta \vdash I_* B \equiv J_* B, \quad \text{and} \quad \Theta; \Gamma, I_* \Delta \vdash I_* C \equiv J_* C. \tag{2.42}$$

We invert $\Xi_{(n)}; \Gamma, \Delta \vdash B \equiv C\ \mathsf{by}\ \Box$ to derive

$$\Xi_{(n)}; \Gamma, \Delta \vdash B\ \mathsf{type} \qquad \text{and} \qquad \Xi_{(n)}; \Gamma, \Delta \vdash C\ \mathsf{type}. \tag{2.43}$$

When we apply Lemma 2.2.14 to (2.43) it gives us (2.42).
Case $\mathcal{b} = (s \equiv t : B\ \mathsf{by}\ \Box)$: Here (2.40) and (2.41) are respecetively

$$\Theta; \Gamma, I_* \Delta \vdash J_* s \equiv J_* t : J_* B \quad \text{and} \quad \Theta; \Gamma, I_* \Delta \vdash I_* s \equiv I_* t : I_* B,$$

The latter follows from the former if we can also derive

$$\begin{aligned} &\Theta; \Gamma, I_* \Delta \vdash I_* B \equiv J_* B, \\ &\Theta; \Gamma, I_* \Delta \vdash I_* s \equiv J_* s : I_* B, \\ &\Theta; \Gamma, I_* \Delta \vdash I_* t \equiv J_* t : I_* B. \end{aligned} \tag{2.44}$$

We invert $\Xi_{(n)}; \Gamma, \Delta \vdash s \equiv t : B\ \mathsf{by}\ \Box$ to derive

$$\Xi_{(n)}; \Gamma, \Delta \vdash B\ \mathsf{type}, \quad \Xi_{(n)}; \Gamma, \Delta \vdash s : B \quad \text{and} \quad \Xi_{(n)}; \Gamma, \Delta \vdash t : B. \tag{2.45}$$

When we apply Lemma 2.2.14 to (2.45) it gives us (2.44). □

Finally, the lemmas can be assembled into an admissibility theorem about judgementally equal derivable instantiations.

Theorem 2.2.17 (Admissibility of instantiation equality)**.** *In a raw type theory, consider derivable instantiations I and J of Ξ over $\Theta; \Gamma$ which are judgementally equal. Suppose that $\vdash \Xi$ mctx and $\Theta \vdash \Gamma$ vctx. If an object judgement $\Xi; \Gamma \vdash \mathcal{J}$ is derivable then so is $\Theta; \Gamma \vdash (I \equiv J)_* \mathcal{J}$.*

Proof. Lemma 2.2.14 applies with empty Δ because the additional precondition for I and J is guaranteed by Lemma 2.2.16. □

2.2.1.3 Presuppositivity of raw theories

Our last metatheorem about raw type theories shows that whenever a judgement is derivable, so are its presuppositions, i.e. its boundary is well-formed.

Theorem 2.2.18 (Presuppositivity). *If a raw type theory derives* $\vdash \Theta\ \mathsf{mctx}$, $\Theta \vdash \Gamma\ \mathsf{vctx}$, *and* $\Theta; \Gamma \vdash \mathscr{B}\boxed{e}$ *then it derives* $\Theta; \Gamma \vdash \mathscr{B}$.

Proof. We proceed by induction on the derivation of $\Theta; \Gamma \vdash \mathscr{B}\boxed{e}$.

Case TT-VAR: By Proposition 2.2.3.

Case TT-META: The presupposition $\Theta; \Gamma \vdash \mathscr{b}[\vec{t}/\vec{x}]$ is available as premise.

Case TT-META-CONGR: Consider a derivation ending with an application of the congruence rule for M whose boundary is $\Theta(\mathsf{M}) = (\{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathscr{b})$:

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

If $\mathscr{b} = (\Box\ \mathsf{type})$, the presupposition $\Theta; \Gamma \vdash \mathsf{M}(\vec{s}) \equiv \mathsf{M}(\vec{t})$ by $\Box$ follows directly by TT-BDRY-EQTY and two uses of TT-META. If $\mathscr{b} = (\Box : C)$, the presuppositions of $\vdash \mathsf{M}(\vec{s}) \equiv \mathsf{M}(\vec{t}) : C[\vec{s}/\vec{x}]$ by $\Box$ follow by TT-BDRY-EQTM:

1. $\Theta; \Gamma \vdash C[\vec{s}/\vec{x}]\ \mathsf{type}$ holds by substitution of $\vec{s}$ for $\vec{x}$ in $\Theta; \Gamma \vdash \{\vec{x}{:}\vec{A}\}\ C\ \mathsf{type}$ much like in the previous case,

2. $\Theta; \Gamma \vdash \mathsf{M}(\vec{s}) : C[\vec{s}/\vec{x}]$ holds by TT-META,

3. $\Theta; \Gamma \vdash \mathsf{M}(\vec{t}) : C[\vec{s}/\vec{x}]$ is derived from $\Theta; \Gamma \vdash \mathsf{M}(\vec{t}) : C[\vec{t}/\vec{x}]$ by conversion along $\Theta; \Gamma \vdash C[\vec{t}/\vec{x}] \equiv C[\vec{s}/\vec{x}]$, which holds by the last premise.

When applying TT-META above, the premise $\Theta; \Gamma \vdash \mathscr{b}[\vec{s}/\vec{x}]$ is required, and likewise for $\vec{t}$. We may derive it by applying Proposition 2.2.3 to $\vdash \Theta\ \mathsf{mctx}$ and substituting $\vec{s}$ for $\vec{x}$ with the help of TT-SUBST, and analogously for $\vec{t}$.

Case TT-ABSTR: Consider an abstraction

$$\frac{\Theta; \Gamma \vdash A\ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma| \qquad \Theta; \Gamma, \mathsf{a}{:}A \vdash \mathscr{J}[\mathsf{a}/x]}{\Theta; \Gamma \vdash \{x{:}A\}\ \mathscr{J}}$$

By induction hypothesis on the last premise, we obtain $\Theta; \Gamma, \mathsf{a}{:}A \vdash \mathscr{B}[\mathsf{a}/x]$ after which we apply TT-BDRY-ABSTR.

Case of a specific rule: The presupposition is available as premise.

Case of a congruence rule: Consider a congruence rulles associated with an object rule R and instantiated with I and J, as in Definition 2.1.13.

If R concludes with $\vdash A$ type, the presuppositions are $\Theta; \Gamma \vdash I_* A$ type and $\Theta; \Gamma \vdash J_* A$ type, which are derivable by $I_* R$ and $J_* R$, respectively.

If R concludes with $\vdash t : A$, the presuppositions are $\Theta; \Gamma \vdash I_* A$ type, $\Theta; \Gamma \vdash I_* t : I_* A$, and $\Theta; \Gamma \vdash J_* t : I_* A$. We derive the first one by applying the induction hypothesis to the premise $\Theta; \Gamma \vdash I_* B \equiv J_* B$, the second one by $I_* R$, and the third one by converting the second one along the aforementioned premise.

Cases TT-EqTy-Refl, TT-EqTy-Sym, TT-EqTy-Trans, TT-EqTm-Refl, TT-EqTm-Sym, TT-EqTm-Trans: These are all dispensed with by straightforward appeals to the induction hypotheses.

Case TT-Conv-Tm: Consider a term conversion

$$\frac{\Theta; \Gamma \vdash t : A \qquad \Theta; \Gamma \vdash A \equiv B}{\Theta; \Gamma \vdash t : B}$$

Then $\Theta; \Gamma \vdash B$ type holds by the induction hypothesis for the second premise.

Case TT-Conv-EqTm: Consider a term equality conversion

$$\frac{\Theta; \Gamma \vdash s \equiv t : A \qquad \Theta; \Gamma \vdash A \equiv B}{\Theta; \Gamma \vdash s \equiv t : B}$$

As in the previous case, the induction hypothesis for the second premise provides $\Theta; \Gamma \vdash B$ type. The induction hypothesis for the first premise yields

$$\Theta; \Gamma \vdash s : A \qquad \text{and} \qquad \Theta; \Gamma \vdash t : A$$

We may convert these to $\Theta; \Gamma \vdash s : B$ and $\Theta; \Gamma \vdash t : B$ using the second premise. □

2.2.2 Metatheorems about finitary theories

Several closure rules contain premises which at first sight seem extraneous, in particular the boundary premises in rule instantiations (Definition 2.1.8) and the last premise in a congruence rule (Definition 2.1.13). While these are needed for raw rules, they ought to be removable for finitary rules, which already have well-formed boundaries. We show that this is indeed the case by providing *economic* versions of the rules, which are admissible in finitary type theories. We also show that the metavariable rules (Definition 2.1.15) have economic versions that are valid in well-formed metavariable contexts.

Proposition 2.2.19. *[Economic version of Definition 2.1.8] Let R be the raw rule $\Xi \Longrightarrow \mathscr{b}\boxed{e}$ with $\Xi = [\mathsf{M}_1{:}\mathscr{B}_1, \ldots, \mathsf{M}_n{:}\mathscr{B}_n]$ such that $\Xi; [\,] \vdash \mathscr{b}$ is derivable, in particular R may be finitary. Then for any instantiation $I = [\mathsf{M}_1 {\mapsto} e_1, \ldots, \mathsf{M}_n {\mapsto} e_n]$ over $\Theta; \Gamma$, the following closure rule is admissible:*

TT-Specific-Eco

$$\frac{\Theta; \Gamma \vdash (I_{(i)*} \mathscr{B}_i)\boxed{e_i} \quad \textit{for } i = 1, \ldots, n}{\Theta; \Gamma \vdash I_*(\mathscr{b}\boxed{e})}$$

Proof. To apply $I_* R$, derive the missing premise $\Theta; \Gamma \vdash I_* \mathcal{B}$ via Theorem 2.2.13. □

Proposition 2.2.20 (Economic version of Definition 2.1.15). *If a raw type theory derives* $\vdash \Theta$ mctx *with* $\Theta(\mathsf{M}) = (\{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathcal{B})$*, the following closure rules are admissible:*

TT-META-ECO
$$\frac{\Theta; \Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m}{\Theta; \Gamma \vdash (\mathcal{B}[\vec{t}/\vec{x}])\boxed{\mathsf{M}(\vec{t})}}$$

TT-META-CONGR-ECO
$$\frac{\Theta; \Gamma \vdash s_j \equiv t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m}{\Theta; \Gamma \vdash (\mathcal{B}[\vec{s}/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}) \equiv \mathsf{M}_k(\vec{t})}}$$

Proof. To prove admissibility of TT-META-ECO, note that by Proposition 2.2.3 we have $\Theta; \Gamma \vdash \{\vec{x}{:}\vec{A}\}\ \mathcal{B}$, so we may derive $\Theta; \Gamma \vdash \mathcal{B}[\vec{t}/\vec{x}]$ by substituting $\vec{t}$ for $\vec{x}$ by repeated applications of TT-SUBST to the premises of TT-META-ECO. We can now apply TT-META.

Next, we address admissibility of TT-META-CONGR-ECO by deriving its conclusion with the aid of TT-META-CONGR. For this purpose we need to derive

$$\begin{array}{ll} \Theta; \Gamma \vdash s_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta; \Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta; \Gamma \vdash C[\vec{s}/\vec{x}] \equiv C[\vec{t}/\vec{x}] & \text{if } \mathcal{B} = (\Box : C) \end{array}$$

The first group follows by Theorem 2.2.18. The second is established by induction on j: by Proposition 2.2.3, $\Theta \vdash \{\vec{x}{:}\vec{A}\}\ \mathcal{B}$ holds, and thus $\Theta \vdash \{\vec{x}_{(j)}{:}\vec{A}_{(j)}\}\ A_j$ type. By applying Lemma 2.2.10, we obtain $\Theta; \Gamma \vdash A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] \equiv A_j[\vec{t}_{(j)}/\vec{x}_{(j)}]$ and we can convert $\Theta; \Gamma \vdash t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}]$ which holds again by Theorem 2.2.18. Finally, the last premise holds again by Lemma 2.2.10, this time applied to $\Theta \vdash \{\vec{x}{:}\vec{A}\}\ C$ type. □

2.2.3 Metatheorems about standard theories

We next investigate to what extent a derivation of a derivable judgement can be reconstructed from the judgement itself. Firstly, a term expression holds enough information to recover a candidate for its type.

Definition 2.2.21. Let 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:

$$\begin{array}{ll} \tau_{\Theta;\Gamma}(\mathsf{a}) = \Gamma(a), & \\ \tau_{\Theta;\Gamma}(\mathsf{M}(\vec{t})) = B[\vec{t}/\vec{x}] & \text{where } \Theta(\mathsf{M}) = (\{\vec{x}{:}\vec{A}\}\ \Box : B) \\ \tau_{\Theta;\Gamma}(\mathsf{S}(\vec{e})) = \langle \vec{\mathsf{M}} \mapsto \vec{e} \rangle_* B & \text{where the rule for } \mathsf{S} \text{ is } \vec{\mathsf{M}}{:}\vec{\mathscr{B}} \Longrightarrow \Box : B \end{array}$$

We prove an inversion principle that recovers the “stump” of a derivation of a derivable object judgement.

Theorem 2.2.22 (Inversion). *If a standard type theory derives an object 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 $\tau_{\Theta;\Gamma}(t) \neq A$*.*

Proof. We proceed by induction on the derivation $\Gamma; \Theta \vdash \mathcal{J}$. If the derivation concludes with TT-Var, TT-Meta, a symbol rule, or TT-Abstr, then it already has the desired form. The remaining case is a derivation D ending with a term conversion rule

$$\frac{\dfrac{D_1}{\Theta; \Gamma \vdash t : A} \qquad \dfrac{D_2}{\Theta; \Gamma \vdash A \equiv B}}{\Theta; \Gamma \vdash t : B}$$

By induction hypothesis we may invert D_1 and obtain a derivation D' of $\Theta; \Gamma \vdash t : A$ as in the statement of the theorem:

1. If D' ends with TT-Var, TT-Meta or a term symbol rule then $A = \tau_{\Theta;\Gamma}(t)$. Either $\tau_{\Theta;\Gamma}(t) = B$ and we use D', or $\tau_{\Theta;\Gamma}(t) \neq B$ and we use D.
2. If D' concludes with a term conversion

$$\frac{\dfrac{D_1'}{\Theta; \Gamma \vdash t : \tau_{\Theta;\Gamma}(t)} \qquad \dfrac{D_2'}{\Theta; \Gamma \vdash \tau_{\Theta;\Gamma}(t) \equiv A}}{\Theta; \Gamma \vdash t : A}$$

   there are again two cases. If $\tau_{\Theta;\Gamma}(t) = B$ we use D_1', otherwise we combine $\tau_{\Theta;\Gamma}(t) \equiv A$ and $A \equiv B$ by transitivity and conversion:

$$\frac{\dfrac{D_1'}{\Theta; \Gamma \vdash t : \tau_{\Theta;\Gamma}(t)} \qquad \dfrac{\dfrac{D_2'}{\Theta; \Gamma \vdash \tau_{\Theta;\Gamma}(t) \equiv A} \qquad \dfrac{D_2}{\Theta; \Gamma \vdash A \equiv B}}{\Theta; \Gamma \vdash \tau_{\Theta;\Gamma}(t) \equiv B}}{\Theta; \Gamma \vdash t : B}$$

□

We may keep applying the theorem to all the object premises of a stump to recover the proof-relevant part of the derivation. The remaining proof-irrelevant parts are the equational premises. The inversion theorem yields further desirable meta-theoretic properties of standard type theories.

Corollary 2.2.23. *If a standard type theory derives* $\Theta; \Gamma \vdash t : A$ *then it derives* $\Theta; \Gamma \vdash \tau_{\Theta;\Gamma}(t) \equiv A$.

Proof. By inversion, $\tau_{\Theta;\Gamma}(t) = A$ or we obtain a derivation of $\vdash \tau_{\Theta;\Gamma}(t) \equiv A$. □

Theorem 2.2.24 (Uniqueness of typing)**.** *For a standard type theory:*

1. *If* $\Theta; \Gamma \vdash t : A$ *and* $\Theta; \Gamma \vdash t : B$ *then* $\Theta; \Gamma \vdash A \equiv B$.
2. *If* $\vdash \Theta \ \mathsf{mctx}$ *and* $\Theta \vdash \Gamma \ \mathsf{vctx}$ *and* $\Theta; \Gamma \vdash s \equiv t : A$ *and* $\Theta; \Gamma \vdash s \equiv t : B$ *then* $\Theta; \Gamma \vdash A \equiv B$.

Proof. The first statement holds because A and B are both judgmentally equal to the natural type of t by Corollary 2.2.23. The second statement reduces to the first one because the presuppositions $\Theta; \Gamma \vdash t : A$ and $\Theta; \Gamma \vdash t : B$ are derivable by Theorem 2.2.18. □

Heron, The natural history of British birds, v.4 (1797).
Source: Biodiversity Heritage Library.

Chapter 3

Context-free type theories

In this chapter we give a second account of type theories, which is better suited for the forward-chaining style of proof development, characteristic of LCF-style theorem provers. These theories are *context-free* because the judgements have no explicit context. Instead, variables are always tagged with typing annotations. In Section 3.2 we establish metatheorems about context-free type theories, that we need to carry out the translation between the context and context-free variants.

Finally, in Section 3.3 we provide faithful translations between the two versions of type theories, thus showing that nothing is lost or gained in terms of expressivity by using one formalism instead of the other. This result opens the way to an implementation of finitary type theories viewed through the lens of context-free type theories in a proof assistant, as described in the next chapter.

3.1 Context-free finitary type theories

In the forward-chaining style, characteristic of LCF-style theorem provers, a judgement is not construed by reducing a goal to subgoals, but as a value of an abstract datatype, and built by applying an abstract datatype constructor to previously derived judgements. What should such a constructor do when its arguments have mismatching variable contexts? It can try to combine them if possible, or require that the user make sure ahead of time that they match. As was already noted by Geuvers et al. in the context of pure type systems (Geuvers et al. 2010), it is best to sidestep the whole issue by dispensing with contexts altogether. In the present section we give a second account of finitary type theories, this time without context and with free variables explicitly annotated with their types.

Our formulation of ***context-free finitary type theories*** is akin to the Γ_∞ formalism for pure type systems (Geuvers et al. 2010). We would like to replace judgements of the form "$\Theta; \Gamma \vdash \mathcal{J}$" with just "$\mathcal{J}$". In traditional accounts of logic, as well as in Γ_∞, this is accomplished by explicit type annotations of free variables: rather than having $\mathsf{a} : A$ in the variable context, each occurrence of a is annotated with its type as a^A.

We use the same idea, although we have to overcome several technical complications, of which the most challenging one is the lack of strengthening, which is the principle stating that if $\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash \mathcal{J}$ is derivable and a does not appear in Δ and $\mathcal{J}$, then $\Theta; \Gamma, \Delta \vdash \mathcal{J}$ is derivable. An example of a rule that breaks strengthening for finitary type theories is equality reflection from Example 2.1.7,

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A} \qquad \vdash \mathsf{p} : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A}}$$

Because the conclusion elides the metavariable p, it will not record the fact that a variable may have been used in the derivation of the fourth premise. Consequently, we cannot tell what variables ought to occur in the context just by looking at the judgement thesis. As it turns out, variables elided by derivations of equations are the only culprit, and the situation can be rectified by modifying equality judgements so that they carry additional information about usage of variables. In the present section we show how this is accomplished by revisiting the definition of type theories from Section 2.1 and making the appropriate modifications.

3.1.1 Raw syntax of context-free type theories

Apart from removing the variable context and annotating free variables with type expressions, we make three further modifications to the raw syntax: we remove metavariable contexts, and instead annotate metavariables with boundaries; we introduce assumption sets that keep track of variables used in equality derivations; and we introduce explicit conversions.

3.1.1.1 Free and bound variables

The ***bound variables*** $x, y, z, \ldots$ are as before, for example they could be de Bruijn indices, whereas the free variables are annotated explicitly with type expressions. More precisely, given a set of names $\mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots$ a ***free variable*** takes the form a^A where A is a type expression, cf. Section 3.1.1.3. Two such variables a^A and b^B are considered syntactically equal when the symbols a and b are the same *and* the type expressions A and B are syntactically equal. Thus it is quite possible to have variables a^A and a^B which are different even though A and B are judgmentally equal. In an implementation it may be a good idea to prevent such extravaganza by generating fresh symbols so that each one receives precisely one annotation.

Similarly, metavariables are tagged with boundaries, where again $\mathsf{M}^{\mathcal{B}}$ and $\mathsf{N}^{\mathcal{B}'}$ are considered equal when both the symbols M and N are equal and the boundaries $\mathcal{B}$ and $\mathcal{B}'$ are syntactically identical.

3.1.1.2 Arities and signatures

Arities of symbols and metavariables are as in Section 2.1.1.2, and so are signatures.

3.1.1.3 Raw expressions

The raw expressions of a context-free type theory are built over a signature Σ, as summarized in the top part of Fig. 3.1.

Type expression A, B	$::= \mathsf{S}(e_1, \ldots, e_n)$	application of a type symbol S
	$\mid \mathsf{M}^{\mathcal{B}}(t_1, \ldots, t_n)$	application of a type metavariable $\mathsf{M}^{\mathcal{B}}$
Term expression s, t	$::= \mathsf{a}^A$	free variable
	$\mid x$	bound variable
	$\mid \mathsf{S}(e_1, \ldots, e_n)$	application of a term symbol S
	$\mid \mathsf{M}^{\mathcal{B}}(t_1, \ldots, t_n)$	application of a term metavariable $\mathsf{M}^{\mathcal{B}}$
	$\mid \kappa(t, \alpha)$	conversion
Assumption set α, β	$::= \{\!\{ \ldots, a_i^A, \ldots, x_j, \ldots, \mathsf{M}_k^{\mathcal{B}}, \ldots \}\!\}$	
Argument e	$::= A$	type argument
	$\mid t$	term argument
	$\mid \alpha$	assumption set
	$\mid \{x\}e$	abstracted argument (x bound in e)
Judgement j	$::= A\ \mathsf{type}$	A is a type
	$\mid t : A$	t has type T
	$\mid A \equiv B\ \mathsf{by}\ \alpha$	A and B are equal types
	$\mid s \equiv t : A\ \mathsf{by}\ \alpha$	s and t are equal terms at A
Abstracted judgement $\mathcal{J}$	$::= j$	non-abstracted judgement
	$\mid \{x{:}A\}\ \mathcal{J}$	abstracted judgement (x bound in $\mathcal{J}$)
Boundary $\mathscr{b}$	$::= \Box\ \mathsf{type}$	a type
	$\mid \Box : A$	a term of type A
	$\mid A \equiv B\ \mathsf{by}\ \Box$	type equation boundary
	$\mid s \equiv t : B\ \mathsf{by}\ \Box$	term equation boundary
Abstracted boundary $\mathcal{B}$	$::= \mathscr{b}$	non-abstracted boundary
	$\mid \{x{:}A\}\ \mathcal{B}$	abstracted boundary (x bound in $\mathcal{B}$)

Figure 3.1: The raw syntax of context-free finitary type theories

A type expression is either a type symbol S applied to arguments $e_1, \ldots, e_n$, or a metavariable $\mathsf{M}^{\mathcal{B}}$ applied to term expressions $t_1, \ldots, t_n$.

The syntax of term expressions differs from the one in Fig. 2.1 in two ways. First, we annotate free variables with type expressions and metavariables with boundaries, as was already discussed, where it should be noted that in an annotation A of a^A or $\mathcal{B}$ of $\mathsf{M}^{\mathcal{B}}$ there may be further free and metavariables, which are also annotated, and so

on. We require that a boundary annotation $\mathcal{B}$ be closed, and that a type annotation A contain no "exposed" bound variables, i.e. A is syntactically valid on its own, without having to appear under an abstraction. Second, we introduce the ***conversion terms*** "$\kappa(t, \alpha)$", which will serve to record the variables used to derive the equality along which t has been converted.

The expressions of syntactic classes EqTy and EqTm are the ***assumption sets***, which are finite sets of free and bound variables, and metavariables. As we are already using the curly braces for abstraction, we write finite set comprehension as $\{\!\!\{ \cdots \}\!\!\}$. Assumption sets record the variables and metavariables that are used in a derivation of an equality judgement but may not appear in the boundary of the conclusion.

We ought to be a bit careful about occurrences of variables, since the free variables may occur in variable annotations, and the metavariables in boundary annotations. Figure 3.2, the context-free analogue of Fig. 2.2, shows the definitions of free, bound and metavariable occurrences. Note the difference between $\mathsf{fv}_0(e)$, which collects *only* the free variable occurrences not appearing in a type annotation, and $\mathsf{fv}(e)$ which collects them all. Bound variables need not be collected from annotations, as they cannot appear there.

The collection of all free, bound and metavariables occurring in an expression is its ***assumption set*** $\mathsf{asm}(e)$. Sometimes we write $\mathsf{asm}(e_1, \ldots, e_n)$ for the union $\bigcup_i \mathsf{asm}(e_i)$.

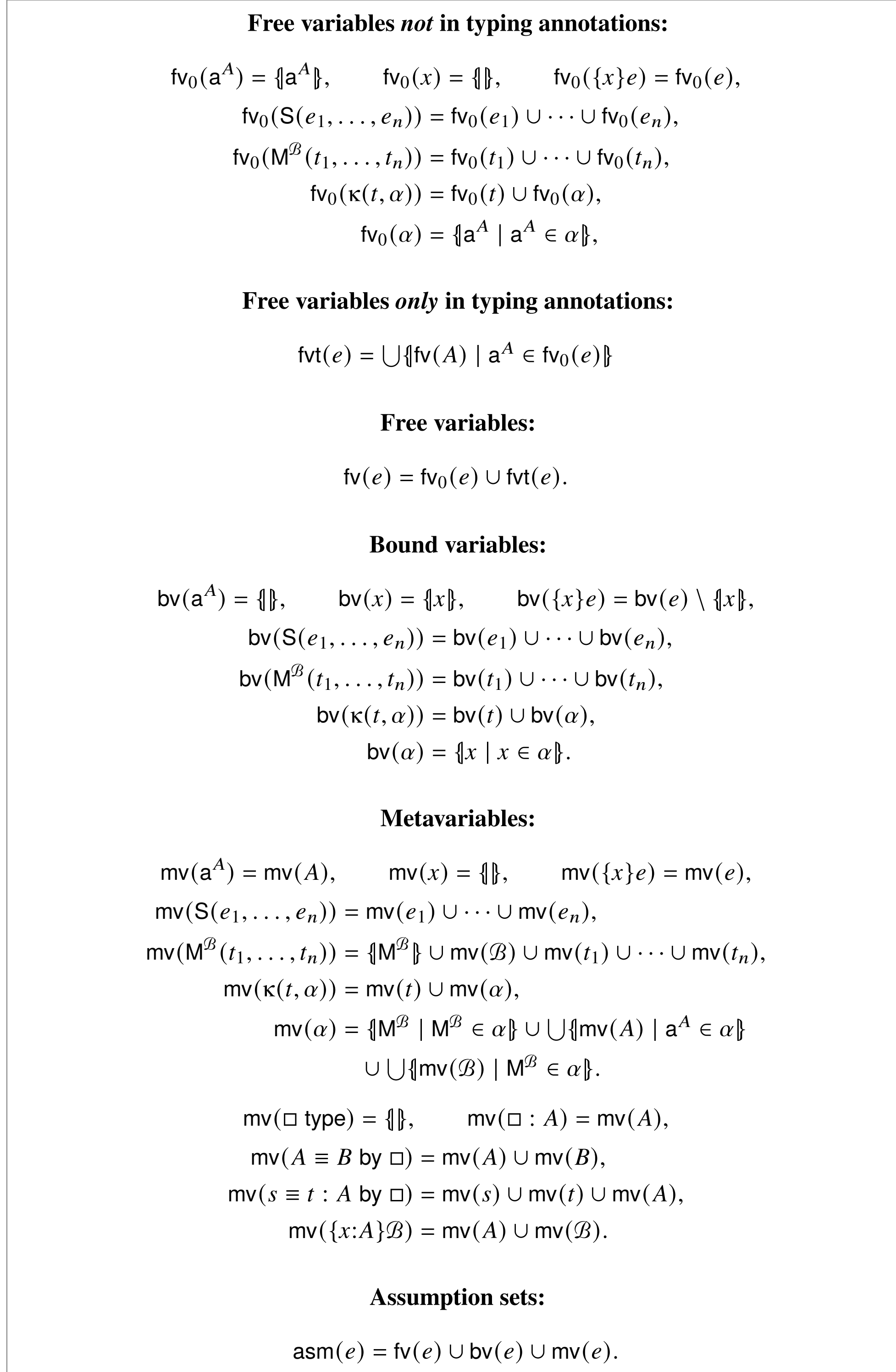
3.1.1.4 Substitution and syntactic equality

We must review substitution and syntactic equality, because they are affected by annotations, assumption sets, and conversion terms.

There are two kinds of substitutions. An ***abstraction*** $e[x/\mathsf{a}^A]$ transforms the free variable a^A in e to a bound variable x, whereas a ***substitution*** $e[s/x]$ replaces the bound variable x with the term s. These are shown in Fig. 3.3. Note that an abstraction $e[x/\mathsf{a}^A]$ is only valid when a^A does not appear in any type annotation in e, $\mathsf{a}^A \notin \mathsf{fvt}(e)$, because type annotations cannot refer to bound variables. Consequently, abstraction of several variables must be carried out in the reverse order of their dependencies. We abbreviate a series of abstractions $((e[x_1/\mathsf{a}_1^{A_1}]) \cdots)[x_n/\mathsf{a}_n^{A_n}]$ as $e[x_1/\mathsf{a}_1^{A_1}, \ldots, x_n/\mathsf{a}_n^{A_n}]$ or just $e[\vec{x}/\vec{\mathsf{a}}_n^{A_n}]$. Similarly, a series of substitutions $((e[s_1/x_1]) \cdots)[s_n/x_n]$ is written $e[s_1/x_1, \ldots, s_n/x_n]$ or just $e[\vec{s}/\vec{x}]$.

Syntactic equality is treated in a standard way, we only have to keep in mind the fact that symbols are considered syntactically equal if the bare symbols are equal *and* their annotations are equal. More interestingly, since conversion terms and assumption sets carry proof-irrelevant information, they should be ignored in certain situations. For this purpose, define the ***erasure*** $\lfloor e \rfloor$ to be the raw expression e with the assumption sets and conversion terms removed:

$$\lfloor \mathsf{a}^A \rfloor = \mathsf{a}^A, \quad \lfloor x \rfloor = x, \quad \lfloor \kappa(t, \alpha) \rfloor = \lfloor t \rfloor, \quad \lfloor \alpha \rfloor = \star, \quad \lfloor \{x\}e \rfloor = \{x\}\lfloor e \rfloor,$$
$$\lfloor \mathsf{S}(e_1, \ldots, e_n) \rfloor = \mathsf{S}(\lfloor e_1 \rfloor, \ldots, \lfloor e_n \rfloor), \qquad \lfloor \mathsf{M}^{\mathcal{B}}(t_1, \ldots, t_n) \rfloor = \mathsf{M}^{\mathcal{B}}(\lfloor t_1 \rfloor, \ldots, \lfloor t_n \rfloor).$$



Free variables *not* in typing annotations:

$$\mathsf{fv}_0(\mathsf{a}^A) = \{\!\{\mathsf{a}^A\}\!\}, \qquad \mathsf{fv}_0(x) = \{\!\{\}\!\}, \qquad \mathsf{fv}_0(\{x\}e) = \mathsf{fv}_0(e),$$
$$\mathsf{fv}_0(\mathsf{S}(e_1, \ldots, e_n)) = \mathsf{fv}_0(e_1) \cup \cdots \cup \mathsf{fv}_0(e_n),$$
$$\mathsf{fv}_0(\mathsf{M}^{\mathcal{B}}(t_1, \ldots, t_n)) = \mathsf{fv}_0(t_1) \cup \cdots \cup \mathsf{fv}_0(t_n),$$
$$\mathsf{fv}_0(\kappa(t, \alpha)) = \mathsf{fv}_0(t) \cup \mathsf{fv}_0(\alpha),$$
$$\mathsf{fv}_0(\alpha) = \{\!\{\mathsf{a}^A \mid \mathsf{a}^A \in \alpha\}\!\},$$

Free variables *only* in typing annotations:

$$\mathsf{fvt}(e) = \bigcup \{\!\{\mathsf{fv}(A) \mid \mathsf{a}^A \in \mathsf{fv}_0(e)\}\!\}$$

Free variables:

$$\mathsf{fv}(e) = \mathsf{fv}_0(e) \cup \mathsf{fvt}(e).$$

Bound variables:

$$\mathsf{bv}(\mathsf{a}^A) = \{\!\{\}\!\}, \qquad \mathsf{bv}(x) = \{\!\{x\}\!\}, \qquad \mathsf{bv}(\{x\}e) = \mathsf{bv}(e) \setminus \{\!\{x\}\!\},$$
$$\mathsf{bv}(\mathsf{S}(e_1, \ldots, e_n)) = \mathsf{bv}(e_1) \cup \cdots \cup \mathsf{bv}(e_n),$$
$$\mathsf{bv}(\mathsf{M}^{\mathcal{B}}(t_1, \ldots, t_n)) = \mathsf{bv}(t_1) \cup \cdots \cup \mathsf{bv}(t_n),$$
$$\mathsf{bv}(\kappa(t, \alpha)) = \mathsf{bv}(t) \cup \mathsf{bv}(\alpha),$$
$$\mathsf{bv}(\alpha) = \{\!\{x \mid x \in \alpha\}\!\}.$$

Metavariables:

$$\mathsf{mv}(\mathsf{a}^A) = \mathsf{mv}(A), \qquad \mathsf{mv}(x) = \{\!\{\}\!\}, \qquad \mathsf{mv}(\{x\}e) = \mathsf{mv}(e),$$
$$\mathsf{mv}(\mathsf{S}(e_1, \ldots, e_n)) = \mathsf{mv}(e_1) \cup \cdots \cup \mathsf{mv}(e_n),$$
$$\mathsf{mv}(\mathsf{M}^{\mathcal{B}}(t_1, \ldots, t_n)) = \{\!\{\mathsf{M}^{\mathcal{B}}\}\!\} \cup \mathsf{mv}(\mathcal{B}) \cup \mathsf{mv}(t_1) \cup \cdots \cup \mathsf{mv}(t_n),$$
$$\mathsf{mv}(\kappa(t, \alpha)) = \mathsf{mv}(t) \cup \mathsf{mv}(\alpha),$$
$$\mathsf{mv}(\alpha) = \{\!\{\mathsf{M}^{\mathcal{B}} \mid \mathsf{M}^{\mathcal{B}} \in \alpha\}\!\} \cup \bigcup \{\!\{\mathsf{mv}(A) \mid \mathsf{a}^A \in \alpha\}\!\}$$
$$\cup \bigcup \{\!\{\mathsf{mv}(\mathcal{B}) \mid \mathsf{M}^{\mathcal{B}} \in \alpha\}\!\}.$$

$$\mathsf{mv}(\Box\ \mathsf{type}) = \{\!\{\}\!\}, \qquad \mathsf{mv}(\Box : A) = \mathsf{mv}(A),$$
$$\mathsf{mv}(A \equiv B\ \mathsf{by}\ \Box) = \mathsf{mv}(A) \cup \mathsf{mv}(B),$$
$$\mathsf{mv}(s \equiv t : A\ \mathsf{by}\ \Box) = \mathsf{mv}(s) \cup \mathsf{mv}(t) \cup \mathsf{mv}(A),$$
$$\mathsf{mv}(\{x{:}A\}\mathcal{B}) = \mathsf{mv}(A) \cup \mathsf{mv}(\mathcal{B}).$$

Assumption sets:

$$\mathsf{asm}(e) = \mathsf{fv}(e) \cup \mathsf{bv}(e) \cup \mathsf{mv}(e).$$

Figure 3.2: Context-free variable occurrences and assumption sets

Abstraction:

$$\mathsf{a}^A[x/\mathsf{a}^A] = x, \qquad y[x/\mathsf{a}^A] = y,$$
$$b^B[x/\mathsf{a}^A] = b^B \quad \text{if } \mathsf{a}^A \neq b^B \text{ and } \mathsf{a}^A \notin \mathsf{fvt}(B)$$
$$\mathsf{S}(e_1, \dots, e_n)[x/\mathsf{a}^A] = \mathsf{S}(e_1[x/\mathsf{a}^A], \dots, e_n[x/\mathsf{a}^A]),$$
$$\mathsf{M}^{\mathcal{B}}(t_1, \dots, t_n)[x/\mathsf{a}^A] = \mathsf{M}^{\mathcal{B}}(t_1[x/\mathsf{a}^A], \dots, t_n[x/\mathsf{a}^A]),$$
$$\kappa(t, \alpha)[x/\mathsf{a}^A] = \kappa(t[x/\mathsf{a}^A], A[x/\mathsf{a}^A])\alpha[x/\mathsf{a}^A],$$
$$(\{y\}e)[x/\mathsf{a}^A] = \{y\}(e[x/\mathsf{a}^A]) \quad \text{if } x \neq y,$$
$$\alpha[x/\mathsf{a}^A] = \alpha \quad \text{if } \mathsf{a}^A \notin \alpha,$$
$$\alpha[x/\mathsf{a}^A] = (\alpha \setminus \{\!\{\mathsf{a}^A\}\!\}) \cup \{\!\{x\}\!\} \quad \text{if } \mathsf{a}^A \in \alpha \text{ and } \mathsf{a}^A \notin \mathsf{fvt}(\alpha),$$

Substitution:

$$\mathsf{a}^A[s/x] = \mathsf{a}^A, \qquad x[s/x] = s, \qquad y[s/x] = y \quad \text{if } x \neq y,$$
$$\mathsf{S}(e_1, \dots, e_n)[s/x] = \mathsf{S}(e_1[s/x], \dots, e_n[s/x])$$
$$\mathsf{M}^{\mathcal{B}}(t_1, \dots, t_n)[s/x] = \mathsf{M}^{\mathcal{B}}(t_1[s/x], \dots, t_n[s/x])$$
$$\kappa(t, \alpha)[s/x] = \kappa(t[s/x], \alpha[s/x])$$
$$(\{y\}e)[s/x] = \{y\}(e[s/x])$$
$$\alpha[s/x] = \alpha \quad \text{if } x \notin \alpha$$
$$\alpha[s/x] = (\alpha \setminus \{\!\{x\}\!\}) \cup \mathsf{asm}(s) \quad \text{if } x \in \alpha.$$

Figure 3.3: Abstraction and substitution

The mapping $e \mapsto \lfloor e \rfloor$ takes the context-free raw syntax of Fig. 3.1 to the type-theoretic raw syntax of Fig. 2.1 where the variables a^A and the metavariables $\mathsf{M}^{\mathcal{B}}$ are construed as atomic symbols, i.e. their annotations are part of the symbol name.

3.1.1.5 Judgements and boundaries

The lower part of Fig. 3.1 summarizes the syntax of context-free judgements and boundaries. Apart from not having contexts, type judgements "A type" and term judgements "$t : A$" are as before. Equality judgements are modified to carry assumption sets: a type equality takes the form "$A \equiv B$ by α" and a term equality "$s \equiv t : A$ by α".

Boundaries do not change, except of course that they have no contexts. The head of a boundary is filled like before, except that assumption sets are used instead of dummy values, see Fig. 3.4.

Filling the placeholder with a head:

$$
\begin{aligned}
(\Box\ \mathsf{type})\boxed{A} &= (A\ \mathsf{type}) \\
(\Box : A)\boxed{t} &= (t : A) \\
(A \equiv B\ \mathsf{by}\ \Box)\boxed{e} &= (A \equiv B\ \mathsf{by}\ \mathsf{asm}(e)) \\
(s \equiv t : A\ \mathsf{by}\ \Box)\boxed{e} &= (s \equiv t : A\ \mathsf{by}\ \mathsf{asm}(e)) \\
(\{x{:}A\}\ \mathcal{B})\boxed{\{x\}e} &= (\{x{:}A\}\ \mathcal{B}\boxed{e}).
\end{aligned}
$$

Filling the placeholder with an equality:

$$
\begin{aligned}
(\Box\ \mathsf{type})\boxed{A_1 \equiv A_2\ \mathsf{by}\ \alpha} &= (A_1 \equiv A_2\ \mathsf{by}\ \alpha), \\
(\Box : A)\boxed{t_1 \equiv t_2\ \mathsf{by}\ \alpha} &= (t_1 \equiv t_2 : A\ \mathsf{by}\ \alpha), \\
(\{x{:}A\}\ \mathcal{B})\boxed{e_1 \equiv e_2\ \mathsf{by}\ \alpha} &= (\{x{:}A\}\ \mathcal{B}\boxed{e_1 \equiv e_2\ \mathsf{by}\ \alpha}),
\end{aligned}
$$

Figure 3.4: Context-free filling the head of a boundary

Free-variable occurrences in judgements and boundaries are defined as follows:

$$
\begin{aligned}
\mathsf{fv}_0(A\ \mathsf{type}) = \mathsf{fv}_0(A), \qquad \mathsf{fv}_0(t : A) &= \mathsf{fv}_0(t) \cup \mathsf{fv}_0(A), \\
\mathsf{fv}_0(A \equiv B\ \mathsf{by}\ \alpha) &= \mathsf{fv}_0(A) \cup \mathsf{fv}_0(B) \cup \mathsf{fv}_0(\alpha), \\
\mathsf{fv}_0(s \equiv t : A\ \mathsf{by}\ \alpha) &= \mathsf{fv}_0(s) \cup \mathsf{fv}_0(t) \cup \mathsf{fv}_0(A) \cup \mathsf{fv}_0(\alpha), \\
\mathsf{fv}_0(\{x : A\}\mathcal{J}) &= \mathsf{fv}_0(A) \cup \mathsf{fv}_0(\mathcal{J}), \\
\mathsf{fv}(\mathcal{J}) &= \mathsf{fv}_0(\mathcal{J}) \cup \mathsf{fvt}(\mathcal{J}).
\end{aligned}
$$

We trust the reader can emulate the above definition to define the set $\mathsf{mv}(\mathcal{J})$ of metavariable occurrences in a judgement $\mathcal{J}$, as well as occurrences of free and metavariables in boundaries.

3.1.1.6 Metavariable instantiations

Next, let us rethink how metavariable instantiations work in the presence of the newly introduced syntactic constructs. As before an ***instantiation*** is a sequence, representing a map,

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

such that $\mathsf{mv}(\mathcal{B}_i) \subseteq \{\!\!\{\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_{i-1}^{\mathcal{B}_{i-1}}\}\!\!\}$ and $\mathsf{ar}(\mathcal{B}_i) = \mathsf{ar}(e_i)$, for each $i = 1, \ldots, n$. As in Section 2.1.1.5, I acts on an expression e, provided that $\mathsf{mv}(e) \subseteq |I|$, by replacing metavariables with the corresponding expressions, see Fig. 3.5. Note that the action of I on a free variable changes the identity of the variable by acting on its typing annotation.

$$I_* x = x, \qquad I_*(\kappa(t, \alpha)) = \kappa(I_* t, I_* \alpha),$$
$$I_* \mathsf{a}^A = \mathsf{a}^{I_* A}, \qquad I_*(\{x\}e) = \{x\}(I_* e),$$
$$I_*(\mathsf{S}(e_1, \ldots, e_k)) = \mathsf{S}(I_* e_1, \ldots, I_* e_k),$$
$$I_*(\mathsf{M}_i^{\mathcal{B}_i}(t_1, \ldots, t_{m_i})) = I(\mathsf{M}_i)[(I_* t_1)/x_1, \ldots, (I_* t_{m_i})/x_{m_i}],$$
$$I_* \alpha = \bigcup \{\mathsf{asm}(I(\mathsf{M}_i)) \mid \mathsf{M}_i^{\mathcal{B}_i} \in \alpha\} \cup \{\!\{x \mid x \in \alpha\}\!\} \cup \{\!\{\mathsf{a}^{I_* A} \mid \mathsf{a}^A \in \alpha\}\!\}.$$

$$I_*(A \ \mathsf{type}) = (I_* A \ \mathsf{type}),$$
$$I_*(t : A) = (I_* t : I_* A),$$
$$I_*(A \equiv B \ \mathsf{by}\ \alpha) = (I_* A \equiv I_* B \ \mathsf{by}\ I_* \alpha),$$
$$I_*(\{x{:}A\}\mathcal{J}) = \{x{:}I_* A\} I_* \mathcal{J},$$
$$I_* \Box = \Box.$$

Figure 3.5: The action of a metavariable instantiation

3.1.2 Context-free rules and type theories

In this section we adapt the notions of raw and finitary rules and type theories to the context-free setting. We shall be rather telegraphic about it, as the changes are straightforward and require little discussion.

Definition 3.1.1. A ***context-free raw rule*** R over a signature Σ has the form

$$\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow j$$

where the ***premises*** $\mathcal{B}_i$ and the ***conclusion*** j are closed and syntactically valid over Σ, $\mathsf{mv}(\mathcal{B}_i) \subseteq \{\!\{\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_{i-1}^{\mathcal{B}_{i-1}}\}\!\}$ for every $i = 1, \ldots, n$, and $\mathsf{mv}(j) = \{\!\{\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n}\}\!\}$. We say that R is an ***object rule*** when j is a type or a term judgement, and an ***equality rule*** when j is an equality judgement.

The condition $\mathsf{mv}(j) = \{\!\{\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n}\}\!\}$ ensures that the conclusion of an instantiation of a raw rule records all uses of variables. We shall need it in the proof of Theorem 3.3.5.

Example 3.1.2. The context-free version of equality reflection from Example 2.1.7 is

$$\mathsf{A}^{\Box\ \mathsf{type}},\quad \mathsf{s}^{\Box : \mathsf{A}^{\Box\ \mathsf{type}}},\quad \mathsf{t}^{\Box : \mathsf{A}^{\Box\ \mathsf{type}}},\quad \mathsf{p}^{\mathsf{Id}(\mathsf{A}^{\Box\ \mathsf{type}}, \mathsf{s}^{\Box : \mathsf{A}^{\Box\ \mathsf{type}}}, \mathsf{t}^{\Box : \mathsf{A}^{\Box\ \mathsf{type}}})}$$
$$\Longrightarrow \mathsf{s}^{\Box : \mathsf{A}^{\Box\ \mathsf{type}}} \equiv \mathsf{t}^{\Box : \mathsf{A}^{\Box\ \mathsf{type}}} : \mathsf{A}^{\Box\ \mathsf{type}} \ \mathsf{by}\ \{\!\{\mathsf{p}^{\mathsf{Id}(\mathsf{A}^{\Box\ \mathsf{type}}, \mathsf{s}^{\Box : \mathsf{A}^{\Box\ \mathsf{type}}}, \mathsf{t}^{\Box : \mathsf{A}^{\Box\ \mathsf{type}}})}\}\!\}$$

which is quite unreadable. We indulge in eliding annotations on any variable that is already typed by a premise or a hypothesis, and write just

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A} \qquad \vdash \mathsf{p} : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A}\ \mathsf{by}\ \{\!\{\mathsf{p}\}\!\}}\ \textsc{CF-Eq-Reflect}$$

As there are no contexts, we could remove $\vdash$ too, but we leave it there out of habit. Note how the assumption set in the conclusion must record dependence on p, or else it would violate the assumption set condition of Definition 3.1.1.

When formulating equality closure rules we face a choice of assumptions sets. For example, what should γ be in the transitivity rule

$$\frac{\vdash A \equiv B \text{ by } \alpha \qquad \vdash B \equiv C \text{ by } \beta}{\vdash A \equiv C \text{ by } \gamma}\ ?$$

Its intended purpose is to record any assumptions used in the premises but not already recorded by A and C, which suggests the requirement

$$\mathsf{asm}(A) \cup \mathsf{asm}(B) \cup \mathsf{asm}(C) \cup \alpha \cup \beta \subseteq \mathsf{asm}(A) \cup \mathsf{asm}(C) \cup \gamma.$$

If we replace $\subseteq$ with $=$ we also avoid any extraneous asumptions, which leads to the following definition.

Definition 3.1.3. In a closure rule $([p_1, \ldots, p_n], \mathscr{b}\boxed{\alpha})$ whose conclusion is an equality judgement, α is ***suitable*** when $\mathsf{asm}(p_1, \ldots, p_n) = \mathsf{asm}(\mathscr{b}\boxed{\alpha})$.

Provided that $\mathsf{asm}(\mathscr{b}) \subseteq \mathsf{asm}(p_1, \ldots, p_n)$, we may always take the minimal suitable assumption set $\alpha = \mathsf{asm}(p_1, \ldots, p_n) \setminus \mathsf{asm}(\mathscr{b})$. We do not insist on minimality, even though an implementation might make an effort to keep the assumption sets small, because minimality is not preserved by instantiations, whereas suitability is. We shall indicate the suitability requirement in an equality closure rule by stating it as the side condition "α suitable".

Definition 3.1.4. A ***context-free raw rule-boundary*** over a signature Σ has the form

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

where the boundaries $\mathscr{B}_i$ and $\mathscr{b}$ are closed and syntactically valid over Σ, $\mathsf{mv}(\mathscr{B}_i) \subseteq \{\!\{\mathsf{M}_1^{\mathscr{B}_1}, \ldots, \mathsf{M}_{i-1}^{\mathscr{B}_{i-1}}\}\!\}$ for every $i = 1, \ldots, n$, and $\mathsf{mv}(\mathscr{b}) \subseteq \{\!\{\mathsf{M}_1^{\mathscr{B}_1}, \ldots, \mathsf{M}_n^{\mathscr{B}_n}\}\!\}$. We say that R is an ***object rule-boundary*** when $\mathscr{b}$ is an object boundary, and an ***equality rule-boundary*** when $\mathscr{b}$ is an equality boundary.

Definition 3.1.5. Given an object rule-boundary

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

over Σ, the ***associated symbol arity*** is $(c, [\mathsf{ar}(\mathcal{B}_1), \ldots, \mathsf{ar}(\mathcal{B}_n)])$, where $c \in \{\mathsf{Ty}, \mathsf{Tm}\}$ is the syntactic class of $\mathcal{b}$. The ***associated symbol rule*** for $\mathsf{S} \notin |\Sigma|$ is the raw rule

$$\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow \mathcal{b}\boxed{\mathsf{S}(\widehat{\mathsf{M}}_1^{\mathcal{B}_1}, \ldots, \widehat{\mathsf{M}}_n^{\mathcal{B}_n})}$$

over the extended signature $\langle \Sigma, \mathsf{S} {\mapsto} (c, [\mathsf{ar}(\mathcal{B}_1), \ldots, \mathsf{ar}(\mathcal{B}_n)]) \rangle$, where $\widehat{\mathsf{M}}^{\mathcal{B}}$ is the ***generic application*** of the metavariable $\mathsf{M}^{\mathcal{B}}$, defined as:

1. $\widehat{\mathsf{M}}^{\mathcal{B}} = \{x_1\} \cdots \{x_k\} \mathsf{M}^{\mathcal{B}}(x_1, \ldots, x_k)$ if $\mathsf{ar}(\mathcal{B}) = (c, k)$ and $c \in \{\mathsf{Ty}, \mathsf{Tm}\}$,
2. $\widehat{\mathsf{M}}^{\mathcal{B}} = \{x_1\} \cdots \{x_k\} \{\!\!\{ \mathsf{M}^{\mathcal{B}}, x_1, \ldots, x_k \}\!\!\}$ if $\mathsf{ar}(\mathcal{B}) = (c, k)$ and $c \in \{\mathsf{EqTy}, \mathsf{EqTm}\}$.

Definition 3.1.6. Given an equality rule-boundary

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

the ***associated equality rule*** is

$$\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow \mathcal{b}\boxed{\{\!\!\{ \mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \}\!\!\} \setminus \mathsf{asm}(\mathcal{b})}.$$

Definition 3.1.7. An ***instantiation*** of a raw rule

$$R = (\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow \mathcal{b}\boxed{e})$$

over a signature Σ is an instantiation $I = \langle \mathsf{M}_1^{\mathcal{B}_1} {\mapsto} e_1, \ldots, \mathsf{M}_n^{\mathcal{B}_n} {\mapsto} e_n \rangle$ of the metavariables of R. The ***closure rule*** $I_* R$ associated with I and R is $([p_1, \ldots, p_n, q], r)$ where p_i is $\vdash I_{(i)*}\mathcal{B}_i)\boxed{e_i}$, q is $\vdash I_* \mathcal{b}$, and r is $\vdash I_*(\mathcal{b}\boxed{e})$.

A minor complication arises when congruence rules (Definition 2.1.13) are adapted to the context-free setting, because conversions must be inserted. Consider the congruence rule (2.1) for Π from Example 2.1.14. The premise $A_1 \equiv A_2$ ensures that the premise $\{x{:}A_1\}\ B_1(x) \equiv B_2(x)$ is well-formed by conversion of x on the right-hand side from A_1 to A_2, thus in the context-free version of the rule we should allow for the possibility of an explicit conversion. However, we should not enforce an unnecessary conversion in case $A_1 = A_2$, nor should we require particular conversions, as there may be many ways to convert a term. We therefore formulate flexible congruence rules as follows: if an occurrence of a term t possibly requires conversion, we allow in its place a term t' such that $\lfloor t \rfloor = \lfloor t' \rfloor$.

Definition 3.1.8. The ***context-free congruence rules*** associated with a context-free raw type rule

$$\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow A \ \mathsf{type}$$

are closure rules, where

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

of the following form:

$$
\frac{
\begin{array}{ll}
\vdash (I_{(i)*}\mathcal{B}_i)\boxed{f_i} & \text{for } i = 1, \ldots, n \\
\vdash (J_{(i)*}\mathcal{B}_i)\boxed{g_i} & \text{for } i = 1, \ldots, n \\
\lfloor g'_i \rfloor = \lfloor g_i \rfloor & \text{for object boundary } \mathcal{B}_i \\
\vdash (I_{(i)*}\mathcal{B}_i)\boxed{f_i \equiv g'_i \text{ by } \alpha_i} & \text{for object boundary } \mathcal{B}_i \\
 & \beta \text{ suitable}
\end{array}
}{
\vdash I_* A \equiv J_* A \text{ by } \beta
}
$$

Similarly, the congruence rule associated with a raw term rule

$$\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow t : A$$

are closure rules of the form

$$
\frac{
\begin{array}{ll}
\vdash (I_{(i)*}\mathcal{B}_i)\boxed{f_i} & \text{for } i = 1, \ldots, n \\
\vdash (J_{(i)*}\mathcal{B}_i)\boxed{g_i} & \text{for } i = 1, \ldots, n \\
\lfloor g'_i \rfloor = \lfloor g_i \rfloor & \text{for object boundary } \mathcal{B}_i \\
\vdash (I_{(i)*}\mathcal{B}_i)\boxed{f_i \equiv g'_i \text{ by } \alpha_i} & \text{for object boundary } \mathcal{B}_i \\
\vdash t' : I_* A & \lfloor t' \rfloor = \lfloor J_* t \rfloor \\
 & \beta \text{ suitable}
\end{array}
}{
\vdash I_* t \equiv t' : I_* A \text{ by } \beta
}
$$

Example 3.1.9. The context-free congruence rules for Π from Example 2.1.14 take the form

$$
\frac{
\begin{array}{rl}
\vdash A_1 \text{ type} & \vdash \{x{:}A_1\}\ B_1 \text{ type} \\
\vdash A_2 \text{ type} & \vdash \{x{:}A_2\}\ B_2 \text{ type} \\
\lfloor A'_2 \rfloor = \lfloor A_2 \rfloor & \lfloor \{x\}B'_2 \rfloor = \lfloor \{x\}B_2 \rfloor \\
\vdash A_1 \equiv A'_2 \text{ by } \alpha_1 & \vdash \{x{:}A_1\}\ B_1 \equiv B'_2 \text{ by } \alpha_2
\end{array}
}{
\vdash \Pi(A_1, \{x\}B_1) \equiv \Pi(A_2, \{x\}B_2) \text{ by } \beta
}
$$

where the minimal suitable β is

$$(\alpha_1 \cup \alpha_2 \cup \mathsf{asm}(A'_2, \{x\}B'_2)) \setminus (\mathsf{asm}(A_1, A_2, \{x\}B_1, \{x\}B_2)).$$

The type expressions A'_2 and B'_2 may be chosen in such a way that the equations $\vdash A_1 \equiv A'_2$ by α_1 and $\vdash \{x{:}A_1\}\ B_1 \equiv B'_2$ by α_2 are well-typed, so long as they match A_2 and B_2 up to erasure. In this case, we expect to be able to directly use A_2 for A'_2. The equation $\vdash \{x{:}A_1\}\ B_1 \equiv B_2$ by α_2 where we use B_2 instead of B'_2 is not obviously well-typed, as B_2 is a family over A_2 rather than A_1. Intuitively, B'_2 should thus be B_2 where uses of x have to first convert along the equation $A_1 \equiv A_2$ by α_1.

The context-free metavariable closure rules are in direct analogy with the usual ones from Definition 2.1.15:

Definition 3.1.10. The ***context-free metavariable rules*** associated with the metavariable $\mathsf{M}^{\mathcal{B}}$ where $\mathcal{B} = (\{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\ b)$ are the closure rules

$$\frac{\textsc{CF-Meta}\quad \vdash t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] \quad \text{for } i = 1, \ldots, n \qquad \vdash b[\vec{t}/\vec{x}]}{\vdash (b[\vec{t}/\vec{x}])\boxed{\mathsf{M}^{\mathcal{B}}(\vec{t})}}$$

where $\vec{x} = (x_1, \ldots, x_m)$, $\vec{t} = (t_1, \ldots, t_m)$. In the second line of premises, we thus substitute the preceding term arguments $t_1, \ldots, t_{j-1}$ for the bound variables $x_1, \ldots, x_{j-1}$ in each type A_j. The last premise ensures the well-formedness of the boundary of the conclusion, just like the definition of the closure rule associated to a raw rule (Def. 2.1.8).

Furthermore, if b is an object boundary, then the ***metavariable congruence rules*** for $\mathsf{M}^{\mathcal{B}}$ are the closure rules CF-Meta-Congr-Ty and CF-Meta-Congr-Tm displayed in Fig. 3.6.

The following definition of context-free raw type theories is analogous to Definition 2.1.16, except that we have to use the context-free versions of structural rules.

Definition 3.1.11. A ***context-free raw type theory*** T over a signature Σ is a family of context-free raw rules, called the ***specific rules*** of T. The ***associated deductive system*** of T consists of:

1. the ***structural rules*** over Σ:
   a) the *variable*, *metavariable*, *metavariable congruence*, and *abstraction* closure rules (Fig. 3.6),
   b) the *equality* closure rules (Fig. 3.7),
   c) the *boundary* closure rules (Fig. 3.8);
2. the instantiations of the specific rules of T (Definition 3.1.7);
3. for each specific object rule of T, the instantiations of the associated congruence rule (Definition 3.1.8).

We write $\vdash_T \mathcal{J}$ when $\vdash \mathcal{J}$ is derivable with respect to the deductive system associated to T, and similarly for $\vdash_T \mathcal{B}$.

The formulations of the abstraction rules CF-Abstr and CF-Bdry-Abstr are suitable for the backward-chaining style of proof, because their conclusions take a general form. For forward-chaining, we may derive abstraction rules with premises in general form as follows:

$$\frac{\textsc{CF-Abstr-Fwd}\quad \vdash A \ \mathsf{type} \qquad \vdash \mathcal{J} \qquad \mathsf{a}^A \notin \mathsf{fvt}(\mathcal{J})}{\vdash \{x{:}A\}\ \mathcal{J}[x/\mathsf{a}^A]} \qquad \frac{\textsc{CF-Bdry-Abstr-Fwd}\quad \vdash A \ \mathsf{type} \qquad \vdash \mathcal{B} \qquad \mathsf{a}^A \notin \mathsf{fvt}(\mathcal{B})}{\vdash \{x{:}A\}\ \mathcal{B}[x/\mathsf{a}^A]}$$

The side condition $\mathsf{a}^A \notin \mathsf{fvt}(\mathcal{J})$ ensures that $\mathsf{a}^A \notin \mathsf{fv}(\mathcal{J}[x/\mathsf{a}^A])$, hence CF-Abstr-Fwd can be derived as the instance of CF-Abstr

$$\frac{\vdash A \;\mathsf{type} \qquad \mathsf{a}^A \notin \mathsf{fv}(\mathcal{J}[x/\mathsf{a}^A]) \qquad \vdash (\mathcal{J}[x/\mathsf{a}^A])[\mathsf{a}^A/x]}{\vdash \{x{:}A\}\mathcal{J}[x/\mathsf{a}^A]}$$

and similarly for boundary abstractions.

CF-Var
$$\frac{}{\vdash \mathsf{a}^A : A}$$

CF-Abstr
$$\frac{\vdash A \;\mathsf{type} \qquad \mathsf{a}^A \notin \mathsf{fv}(\mathcal{J}) \qquad \vdash \mathcal{J}[\mathsf{a}^A/x]}{\vdash \{x{:}A\}\;\mathcal{J}}$$

CF-Meta
$$\frac{\begin{array}{l}\vdash t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] \quad \text{for } i = 1,\ldots,n \\ \vdash \mathcal{b}[\vec{t}/\vec{x}]\end{array}}{\vdash (\mathcal{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}^{\{\vec{x}:\vec{A}\}\mathcal{b}}(\vec{t})}}$$

CF-Meta-Congr-Ty
$$\frac{\begin{array}{ll}\mathcal{B} = \{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\;\Box\;\mathsf{type} & \\ \vdash s_j : A[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\ldots,m \\ \vdash t_j : A[\vec{t}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\ldots,m \\ \lfloor t_j \rfloor = \lfloor t'_j \rfloor & \text{for } j = 1,\ldots,m \\ \vdash s_j \equiv t'_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] \;\mathsf{by}\; \alpha_j & \text{for } j = 1,\ldots,m \\ \beta \text{ suitable} & \end{array}}{\vdash \mathsf{M}^{\mathcal{B}}(\vec{s}) \equiv \mathsf{M}^{\mathcal{B}}(\vec{t}) \;\mathsf{by}\; \beta}$$

CF-Meta-Congr-Tm
$$\frac{\begin{array}{ll}\mathcal{B} = \{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\;\Box : B & \\ \vdash s_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\ldots,m \\ \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1,\ldots,m \\ \lfloor t_j \rfloor = \lfloor t'_j \rfloor & \text{for } j = 1,\ldots,m \\ \vdash s_j \equiv t'_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] \;\mathsf{by}\; \alpha_j & \text{for } j = 1,\ldots,m \\ \multicolumn{2}{c}{\vdash v : B[\vec{s}/\vec{x}] \qquad \lfloor \mathsf{M}^{\mathcal{B}}(\vec{t}) \rfloor = \lfloor v \rfloor \qquad \beta \text{ suitable}}\end{array}}{\vdash \mathsf{M}^{\mathcal{B}}(\vec{s}) \equiv v : B[\vec{s}/\vec{x}] \;\mathsf{by}\; \beta}$$

Figure 3.6: Context-free free variable, metavariable, and abstraction closure rules

The context-free analogues of the auxiliary judgements $\vdash \Theta \;\mathsf{mctx}$ and $\Theta \vdash \Gamma \;\mathsf{vctx}$ are as follows. For simplicity we define a single notion that encompasses the well-formedness of all annotations.

CF-EqTy-Refl
$$\frac{\vdash A_1 \text{ type} \qquad \vdash A_2 \text{ type} \qquad \lfloor A_1 \rfloor = \lfloor A_2 \rfloor}{\vdash A_1 \equiv A_2 \text{ by } \{\!\!\}}$$

CF-EqTy-Sym
$$\frac{\vdash A \equiv B \text{ by } \alpha}{\vdash B \equiv A \text{ by } \alpha}$$

CF-EqTy-Trans
$$\frac{\vdash A \equiv B \text{ by } \alpha \qquad \vdash B \equiv C \text{ by } \beta \qquad \gamma \text{ suitable}}{\vdash A \equiv C \text{ by } \gamma}$$

CF-EqTm-Refl
$$\frac{\vdash t_1 : A \qquad \vdash t_2 : A \qquad \lfloor t_1 \rfloor = \lfloor t_2 \rfloor}{\vdash t_1 \equiv t_2 : A \text{ by } \{\!\!\}}$$

CF-EqTm-Sym
$$\frac{\vdash s \equiv t : A \text{ by } \alpha}{\vdash t \equiv s : A \text{ by } \alpha}$$

CF-EqTm-Trans
$$\frac{\vdash s \equiv t : A \text{ by } \alpha \qquad \vdash t \equiv u : A \text{ by } \beta \qquad \gamma \text{ suitable}}{\vdash s \equiv u : A \text{ by } \gamma}$$

CF-Conv-Tm
$$\frac{\vdash t : A \qquad \vdash A \equiv B \text{ by } \alpha \qquad \mathsf{asm}(t, A, B, \alpha) = \mathsf{asm}(t, B, \beta)}{\vdash \kappa(t, \beta) : B}$$

CF-Conv-EqTm
$$\frac{\vdash s \equiv t : A \text{ by } \alpha \qquad \vdash A \equiv B \text{ by } \beta \qquad \mathsf{asm}(s, A, B, \beta) = \mathsf{asm}(s, B, \gamma) \qquad \mathsf{asm}(t, A, B, \beta) = \mathsf{asm}(t, B, \delta)}{\vdash \kappa(s, \gamma) \equiv \kappa(t, \delta) : B \text{ by } \alpha}$$

Figure 3.7: Context-free closure rules for equality

CF-Bdry-Ty
$$\frac{}{\vdash \Box \text{ type}}$$

CF-Bdry-Tm
$$\frac{\vdash A \text{ type}}{\vdash \Box : A}$$

CF-Bdry-EqTy
$$\frac{\vdash A \text{ type} \qquad \vdash B \text{ type}}{\vdash A \equiv B \text{ by } \Box}$$

CF-Bdry-EqTm
$$\frac{\vdash A \text{ type} \qquad \vdash s : A \qquad \vdash t : A}{\vdash s \equiv t : A \text{ by } \Box}$$

CF-Bdry-Abstr
$$\frac{\vdash A \text{ type} \qquad \mathsf{a}^A \notin \mathsf{fv}(\mathcal{B}) \qquad \vdash \mathcal{B}[\mathsf{a}^A/x]}{\vdash \{x{:}A\}\ \mathcal{B}}$$

Figure 3.8: Well-formed context-free abstracted boundaries

Definition 3.1.12. An expression e has ***well-typed annotations*** when $\vdash \mathcal{B}$ for every $\mathsf{M}^{\mathcal{B}} \in \mathsf{asm}(e)$ and $\vdash A$ type for every $\mathsf{a}^A \in \mathsf{asm}(e)$. The notion evidently extends to judgements and boundaries.

The context-free version of finitary rules and type theories is quite similar to the original one.

Definition 3.1.13. Given a raw theory T over a signature Σ, a context-free raw rule $\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow \mathscr{b}\boxed{e}$ over Σ is ***finitary*** over T when $\vdash_T \mathcal{B}_i$ for $k = 1, \ldots, n$, and $\vdash_T \mathscr{b}$, Similarly, a raw rule-boundary $\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow \mathscr{b}$ is finitary over T when $\vdash_T \mathcal{B}_i$ for $k = 1, \ldots, n$, and $\vdash_T \mathscr{b}$.

A ***context-free finitary type theory*** is a context-free raw type theory $(R_i)_{i \in I}$ for which there exists a well-founded order $(I, <)$ such that each R_i is finitary over $(R_j)_{j<i}$.

Definition 3.1.14. A context-free finitary type theory is ***standard*** if its specific object rules are symbol rules, and each symbol has precisely one associated rule.

3.2 Metatheorems about context-free theories

The metatheorems from Section 2.2 carry over to the context-free setting. Unfortunately, there seems to be no wholesale method for transferring the proofs, and one simply has to adapt them manually to the context-free setting. The process is quite straightforward, so we indulge in omitting the details.

3.2.1 Metatheorems about context-free raw theories

In the context-free setting, a renaming is still an injective map ρ taking unannotated symbols to unannotated symbols. Its action $\rho_* e$ on an expression e recursively descends into e, including into variable annotations, i.e. $\rho_*(\mathsf{a}^A) = \rho(\mathsf{a})^{\rho_* A}$ and $\rho_*(\mathsf{M}^{\mathcal{B}}) = \rho(\mathsf{M})^{\rho_* \mathcal{B}}$. The action is extended to judgements and boundaries in a straightforward manner. Renaming preserves the size of an expression, as long as all symbols are deemed to have the same size.

Proposition 3.2.1 (Context-free renaming)**.** *If a context-free raw type theory derives a judgement or a boundary, then it also derives its renaming.*

Proof. Straightforward induction on the derivation. □

Weakening (Proposition 2.2.2) is not applicable, as there is no context that could be weakened, and no variable ever occurs in the conclusion of a judgement without it being used in the derivation.

We next prove admissibility of substitution rules. We take a slightly different route than in Section 2.2.1 in order to avoid substituting a term for a free variable, as that changes type annotations and therefore the identity of variables. Lemmas 3.2.2 and 3.2.3 are proved by mutual structural induction, with a further structural induction within each lemma.

Lemma 3.2.2. *If a context-free raw type theory derives*

$$\vdash \{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\ \mathcal{J} \qquad \text{and}$$
$$\vdash t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] \qquad \text{for } i = 1, \ldots, n$$

then it derives $\vdash \mathcal{J}[\vec{t}/\vec{x}]$.

Proof. We may invert the derivation of $\vdash \{\vec{x}{:}\vec{A}\}\ \mathcal{J}$ to obtain a series of applications of CF-Abstr, yielding types $A'_1, \ldots, A'_n$ and (suitably fresh) free variables $\mathsf{a}_1^{A'_1}, \ldots, \mathsf{a}_n^{A'_n}$ where, for $i = 1, \ldots, n$,

$$A'_i = A_i[\mathsf{a}_1^{A'_1}/x_1, \ldots, \mathsf{a}_{i-1}^{A'_{i-1}}/x_{i-1}] \quad \text{and} \quad \vdash A'_i \ \mathsf{type}.$$

At the top of the abstractions sits a derivation D of the judgement

$$\mathcal{J}[\mathsf{a}_1^{A'_1}/x_1, \ldots, \mathsf{a}_n^{A'_n}/x_n].$$

The proof proceeds by induction on the derivation D, i.e. we only ever apply the induction hypotheses to derivations that have a series of abstractions, and on the top a derivation that is structurally smaller than D. Let us write

$$\theta = [\mathsf{a}_1^{A'_1}/x_1, \ldots, \mathsf{a}_n^{A'_n}/x_n],$$
$$\zeta = [x_n/\mathsf{a}_n^{A'_n}, \ldots, x_1/\mathsf{a}_1^{A'_1}],$$
$$\tau = [t_1/x_1, \ldots, t_n/x_n].$$

Case CF-Var: Suppose the derivation ends with the variable rule

$$\frac{}{\vdash \mathsf{b}^B : B}$$

If b^B is one of $\mathsf{a}_i^{A'_i}$ then $\mathcal{J} = \{\vec{x}{:}\vec{A}\}\ x_i : A_i$, hence $\mathcal{J}\tau = (t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}])$, which is derivable by assumption. If b^B is none of $\mathsf{a}_i^{A_i}$'s then $\mathsf{a}_i^{A_i} \notin \mathsf{fv}(B)$ by freshness, hence $\mathcal{J}\tau = (\mathsf{b}^B : B)$, so we may reuse the same variable rule.

Case CF-Abstr: Suppose the derivation ends with an abstraction

$$\frac{\vdash (A_{n+1}\theta)\ \mathsf{type} \qquad \mathsf{a}_{n+1}^{A_{n+1}\theta} \notin \mathsf{fv}(\mathcal{J}'\theta) \qquad (\mathcal{J}'\theta)[\mathsf{a}_{n+1}^{A_{n+1}\theta}/x_{n+1}]}{\vdash \{x_{n+1}{:}A_{n+1}\theta\}\ (\mathcal{J}'\theta)}$$

We extend the substitution by $t_{n+1} = \mathsf{a}_{n+1}^{A_{n+1}\tau}$ and apply the induction hypothesis to the abstracted derivation of the right-hand premise, whose conclusion is $\{\vec{x} : \vec{A}\}\{x_{n+1} : A_{n+1}\}\ \mathcal{J}'$, to obtain $\vdash \mathcal{J}'[\tau, t_{n+1}/x_{n+1}]$. We may abstract $\mathsf{a}_{n+1}^{A_{n+1}\tau}$ to get the desired judgement $\vdash \{x_{n+1}{:}A_{n+1}\tau\}\ (\mathcal{J}'\tau)$.

All other cases: The remaining cases all follow the same pattern: abstract the premises, apply the induction hypotheses to them, and conclude with the same rule. We

demonstrate how this works in case of D ending with an instance of a specific rule $R = (\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow \mathscr{b}\boxed{e})$ instantiated with $I = \langle \mathsf{M}_1^{\mathcal{B}_1} \mapsto e_1, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \mapsto e_n \rangle$:

$$\frac{\vdash (I_{(i)*}\mathcal{B}_i)\boxed{e_i} \quad \text{for } i = 1, \ldots, n \qquad \vdash I_*\mathscr{b}}{\vdash I_*(\mathscr{b}\boxed{e})}$$

Define the instantiation J of the premises of R by $J(\mathsf{M}_i) = e_i' = (e_i\zeta)\tau$. Note that $\vdash (I_*(\mathscr{b}\boxed{e}))\tau$ equals $\vdash J_*(\mathscr{b}\boxed{e})$, therefore we may derive it by J_*R. The last premise of J_*R is $\vdash (I_*\mathscr{b})\tau$, and it follows by Lemma 3.2.3 applied to the last premise of I_*R. For $i = 1, \ldots, n$, abstract $\vdash (I_{(i)*}\mathcal{B}_i)\boxed{e_i}$ to

$$\vdash \{\vec{x}{:}\vec{A}\}\ ((I_{(i)*}\mathcal{B}_i)\boxed{e_i})\zeta$$

and apply the induction hypothesis to derive $\vdash (((I_{(i)*}\mathcal{B}_i)\boxed{e_i})\zeta)\tau$, which equals $\vdash (((I_{(i)*}\mathcal{B}_i)\zeta)\tau)\boxed{(e_i\zeta)\tau}$ and because $\mathcal{B}_i$ does not contain any free variables, also to $\vdash (J_{(i)*}\mathcal{B}_i)\boxed{e_i'}$. □

Lemma 3.2.3. *If a context-free raw type theory derives*

$$\begin{array}{ll} \vdash \{x_1{:}A_1\} \cdots \{x_n{:}A_n\}\ \mathcal{B} & \textit{and} \\ \vdash t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] & \textit{for } i = 1, \ldots, n \end{array}$$

then it derives $\vdash \mathcal{B}[\vec{t}/\vec{x}]$.

Proof. We proceed as in the proof of Lemma 3.2.2, where CF-Bdry-Abstr is treated like CF-Abstr, and the remaining ones invert to Lemma 3.2.2. □

Theorem 3.2.4 (Context-free admissibility of substitution)**.** *In a context-free raw type theory, the following substitution rules are admissible:*

$$\textsc{CF-Subst}\ \frac{\vdash \{x{:}A\}\ \mathcal{J} \qquad \vdash t : A}{\vdash \mathcal{J}[t/x]} \qquad\qquad \textsc{CF-Bdry-Subst}\ \frac{\vdash \{x{:}A\}\ \mathcal{B} \qquad \vdash t : A}{\vdash \mathcal{B}[t/x]}$$

Proof. The admissibility of CF-Subst and CF-Bdry-Subst corresponds to the case $n = 1$ of Lemma 3.2.2 and Lemma 3.2.3, respectively. □

Before addressing the context-free versions of TT-Subst-EqTy and TT-Subst-EqTm, we prove the context-free presuppositivity theorem.

Of course, presuppositivity holds in the context-free setting as well.

Theorem 3.2.5 (Context-free presuppositivity)**.** *If a context-free raw type theory derives* $\vdash \mathcal{B}\boxed{e}$ *and* $\mathcal{B}\boxed{e}$ *has well-typed annotations, then it derives* $\vdash \mathcal{B}$.

Proof. The proof proceeds by induction on the number of metavariables appearing in the judgement and the derivation of $\vdash \mathcal{B}\boxed{e}$. That is, each appeal to the induction hypothesis reduces the number of metavariables, or is applied to a subderivation.

Case CF-VAR: Immediate, by the well-typedness of annotations.

Case CF-META: Immedate as the desired judgement is a premise of the rule.

Case CF-META-CONGR-TM: Suppose $\mathcal{B} = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \Box : B$ and consider a derivation ending with the metavariable congruence rule

$$\frac{\begin{array}{l}\text{for } k = 1, \ldots, m: \\ \vdash s_k : A_k[\vec{s}_{(k)}/\vec{x}_{(k)}] \\ \vdash t_k : A_k[\vec{t}_{(k)}/\vec{x}_{(k)}] \\ \lfloor t_k \rfloor = \lfloor t'_k \rfloor \\ \vdash s_k \equiv t'_k : A_k[\vec{s}_{(k)}/\vec{x}_{(k)}] \text{ by } \alpha_k \\ \vdash v : B[\vec{s}/\vec{x}] \qquad \lfloor \mathsf{M}^{\mathcal{B}}(\vec{t}) \rfloor = \lfloor v \rfloor \\ \beta \text{ suitable}\end{array}}{\vdash \mathsf{M}^{\mathcal{B}}(\vec{s}) \equiv v : B[\vec{s}/\vec{x}] \text{ by } \beta}$$

The presuppositions are derived as follows:

- $\vdash B[\vec{s}/\vec{x}]$ type follows by CF-SUBST from $\vdash \vec{x}{:}\vec{A}\ B$ type, which in turn follows by inversion on $\vdash \mathcal{B}$.
- $\vdash \mathsf{M}^{\mathcal{B}}(\vec{s}) : B[\vec{s}/\vec{x}]$ follows by CF-META.
- $v : B[\vec{s}/\vec{x}]$ is a premise.

Case CF-ABSTR: Consider an abstraction

$$\frac{\vdash A \text{ type} \qquad \mathsf{a}^A \notin \mathsf{fv}(\mathcal{B}\boxed{e}) \qquad \vdash (\mathcal{B}\boxed{e})[\mathsf{a}^A/x]}{\vdash \{x{:}A\}\ \mathcal{B}\boxed{e}}$$

By induction hypothesis on the last premise, we obtain $\vdash \mathcal{B}[\mathsf{a}^A/x]$ after which we apply CF-BDRY-ABSTR.

Case of a specific rule: Immediate, as the well-formedness of the boundary is a premise.

Case of a congruence rule: Consider a congruence rulles associated with an object rule R and instantiated with I and J, as in Definition 3.1.8.

If R concludes with $\vdash A$ type, the presuppositions are $\vdash I_*A$ type and $\vdash J_*A$ type, which are derivable by I_*R and J_*R, respectively.

If R concludes with $\vdash t : A$, the presuppositions are $\vdash I_*A$ type, $\vdash I_*t : I_*A$, and $\vdash t' : I_*A$. We derive the first one by applying the induction hypothesis to the premise $\vdash t' : I_*A$, the second one by I_*R, while the third one is a premise.

Cases CF-EqTy-Refl, CF-EqTy-Sym, CF-EqTy-Trans, CF-EqTm-Refl, CF-EqTm-Sym, CF-EqTm-Trans: These are all dispensed with straightforward appeals to the induction hypotheses.

Case CF-Conv-Tm: Consider a term conversion

$$\frac{\vdash t : A \qquad \vdash A \equiv B \text{ by } \alpha \qquad \mathsf{asm}(t, A, B, \alpha) = \mathsf{asm}(t, B, \beta)}{\vdash \kappa(t, \beta) : B}$$

By induction hypothesis for the second premise, $\vdash B$ type.

Case CF-Conv-EqTm: Consider a term equality conversion

$$\frac{\vdash s \equiv t : A \text{ by } \alpha \qquad \vdash A \equiv B \text{ by } \beta \qquad \mathsf{asm}(s, A, B, \beta) = \mathsf{asm}(s, B, \gamma) \qquad \mathsf{asm}(t, A, B, \beta) = \mathsf{asm}(t, B, \delta)}{\vdash \kappa(s, \gamma) \equiv \kappa(t, \delta) : B \text{ by } \alpha}$$

As in the previous case, the induction hypothesis for the second premise provides $\vdash B$ type. The induction hypothesis for the first premise yields

$$\vdash s : A \qquad \text{and} \qquad \vdash t : A$$

We may convert these to $\vdash \kappa(s, \gamma) : B$ and $\vdash \kappa(t, \delta) : B$ using the second premise. □

Let us now turn to meta-theorems stating that equal substitutions act equally. Once again we need to account for insertion of conversions. In congruence rules such conversions appeared in premises: equations associated to object premises of the shape $(I_{(i)*}\mathcal{B}_i)\boxed{f_i \equiv g'_i \text{ by } \alpha_i}$ referred to a primed version of g_i to allow the use of conversions in g_i. In the following lemma, conversions appear in the result of a substitution. Therefore, rather than being permissive about insertions of conversions, we are faced with showing that it is possible to insert them. Similarly to Lemma 2.2.7, we prove that equal terms can be substituted into a judgement to yield equal results, but the right hand side of these results is only prescribed up to erasure, namely as C' and u'.

Lemma 3.2.6. *If a context-free raw type theory derives*

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

where $\{\vec{x}{:}\vec{A}\} \ \mathcal{J}$ *has well-typed annotations, and for* $i = 1, \ldots, n$

$$\begin{aligned} &\vdash s_i : A_i[\vec{s}_{(i)}/\vec{x}_{(i)}] \\ &\vdash t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] \\ &\vdash s_i \equiv t'_i : A_i[\vec{s}_{(i)}/\vec{x}_{(i)}] \text{ by } \alpha_i \qquad \textit{and } \lfloor t'_i \rfloor = \lfloor t_i \rfloor. \end{aligned} \tag{3.1}$$

then:

1. *if* $\mathcal{J} = (\{\vec{y}{:}\vec{B}\}\ C\ \mathsf{type})$ *then there are* γ *and* C' *such that* $\lfloor C[\vec{t}/\vec{x}] \rfloor = \lfloor C' \rfloor$,

$$\vdash \{\vec{y}{:}\vec{B}[\vec{s}/\vec{x}]\}\ C[\vec{s}/\vec{x}] \equiv C' \text{ by } \gamma,$$

2. *if* $\mathcal{J} = (\{\vec{y}{:}\vec{B}\}\ u : C)$ *then there are* δ *and* u' *such that* $\lfloor u[\vec{t}/\vec{x}] \rfloor = \lfloor u' \rfloor$ *and*

$$\vdash \{\vec{y}{:}\vec{B}[\vec{s}/\vec{x}]\}\ u[\vec{s}/\vec{x}] \equiv u' : C[\vec{s}/\vec{x}] \text{ by } \delta.$$

Furthermore, no extraneous assumptions are introduced by γ, C', δ *and* u'*:*

$$\mathsf{asm}(\{\vec{y}\}\gamma, \{\vec{y}\}C', \{\vec{y}\}\delta, \{\vec{y}\}u') \subseteq \mathsf{asm}(\vec{s}, \vec{t}, \vec{t}', \vec{\alpha}, \{\vec{x}{:}\vec{A}\}\ \mathcal{J}).$$

Proof. As in the proof of Lemma 3.2.2, we invert the derivation of $\vdash \{\vec{x}{:}\vec{A}\}\ \mathcal{J}$ to obtain types $A'_1, \ldots, A'_n$ and (suitably fresh) free variables $\mathsf{a}_1^{A'_1}, \ldots, \mathsf{a}_n^{A'_n}$ where, for $i = 1, \ldots, n$,

$$A'_i = A_i[\mathsf{a}_1^{A'_1}/x_1, \ldots, \mathsf{a}_{i-1}^{A'_{i-1}}/x_{i-1}] \quad \text{and} \quad \vdash A'_i \text{ type}$$

and a derivation D of the judgement

$$\mathcal{J}[\mathsf{a}_1^{A'_1}/x_1, \ldots, \mathsf{a}_{i-1}^{A'_n}/x_n].$$

The proof proceeds by induction on the well-founded ordering of the rules, the number of metavariables, with a subsidiary induction on the derivation D. That is, each appeal to the induction hypotheses either decreases the number of metavariables appearing in the judgement, or descends to a subderivation of D. Let us write

$$\begin{aligned} \theta &= [\mathsf{a}_1^{A'_1}/x_1, \ldots, \mathsf{a}_{i-1}^{A'_n}/x_n], \\ \sigma &= [s_1/x_1, \ldots, s_n/x_n], \\ \tau &= [t_1/x_1, \ldots, t_n/x_n]. \end{aligned}$$

Case CF-VAR: Suppose the derivation ends with the variable rule

$$\frac{}{\vdash \mathsf{b}^B : B}$$

If b^B is one of $\mathsf{a}_i^{A'_i}$ then $\mathcal{J} = \{\vec{x}{:}\vec{A}\}\ x_i : A_i$, hence (2) is satisfied by (3.1). If b^B is none of $\mathsf{a}_i^{A_i}$'s then $\mathsf{a}_i^{A_i} \notin \mathsf{fv}(B)$ by freshness, hence (2) is satisfied by $\vdash \mathsf{b}^B \equiv \mathsf{b}^B : B$ by $\{\!\!\}\!\!\}$, which holds by CF-EQTM-REFL.

Case CF-ABSTR: Suppose the derivation ends with an abstraction

$$\frac{\vdash (A_{n+1}\theta) \text{ type} \qquad \mathsf{a}_{n+1}^{A_{n+1}\theta} \notin \mathsf{fv}(\mathcal{J}\theta) \qquad (\mathcal{J}'\theta)[\mathsf{a}_{n+1}^{A_{n+1}\theta}/y]}{\vdash \{x_{n+1}{:}A_{n+1}\theta\}\ (\mathcal{J}'\theta)} \tag{3.2}$$

We may abstract the first premise to $\vdash \{\vec{x}{:}\vec{A}\}\ A_{n+1}$ type, apply Lemma 3.2.2 to derive $\vdash A_{n+1}\tau$ type, and the induction hypothesis to obtain β_{n+1} and A' such that $\lfloor A_{n+1}\tau \rfloor = \lfloor A' \rfloor$,

$$\vdash A' \text{ type} \qquad \text{and} \qquad \vdash A_{n+1}\sigma \equiv A' \text{ by } \beta_{n+1}.$$

By CF-EqTy-Trans and CF-EqTy-Refl it follows that for some $\gamma_{n+1} \subseteq \beta_{n+1}$

$$\vdash A_{n+1}\sigma \equiv A_{n+1}\tau \text{ by } \gamma_{n+1}.$$

Let $\mathsf{a}_{n+1}^{A_{n+1}\sigma}$ be fresh, and define

$$s_{n+1} = t'_{n+1} = \mathsf{a}_{n+1}^{A_{n+1}\sigma}, \qquad t_{n+1} = \kappa(\mathsf{a}_{n+1}^{A_{n+1}\sigma}, \gamma_{n+1}), \quad \text{and} \quad \alpha_{n+1} = \{\!\!\{\}\!\!\}.$$

We may abstract the last premise of (3.2) to

$$\vdash \{x_1{:}A_1\} \cdots \{x_{n+1}{:}A_{n+1}\}\ \mathcal{J}'$$

apply the induction hypothesis with the given s_{n+1}, t_{n+1} and t'_{n+1} to derive either (1) or (2), and abstract $\mathsf{a}_{n+1}^{A_{n+1}\sigma}$ to get the desired judgements.

Case CF-Meta: We consider the case of an object metavariable, and leave the easier case of a type metavariable to the reader. Let $\mathcal{B} = (\{\vec{y}{:}\vec{B}\}\ \Box : C)$, and suppose the derivation ends with an application of the metavariable rule,

$$\frac{\begin{array}{l}\vdash u_j\theta : B_j[\vec{u}_{(j)}\theta/\vec{y}_{(j)}] \quad \text{for } j = 1, \ldots, m \\ \vdash \Box : C[\vec{u}\theta/\vec{y}]\end{array}}{\vdash \mathsf{M}^{\mathcal{B}}(\vec{u}\theta) : C[\vec{u}\theta/\vec{y}]} \tag{3.3}$$

For each $j = 1, \ldots, m$ we may abstract the premise of (3.3) to

$$\vdash \{\vec{x}{:}\vec{A}\}\ u_j : B_j[\vec{u}_{(j)}/\vec{y}_{(j)}]$$

and apply Lemma 3.2.2, once with $\vec{s}$ and once with $\vec{t}$, to derive

$$\vdash u_j\sigma : B_j[(\vec{u}\sigma)_{(j)}/\vec{y}_{(j)}],$$
$$\vdash u_j\tau : B_j[(\vec{u}\tau)_{(j)}/\vec{y}_{(j)}],$$

where we took into account the fact that B_j does not contain any bound variables. Also, by induction hypothesis there is $\delta_j \subseteq \bigcup_i \alpha_i$ and u'_j such that $\lfloor u_j\tau \rfloor = \lfloor u'_j \rfloor$ and

$$\vdash u_j\sigma \equiv u'_j \text{ by } B_j[(\vec{u}\sigma)_{(j)}/\vec{y}_{(j)}] \text{ by } \delta_j.$$

Next, we invert the last premise of (3.3) and abstract it to $\vdash \{\vec{x}{:}\vec{A}\}\ C[\vec{u}/\vec{y}]$ type. By induction hypothesis we obtain $\delta' \subseteq \bigcup_j \delta_j$ such that $\vdash C[\vec{u}\sigma/\vec{y}] \equiv C'$ by δ' and

$\lfloor C' \rfloor = \lfloor C[\vec{u}\tau/\vec{y}] \rfloor$, hence $\vdash C[\vec{u}\sigma/\vec{y}] \equiv C[\vec{u}\tau/\vec{y}]$ by δ for some $\delta \subseteq \delta'$. Now (2) is satisfied by

$$\vdash \kappa(\mathsf{M}^{\mathcal{B}}(\vec{u}\tau), \delta) : C[\vec{u}\sigma/\vec{y}],$$
$$\vdash \mathsf{M}^{\mathcal{B}}(\vec{u}\sigma) \equiv \kappa(\mathsf{M}^{\mathcal{B}}(\vec{u}\tau), \delta) : C[\vec{u}\sigma/\vec{y}] \text{ by } \textstyle\bigcup_j \delta_j.$$

where the last judgement follows by the congruence rule for $\mathsf{M}^{\mathcal{B}}$.

Case of a specific term rule: Suppose the derivation ends with a specific rule $R = (\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_m^{\mathcal{B}_m} \Longrightarrow u : B)$ instantiated with $I' = \langle \mathsf{M}_1^{\mathcal{B}_1} \mapsto e'_1, \ldots, \mathsf{M}_m^{\mathcal{B}_m} \mapsto e'_m \rangle$:

$$\frac{\vdash (I'_{(j)*}\mathcal{B}_j)\boxed{e'_j} \quad \text{for } j = 1, \ldots, n \qquad \vdash (\Box : I'_* B)}{\vdash I'_* u : I'_* B}$$

Let $\zeta = [x_n/\mathsf{a}_n^{A'_n}, \ldots, x_1/\mathsf{a}_1^{A'_1}]$ be the abstraction that undoes θ. Define $e_j = e'_j\zeta$ and $I = I'\zeta$, so that $e'_j = e_j\theta$ and $I' = I\theta$, which allows us to write the above judgement as

$$\frac{\vdash ((I_{(j)*}\mathcal{B}_j)\boxed{e_j})\theta \quad \text{for } j = 1, \ldots, n \qquad \vdash (\Box : I_* B)\theta}{\vdash (I_* u)\theta : (I_* B)\theta}$$

We invert the last premise, abstract to $\vdash \{\vec{x}{:}\vec{A}\}\ I_* B$ type, and apply Lemma 3.2.2 to derive $\vdash (I_* B)\sigma$ type. Next, the induction hypothesis provides $\beta \subseteq \bigcup_i \alpha_i$ and B' such that $\lfloor B' \rfloor = \lfloor (I_* B)\tau \rfloor$ and $\vdash (I_* B)\sigma \equiv B'$ by β. Therefore, we have $\beta' \subseteq \beta$ such that

$$\vdash (I_* B)\sigma \equiv (I_* B)\tau \text{ by } \beta'.$$

It suffices to find $\delta \subseteq \bigcup_i \alpha_i$ such that

$$\vdash (I_* u)\sigma \equiv \kappa((I_* u)\tau, \beta') : (I_* B)\sigma \text{ by } \delta.$$

This is precisely the conclusion of the congruence rule for R, so we derive its premises. For any $j = 1, \ldots, m$ we may abstract the j-th premise to

$$\vdash \{\vec{x}{:}\vec{A}\}\ (I_{(j)*}\mathcal{B}_j)\boxed{e_j}, \tag{3.4}$$

and apply Lemma 3.2.2, once with $\vec{s}$ and once with $\vec{t}$, to derive

$$\vdash ((I\sigma)_{(j)*}\mathcal{B}_j)\boxed{e_j\sigma} \qquad \text{and} \qquad \vdash ((I\tau)_{(j)*}\mathcal{B}_j)\boxed{e_j\tau}.$$

For each object premise with boundary $\mathcal{B}_j$, the remaining premises are provided precisely by the induction hypotheses.

Case of a specific type rule: Suppose the derivation ends with a specific rule $R = (\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_m^{\mathcal{B}_m} \Longrightarrow B \text{ type})$ instantiated with $I' = \langle \mathsf{M}_1^{\mathcal{B}_1} {\mapsto} e'_1, \ldots, \mathsf{M}_m^{\mathcal{B}_m} {\mapsto} e'_m \rangle$:

$$\frac{\vdash (I'_{(j)*}\mathcal{B}_j)\boxed{e'_j} \quad \text{for } j = 1, \ldots, n \qquad \vdash \Box \text{ type}}{\vdash I'_* B \text{ type}}$$

With ζ and I as in the previous case, we may write the above as

$$\frac{\vdash ((I_{(j)*}\mathcal{B}_j)\boxed{e_j})\theta \quad \text{for } j = 1, \ldots, n}{\vdash (I_* B)\theta \text{ type}}$$

where we elided the trivial boundary premise. It suffices to find $\gamma \subseteq \bigcup_i \alpha_i$ such that

$$\vdash (I_* B)\sigma \equiv (I_* B)\tau \text{ by } \gamma.$$

This is precisely the conclusion of the congruence rule for R, whose premises are derived as in the previous case.

Case CF-Conv-Tm: Suppose the derivation ends with an application of the conversion rule

$$\frac{\vdash u\theta : B\theta \qquad \vdash B\theta \equiv C\theta \text{ by } \beta\theta}{\vdash \kappa(u\theta, \beta\theta \cup \mathsf{asm}(B\theta)) : C\theta}$$

We abstract the first premise to $\vdash \{\vec{x}{:}\vec{A}\}\ u : B$ and apply the induction hypothesis to obtain $\delta \subseteq \bigcup_i \alpha_i$ and u' such that $\lfloor u' \rfloor = \lfloor u\tau \rfloor$ and

$$\vdash u\sigma \equiv u' : B\sigma \text{ by } \delta.$$

We abstract the second premise to $\vdash \{\vec{x}{:}\vec{A}\}\ B \equiv C$ by β, apply Lemma 3.2.2 to derive $\vdash B\sigma \equiv C\sigma$ by $\beta\sigma$, and use CF-Conv-EqTm to conclude:

$$\vdash \kappa(u\sigma, \beta\sigma \cup \mathsf{asm}(B\sigma)) \equiv \kappa(u', \beta\sigma \cup \mathsf{asm}(B\sigma)) : C\sigma \text{ by } \delta. \qquad \Box$$

Theorem 3.2.7. *In a context-free raw type theory, the following rules are admissible:*

CF-Subst-EqTy

$$\frac{\begin{array}{ll} \vdash \{\vec{x}{:}\vec{A}\}\{\vec{y}{:}\vec{B}\}\ C\ \mathsf{type} & \\ \vdash s_i : A_i[\vec{s}_{(i)}/\vec{x}_{(i)}] & \textit{for } i = 1,\dots,n \\ \vdash t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] & \textit{for } i = 1,\dots,n \\ \lfloor t_i \rfloor = \lfloor t'_i \rfloor & \textit{for } i = 1,\dots,n \\ \vdash s_i \equiv t'_i : A_i[\vec{s}_{(i)}/\vec{x}_{(i)}]\ \mathsf{by}\ \alpha_i & \textit{for } i = 1,\dots,n \\ & \beta\ \textit{suitable} \end{array}}{\vdash \{\vec{y}{:}\vec{B}[\vec{s}/\vec{x}]\}\ C[\vec{s}/x] \equiv C[\vec{t}/x]\ \mathsf{by}\ \beta}$$

CF-Subst-EqTm

$$\frac{\begin{array}{ll} \vdash \{\vec{x}{:}\vec{A}\}\{\vec{y}{:}\vec{B}\}\ u : C & \\ \vdash s_i : A_i[\vec{s}_{(i)}/\vec{x}_{(i)}] & \textit{for } i = 1,\dots,n \\ \vdash t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] & \textit{for } i = 1,\dots,n \\ \lfloor t_i \rfloor = \lfloor t'_i \rfloor & \textit{for } i = 1,\dots,n \\ \vdash s_i \equiv t'_i : A_i[\vec{s}_{(i)}/\vec{x}_{(i)}]\ \mathsf{by}\ \alpha_i & \textit{for } i = 1,\dots,n \\ & \beta\ \textit{suitable} \end{array}}{\vdash \{\vec{y}{:}\vec{B}[\vec{s}/\vec{x}]\}\ u[\vec{s}/\vec{x}] \equiv \kappa(u[\vec{t}/\vec{x}],\beta) : C[\vec{s}/x]\ \mathsf{by}\ \beta}$$

Proof. Lemma 3.2.6 applied to the premises of CF-Subst-EqTy provides γ and C' such that $\lfloor C[\vec{t}/\vec{x}] \rfloor = \lfloor C' \rfloor$ and

$$\vdash \{\vec{y}{:}\vec{B}[\vec{s}/\vec{x}]\}\ C[\vec{s}/\vec{x}] \equiv C'\ \mathsf{by}\ \gamma. \tag{3.5}$$

We would like to replace C' in the right-hand side with $C[\vec{t}/\vec{x}]$, which we can so long as

$$\vdash \{\vec{y}{:}\vec{B}[\vec{s}/\vec{x}]\}\ C'\ \mathsf{type} \quad \text{and} \quad \vdash \{\vec{y}{:}\vec{B}[\vec{s}/\vec{x}]\}\ C[\vec{t}/\vec{x}].$$

The first judgement holds by Theorem 3.2.5 applied to (3.5) under the abstraction, while the second one is a substitution instance of the first premise. This establishes admissibility of CF-Subst-EqTy.

In case of CF-Subst-EqTm the same lemma yields δ and u' such that $\lfloor u[\vec{t}/\vec{x}] \rfloor = \lfloor u' \rfloor$ and

$$\vdash \{\vec{y}{:}\vec{B}[\vec{s}/\vec{x}]\}\ u[\vec{s}/\vec{x}] \equiv u' : C[\vec{s}/x]\ \mathsf{by}\ \delta.$$

We would like to replace u' with a converted $u[\vec{t}/\vec{x}]$, which we can by an argument similar to the the one above. □

Lastly, we prove the context-free counterpart of instantiation admissibility Theorem 2.2.13. The notion of a derivable instantiation carries over easily to the context-free setting: $I = \langle \mathsf{M}_1^{\mathcal{B}_1} \mapsto e_1, \dots, \mathsf{M}_n^{\mathcal{B}_n} \mapsto e_n \rangle$ is ***derivable*** when $\vdash (I_{(i)*}\mathcal{B}_i)\boxed{e_i}$ for every $i = 1, \dots, n$.

Proposition 3.2.8 (Context-free admissibility of instantiation)**.** *In a raw type theory, if* $\vdash \mathcal{J}$ *is derivable, it has well-typed annotations, and* I *is a derivable instantiation such that* $\mathsf{mv}(\mathcal{J}) \subseteq |I|$*, then* $\vdash I_* \mathcal{J}$ *is derivable, and similarly for boundaries.*

Proof. We proceed by induction on the derivation of $\vdash \mathcal{J}$. We only devote attention to the metavariable and abstraction rules, as all the other cases are straightforward. Suppose $I = \langle \mathsf{M}_1^{\mathcal{B}_1} \mapsto e_1, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \mapsto e_n \rangle$.

Case CF-Meta: Consider an application of the metavariable rule for $\mathsf{M}_i^{\mathcal{B}_i}$ with $\mathcal{B}_i = (\{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathcal{b})$ and $e_i = \{\vec{x}\}e$:

$$\frac{\begin{array}{l}\vdash t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] \quad \text{for } i = 1, \ldots, n \\ \vdash \mathcal{b}[\vec{t}/\vec{x}]\end{array}}{\vdash ((\mathcal{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_i^{\mathcal{B}_i}(\vec{t})})}$$

Because $I_*((\mathcal{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_i^{\mathcal{B}_i}(\vec{t})}) = ((I_*\mathcal{b})[I_*\vec{t}/\vec{x}])\boxed{e[I_*\vec{t}/\vec{x}]}$ we need to derive

$$\vdash ((I_*\mathcal{b})[I_*\vec{t}/\vec{x}])\boxed{e[I_*\vec{t}/\vec{x}]}. \tag{3.6}$$

Because I is derivable, we know that $\vdash \{\vec{x}{:}I_{(i)*}\vec{A}\}\ (I_{(i)*}\mathcal{b})\boxed{e}$. By induction hypothesis $\vdash I_{(i)*}t_j : (I_{(i)*}A_j)[I_{(i)*}\vec{t}_{(j)}/\vec{x}_{(j)}])$ for each $j = 1, \ldots, m$, so by Lemma 3.2.2 we derive $\vdash ((I_{(i)*}\mathcal{b})\boxed{e})[I_{(i)*}\vec{t}/\vec{x}]$, which coincides with (3.6).

Case CF-Meta-Congr-Ty: We consider the congruence rule for types only. Suppose the derivation ends with an application of the congruence rule for $\mathsf{M}_i^{\mathcal{B}_i}$ with $\mathcal{B}_i = (\{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \Box\ \mathsf{type})$ and $e_i = \{\vec{x}\}B$:

$$\frac{\begin{array}{l}\text{for } k = 1, \ldots, m: \\ \quad \vdash s_k : A[\vec{s}_{(k)}/\vec{x}_{(k)}] \\ \quad \vdash t_k : A[\vec{t}_{(k)}/\vec{x}_{(k)}] \\ \lfloor t_k \rfloor = \lfloor t'_k \rfloor \\ \quad \vdash s_k \equiv t'_k : A[\vec{s}_{(k)}/\vec{x}_{(k)}] \text{ by } \alpha_k \\ \beta \text{ suitable}\end{array}}{\vdash \mathsf{M}_i^{\mathcal{B}_i}(\vec{s}) \equiv \mathsf{M}_i^{\mathcal{B}_i}(\vec{t}) \text{ by } \beta}$$

Because I is derivable, we know that $\vdash \{\vec{x}{:}I_{(i)*}\vec{A}\}\ B\ \mathsf{type}$, hence Lemma 3.2.6 applies.

Case CF-Abstr: Suppose the derivation ends with an abstraction

$$\frac{\vdash A\ \mathsf{type} \qquad \mathsf{a}^A \notin \mathsf{fv}(\mathcal{J}) \qquad \vdash \mathcal{J}[\mathsf{a}^A/x]}{\vdash \{x{:}A\}\ \mathcal{J}}$$

Without loss of generality we may assume that $\mathsf{a}^{I_*A} \notin \mathsf{fv}(I_*\mathcal{J})$. (If not, rename a to a fresh symbol.) We may apply the induction hypotheses to both premises and get

$$\vdash I_*A\ \mathsf{type} \qquad \text{and} \qquad \vdash (I_*\mathcal{J})[\mathsf{a}^{I_*A}/x].$$

and derive the desired judgement $\vdash \{x{:}I_* A\}\ I_* \mathcal{J}$ by abstracting $\mathsf{a}^{I_* A}$ in the right-hand judgement. □

3.2.2 Metatheorems about context-free finitary theories

The context-free economic rules for finitary theories carry over to the context-free setting. The proofs are analogous to those of Section 2.2.2 so we omit them.

Proposition 3.2.9. *[Economic version of Definition 3.1.7] Let R be the context-free raw rule $\Xi \Longrightarrow \mathcal{b}\boxed{e}$ with $\Xi = [\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n}]$ such that $\vdash \mathcal{b}$ is derivable, in particular R may be finitary. Then for any instantiation $I = [\mathsf{M}_1^{\mathcal{B}_1} \mapsto e_1, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \mapsto e_n]$, the following closure rule is admissible:*

CF-Specific-Eco

$$\frac{\vdash (I_{(i)*}\mathcal{B}_i)\boxed{e_i} \quad \textit{for } i = 1, \ldots, n}{\vdash I_*(\mathcal{b}\boxed{e})}$$

Proposition 3.2.10 (Economic version of Definition 3.1.10)**.** *In a context-free raw type theory, if $\mathcal{B} = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \mathcal{b}$, $\vec{s}$, and $\vec{t}$ have well-typed annotations, then the following closure rules are admissible:*

CF-Meta-Eco

$$\frac{\vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \textit{for } j = 1, \ldots, m}{\vdash (\mathcal{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}(\vec{t})}}$$

CF-Meta-Congr-Eco

$$\frac{\vdash s_j \equiv t_j : A_j[\vec{s}_{(j)}/\vec{x}_{(j)}] \quad \textit{for } j = 1, \ldots, m}{\vdash (\mathcal{b}[\vec{s}/\vec{x}])\boxed{\mathsf{M}_k(\vec{s}) \equiv \mathsf{M}_k(\vec{t})}}$$

3.2.3 Metatheorems about context-free standard theories

Inversion and uniqueness of typing (Theorems 2.2.22 and 2.2.24) carry over to context-free finitary theories. First, the notion of natural type is simpler, as it does not depend on the context anymore.

Definition 3.2.11. Let T be a finitary type theory. The *natural type* $\tau(t)$ of a term expression t is defined by:

$$\begin{aligned}
\tau(\mathsf{a}^A) &= A, \\
\tau(\mathsf{M}^{\mathcal{B}}(t_1, \ldots, t_m)) &= A[t_1/x_1, \ldots, t_m/x_m] \\
&\qquad \text{where } \mathcal{B} = (\{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \square : A) \\
\tau(\mathsf{S}(e_1, \ldots, e_n)) &= \langle \mathsf{M}_1 \mapsto e_1, \ldots, \mathsf{M}_n \mapsto e_n \rangle_* B \\
&\qquad \text{where the symbol rule for } \mathsf{S} \text{ is} \\
&\qquad \mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow \square : B \\
\tau(\kappa(t, \alpha)) &= \tau(t)
\end{aligned}$$

Next, we define an operation which peels conversions off a term, and another one that collects the peeled assumptions sets. We shall use these in the formulation of the context-free inversion theorem.

Definition 3.2.12. The ***conversion-stripping*** $\mathfrak{s}(t)$ of a term expression t is defined by:

$$\mathfrak{s}(t) = \begin{cases} \mathfrak{s}(t') & \text{if } t = \kappa(t', \alpha), \\ t & \text{otherwise.} \end{cases}$$

The ***conversion-residue*** $\mathfrak{r}(t)$ is defined by

$$\mathfrak{r}(t) = \begin{cases} \alpha \cup \mathfrak{r}(t') & \text{if } t = \kappa(t', \alpha), \\ \{\!\!\}\ & \text{otherwise.} \end{cases}$$

Note that $\lfloor t \rfloor = \lfloor \mathfrak{s}(t) \rfloor$ and that $\mathsf{asm}(t) = \mathsf{asm}(\mathfrak{s}(t), \mathfrak{r}(t))$.

Lemma 3.2.13. *If a context-free standard type theory derives* $\vdash t : A$ *then*

1. *it derives* $\vdash \mathfrak{s}(t) : \tau(t)$ *by an application of* CF-Var, CF-Meta, *or an instantiation of a term symbol rule, and*
2. *it derives* $\vdash \tau(t) \equiv A$ by $\mathfrak{r}(t)$.

Proof. We proceed by induction on the derivation of $\vdash t : A$.

Cases CF-Var, CF-Meta, *and symbol rule* : In these cases $t = \mathfrak{s}(t)$ and $\tau(t) = A$, so we already have $\vdash \mathfrak{s}(t) : \tau(t)$, while $\vdash \tau(t) \equiv A$ by $\{\!\!\}$ holds by reflexivity.

Case CF-Conv-Tm: Consider a derivation ending with a conversion

$$\frac{\vdash t : B \qquad \vdash B \equiv A \text{ by } \beta}{\vdash \kappa(t, \alpha) : A}$$

where $\mathsf{asm}(t, B, A, \beta) = \mathsf{asm}(t, A, \alpha)$. By induction hypothesis for the first premise we obtain $\vdash \mathfrak{s}(t) : \tau(t)$ and $\vdash \tau(t) \equiv B$ by $\mathfrak{r}(t)$, derived by one of the desired rules. Because $\mathfrak{s}(t) = \mathfrak{s}(\kappa(t, \alpha))$ and $\tau(t) = \tau(\kappa(t, \alpha))$, the first claim is established. For the second one, we apply CF-EqTy-Trans like this:

$$\frac{\vdash \tau(t) \equiv B \text{ by } \mathfrak{r}(t) \qquad \vdash B \equiv A \text{ by } \beta}{\vdash \tau(t) \equiv A \text{ by } \mathfrak{r}(t) \cup \alpha}$$

Suitability of $\mathfrak{r}(t) \cup \alpha$ is implied by $\mathsf{asm}(\tau(t), \mathfrak{r}(t)) = \mathsf{asm}(t)$:

$$\begin{aligned} \mathsf{asm}(\tau(t), B, \mathfrak{r}(t), A, \beta) &= \mathsf{asm}(t, B, A, \beta) \\ &= \mathsf{asm}(t, A, \alpha) \\ &= \mathsf{asm}(\tau(t), A, \mathfrak{r}(t), \alpha). \end{aligned}$$

□

Theorem 3.2.14 (Context-free inversion)**.** *If a context-free standard type theory derives* $\vdash t : A$*, then:*

- *if* $A = \tau(t)$*, it derives* $\vdash \mathscr{s}(t) : \tau(t)$ *by a derivation which concludes with* CF-Var*,* CF-Meta*, or an instantiation of a term symbol rule;*
- *if* $A \neq \tau(t)$*, it derives* $\vdash \kappa(\mathscr{s}(t), \mathscr{z}(t)) : A$ *by* CF-Conv-Tm*.*

Proof. Apply Lemma 3.2.13 and, depending on whether $A = \tau(t)$, either use $\vdash \mathscr{s}(t) : \tau(t)$ so obtained directly or convert it along $\vdash \tau(t) \equiv A$ by $\mathscr{z}(t)$, observing that the side condition $\mathsf{asm}(\mathscr{s}(t), \tau(t), A, \mathscr{z}(t)) = \mathsf{asm}(\mathscr{s}(t), \mathscr{z}(t), A)$ holds because $\mathsf{asm}(\tau(t)) \subseteq \mathsf{asm}(t) = \mathsf{asm}(\mathscr{s}(t), \mathscr{z}(t))$. □

Theorem 3.2.15 (Context-free uniqueness of typing)**.** *For a context-free standard type theory:*

1. *If* $\vdash t : A$ *and* $\vdash t : B$*, then* $\vdash A \equiv B$ by α *for some assumption set* α*.*
2. *If* $\vdash s \equiv t : A$ by β_1 *and* $\vdash s \equiv t : B$ by β_2*, with well-typed variables, then* $\vdash A \equiv B$ by α *for some assumption set* α*.*

In both cases, $\alpha \subseteq \mathsf{asm}(t)$ *can be computed from the judgements involved, without recourse to their derivations.*

Proof. The first statement holds because A and B are both judgmentally equal to the natural type of t by Lemma 3.2.13. The second statement reduces to the first one because the presuppositions $\vdash t : A$ and $\vdash t : B$ are derivable by Theorem 3.2.5. □

3.2.4 Special metatheorems about context-free theories

We prove several metatheorems which are specific to context-free type theories. The example of the equality reflection rule in the beginning of Section 3.1 showcased that finitary type theories do not enjoy strengthening. Context-free type theories, however, do satisfy this meta-property.

Theorem 3.2.16 (Strengthening)**.** *If a context-free raw type theory derives*

$$\vdash \{\vec{y}{:}\vec{B}\}\{x{:}A\}\ \mathcal{J}$$

and $x \notin \mathsf{bv}(\mathcal{J})$ *then it also derives* $\vdash \{\vec{y}{:}\vec{B}\}\ \mathcal{J}$*.*

Proof. We proceed by induction on the derivation of $\{\vec{y}{:}\vec{B}\}\{x{:}A\}\ \mathcal{J}$. The only case to consider is CF-Abstr. If the outer abstraction is empty, then the derivation ends with the abstraction

$$\frac{\vdash A\ \mathsf{type} \qquad \mathsf{a}^A \notin \mathsf{fv}(\mathcal{J}) \qquad \vdash \mathcal{J}[\mathsf{a}^A/x]}{\vdash \{x{:}A\}\ \mathcal{J}} \tag{3.7}$$

Because $x \notin \mathsf{bv}(\mathcal{J})$, it follows that $\mathsf{a}^A \notin \mathsf{fv}_0(\mathcal{J}[\mathsf{a}^A/x])$ and that $\mathcal{J}[\mathsf{a}^A/x] = \mathcal{J}$, which is the second premise, hence derivable. The other possibility is that the derivation ends with

$$\frac{\vdash A\ \mathsf{type} \qquad \mathsf{c}^C \notin \mathsf{fv}(\{\vec{y}{:}\vec{B}\}\{x{:}A\}\ \mathcal{J}) \qquad \vdash \{\vec{y}{:}\vec{B}[\mathsf{c}^C/z]\}\{x{:}A[\mathsf{c}^C/z]\}\ \mathcal{J}[\mathsf{c}^C/z]}{\vdash \{z{:}C\}\{\vec{y}{:}\vec{B}\}\{x{:}A\}\ \mathcal{J}}$$

From $x \notin \mathsf{bv}(\mathcal{J})$ it follows that $x \notin \mathsf{bv}(\mathcal{J}[\mathsf{c}^C/z])$, hence we may apply the induction hypothesis to the second premise and conclude by abstracting c^C. □

Why cannot we adapt the above proof to type theories with contexts? In the derivation (3.7), the second premise turns out to be precisely the desired conclusion, whereas TT-Abstr would yield $\Theta; \Gamma, \mathsf{a}{:}A \vdash \mathcal{J}$ where $\Theta; \Gamma \vdash \mathcal{J}$ is needed. Indeed, strengthening is not generally valid for type theories with contexts.

The next lemma can be used to modify the head of a judgement so that it fits another boundary, as long as there is agreement up to erasure.

Lemma 3.2.17 (Boundary conversion)**.** *In a context-free raw theory, if* $\vdash \mathcal{B}_1$, $\vdash \mathcal{B}_2$, $\vdash \mathcal{B}_1\boxed{e_1}$ *and* $\lfloor \mathcal{B}_1 \rfloor = \lfloor \mathcal{B}_2 \rfloor$ *then there is* e_2 *such that* $\vdash \mathcal{B}_2\boxed{e_2}$, $\mathsf{asm}(e_2) \subseteq \mathsf{asm}(\mathcal{B}_1\boxed{e_1})$ *and* $\lfloor e_1 \rfloor = \lfloor e_2 \rfloor$.

Proof. We proceed by induction on the derivation of $\vdash \mathcal{B}_1$.

Case CF-Bdry-Ty: If $\mathcal{B}_1 = (\Box\ \mathsf{type})$ then $\mathcal{B}_2 = (\Box\ \mathsf{type})$ and we may take $e_2 = e_1$.

Case CF-Bdry-Tm: If $\mathcal{B}_1 = (\Box : A_1)$ then $\mathcal{B}_2 = (\Box : A_2)$ and $\lfloor A_1 \rfloor = \lfloor A_2 \rfloor$, therefore $\vdash A_1 \equiv A_2$ by $\{\!\!\{\}\!\!\}$ by CF-EqTy-Refl. We may take $e_2 = \kappa(e_1, \mathsf{asm}(A_1) \setminus \mathsf{asm}(A_2))$ and derive $\vdash e_2 : A_2$ by CF-Conv-Tm.

Case CF-Bdry-EqTy: If $\mathcal{B}_1 = (A_1 \equiv B_1$ by $\Box)$ then $\mathcal{B}_2 = (A_2 \equiv B_2$ by $\Box)$, $\lfloor A_1 \rfloor = \lfloor A_2 \rfloor$ and $\lfloor B_1 \rfloor = \lfloor B_2 \rfloor$. By CF-EqTy-Refl we obtain $\vdash A_2 \equiv A_1$ by $\{\!\!\{\}\!\!\}$ and $\vdash B_1 \equiv B_2$ by $\{\!\!\{\}\!\!\}$. We take $e_2 = (e_1 \cup \mathsf{asm}(A_1) \cup \mathsf{asm}(B_1)) \setminus (\mathsf{asm}(A_2) \cup \mathsf{asm}(B_2))$ and derive $\vdash A_2 \equiv B_2$ by e_2 by two applications of CF-EqTy-Trans.

Case CF-Bdry-EqTm: If $\mathcal{B}_1 = (s_1 \equiv t_1 : A_1$ by $\Box)$ then $\mathcal{B}_2 = (s_2 \equiv t_2 : A_2$ by $\Box)$, $\lfloor s_1 \rfloor = \lfloor s_2 \rfloor$, $\lfloor t_1 \rfloor = \lfloor t_2 \rfloor$ and $\lfloor A_1 \rfloor = \lfloor A_2 \rfloor$. By CF-EqTy-Refl we obtain $\vdash A_1 \equiv A_2$ by $\{\!\!\{\}\!\!\}$, then by CF-Conv-EqTm

$$\vdash \kappa(s_1, \gamma) \equiv \kappa(t_1, \delta) : A_2 \text{ by } e_1$$

where $\gamma = \mathsf{asm}(A_1) \setminus \mathsf{asm}(s_1, A_2)$ and $\delta = \mathsf{asm}(A_1) \setminus \mathsf{asm}(t_1, A_2)$. Next, by reflexivity

$$\vdash s_2 \equiv \kappa(s_1, \gamma) : A_2 \text{ by } \{\!\!\{\}\!\!\}$$
$$\vdash \kappa(t_1, \delta) \equiv t_2 : A_2 \text{ by } \{\!\!\{\}\!\!\}$$

We may chain these together by transitivity to derive

$$\vdash s_2 \equiv t_2 : A_2 \text{ by } e_2$$

where $e_2 = \mathsf{asm}(e_1, s_1, t_1, A_1) \setminus \mathsf{asm}(s_2, t_2, A_2)$.

Case CF-Bdry-Abstr: If $\mathcal{B}_1 = (\{x{:}A_1\}\ \mathcal{B}_1')$ then $e_1 = \{x\}e_1'$, $\mathcal{B}_2 = \{x{:}A_2\}\ \mathcal{B}_2'$, $\lfloor A_1 \rfloor = \lfloor A_2 \rfloor$, and $\lfloor \mathcal{B}_1' \rfloor = \lfloor \mathcal{B}_2' \rfloor$. There is $\mathsf{a}^{A_2} \notin \mathsf{fv}(\mathcal{B}_2')$ such that $\vdash \mathcal{B}_2'[\mathsf{a}^{A_2}/x]$. We may apply Lemma 3.2.2 to $\vdash \{x{:}A_1\}\ \mathcal{B}_1'\boxed{e_1'}$ and $\vdash \kappa(\mathsf{a}^{A_2}, \{\!\!\{\}\!\!\}) : A_1$ to derive

$$\vdash (\mathcal{B}_1'[\kappa(\mathsf{a}^{A_2}, \{\!\!\{\}\!\!\})/x])\boxed{e_1'[\kappa(\mathsf{a}^{A_2}, \{\!\!\{\}\!\!\})/x]}.$$

By CF-BDRY-SUBST we have $\vdash \mathcal{B}_1'[\kappa(\mathsf{a}^{A_2}, \{\!\!\}\!\!\})/x]$, hence we may apply the induction hypothesis to obtain e_2'' such that $\lfloor e_2'' \rfloor = \lfloor e_1'[\kappa(\mathsf{a}^{A_2}, \{\!\!\}\!\!\})/x] \rfloor$, $\mathsf{asm}(e_2'') \subseteq \mathsf{asm}(\mathcal{B}_1'[\kappa(\mathsf{a}^{A_2}, \{\!\!\}\!\!\})/x])$, and $\vdash (\mathcal{B}_2'[\mathsf{a}^{A_2}/x])\boxed{e_2''}$. Set $e_2' = e_2''[x/\mathsf{a}^{A_2}]$ and apply CF-BDRY-ABSTR to derive $\vdash \{x{:}A_2\}\ \mathcal{B}_2'\boxed{e_2'}$. Thus we may take $e_2 = \{x\}e_2'$. □

3.3 A correspondence between theories with and without contexts

We now establish a correpondence between finitary type theories with and without contexts. We use the prefixes "tt" (for "traditional types") and "cf" (for "context-free") to disambiguate between the two versions of type theory. Thus the raw tt-syntax is the one from Fig. 2.1, and the raw cf-syntax the one from Fig. 3.1.

To ease the translation between the two versions of type theory, we shall use annotated free variables a^A and annotated metavariables $\mathsf{M}^{\mathcal{B}}$ in both version of raw syntax, where the annotations A and $\mathcal{B}$ are those of the cf-syntax. In the tt-syntax these annotations are considered part of the symbol names, and do not carry any type-theoretic significance.

3.3.1 Translation from cf-theories to tt-theories

We first show how to translate constituents of cf-theories to corresponding constituents of tt-theories. The plan is simple enough: move the annotations to contexts, elide the conversion terms, and replace the assumption sets with the dummy value.

The first step towards the translation was taken in Section 3.1.1.4, where we defined the erasure operation taking a cf-expression e to a tt-expression $\lfloor e \rfloor$ by removing conversions and replacing assumptions sets with the dummy value. Note that erasure and substitution commute, $\lfloor e[t/x] \rfloor = \lfloor e \rfloor[\lfloor t \rfloor/x]$, by an induction on the syntactic structure of e.

Next, in order to translate cf-judgements to tt-judgements, we need to specify when a context correctly encodes the information provided by cf-annotations.

Definition 3.3.1. We say that Θ is a ***suitable metavariable context*** for a set of cf-metavariables S when $S \subseteq |\Theta|$ and $\Theta(\mathsf{M}^{\mathcal{B}}) = \lfloor \mathcal{B} \rfloor$ for all $\mathsf{M}^{\mathcal{B}} \in S$. Similarly, Γ is a ***suitable variable context*** for a set of free cf-variables V when $V \subseteq |\Gamma|$ and $\Gamma(\mathsf{a}^A) = \lfloor A \rfloor$ for all $\mathsf{a}^A \in V$. We say that $\Theta;\Gamma$ is a ***suitable context*** for S and V when Θ is suitable for S an Γ for V.

As a shorthand, we say that $\Theta;\Gamma$ is *suitable* for a syntactic entity e when it is suitable for $\mathsf{mv}(e)$ and $\mathsf{fv}(e)$. As suitability only depends on the assumption set, it follows from suitability of $\Theta;\Gamma$ for e and $\mathsf{asm}(e') \subseteq \mathsf{asm}(e)$ that $\Theta;\Gamma$ is also suitable for e'.

Next, say that a free cf-variable a^A ***depends*** on a free cf-variable b^B, written $\mathsf{b}^B \prec \mathsf{a}^A$, when $\mathsf{b}^B \in \mathsf{fv}(A)$, and that a set S of free cf-variables is ***closed under dependence*** when $\mathsf{b}^B \prec \mathsf{a}^A \in S$ implies $\mathsf{b}^B \in S$. Every set S of cf-variables is

contained in the least closed set, which is $\bigcup\{\!|\mathsf{fv}(\mathsf{a}^A) \mid \mathsf{a}^A \in S|\!\}$. We similarly define dependence for cf-metavariables.

The following lemma shows how to construct suitable contexts.

Lemma 3.3.2. *For every finite set of cf-metavariables S there exists a suitable metavariable context Θ, such that $|\Theta|$ is the closure of S with respect to dependence. For every finite set of free cf-variables V there exists a suitable variable context Γ, such that $|\Gamma|$ is the closure of V with respect to dependence.*

Proof. Given a finite set of free cf-variables S, the well-founded order $\prec$ on $\bigcup\{\!|\mathsf{fv}(\mathsf{a}^A) \mid \mathsf{a}^A \in S|\!\}$ may be extended to a total one, say $\mathsf{a}_1^{A_1}, \ldots, \mathsf{a}_n^{A_n}$. Now take Γ to be the variable context $\mathsf{a}_1^{A_1} : \lfloor A_1 \rfloor, \ldots, \mathsf{a}_n^{A_n} : \lfloor A_n \rfloor$. The argument for metavariables is analogous. □

A totally ordered extension of $\prec$ can be given explicitly, so the preceding proof yields an explicit construction of a suitable contexts. Notice that the construction does not introduce any spurious assumptions, in the sense that for a variable context Γ the constructed suitable set V contains only the variables appearing in Γ and the annotations of types appearing in Γ.

Proposition 3.3.3. *If $\Theta; \Gamma$ is suitable for a cf-judgement $\mathcal{J}$ then $\Theta; \Gamma \vdash \lfloor \mathcal{J} \rfloor$ is a syntactically valid tt-judgement, and similarly for boundaries.*

Proof. A straightforward induction on the structrure of the judgement $\mathcal{J}$. □

Next we translate rules, theories, and derivations.

Proposition 3.3.4. *A cf-rule and a cf-rule-boundary*

$$\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow j \qquad \textit{and} \qquad \mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow \mathscr{b}$$

respectively translate to the raw tt-rule and the tt-rule-boundary

$$\mathsf{M}_1^{\mathcal{B}_1}{:}\lfloor \mathcal{B}_1 \rfloor, \ldots, \mathsf{M}_n^{\mathcal{B}_n}{:}\lfloor \mathcal{B}_n \rfloor \Longrightarrow \lfloor j \rfloor$$

and

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

A raw-cf theory $T = \langle R_i \rangle_{i \in I}$ over a signature Σ is thus translated rule-wise to the raw tt-theory $T_{\mathrm{tt}} = \langle (R_i)_{\mathrm{tt}} \rangle_{i \in I}$ over the same signature.

Proof. The conditions in Definition 3.1.1 guarantee that $\mathsf{M}_1^{\mathcal{B}_1}{:}\lfloor \mathcal{B}_1 \rfloor, \ldots, \mathsf{M}_n^{\mathcal{B}_n}{:}\lfloor \mathcal{B}_n \rfloor$ is a metavariable context and that it is suitable for $\lfloor j \rfloor$ and $\lfloor \mathscr{b} \rfloor$. □

Theorem 3.3.5 (Translation from finitary cf- to tt-theories)**.**

1. *The translation of a finitary cf-theory is finitary.*

2. *Suppose T is a finitary cf-theory whose translation T_{tt} is also finitary. Let $\Theta;\Gamma$ be tt-context such that $\vdash_{T_{\mathrm{tt}}} \Theta\ \mathsf{mctx}$ and $\Theta \vdash_{T_{\mathrm{tt}}} \Gamma\ \mathsf{vctx}$. If $\vdash_T \mathcal{J}$ and $\Theta;\Gamma$ is suitable for $\mathcal{J}$, then $\Theta;\Gamma \vdash_{T_{\mathrm{tt}}} \lfloor \mathcal{J} \rfloor$.*

3. *With T, $\Theta;\Gamma$ as in (2), if $\vdash_T \mathcal{B}$ and $\Theta;\Gamma$ is suitable for $\mathcal{B}$ then $\Theta;\Gamma \vdash_{T_{\mathrm{tt}}} \lfloor \mathcal{B} \rfloor$.*

Proof. We proceed by mutual structural induction on all three statements.

To prove statement (1), consider a finitary cf-theory $T = (R_i)_{i \in I}$, and let $(I, \prec)$ be a well-founded order witnessing the finitary character of T (Definition 3.1.13). We prove that T_{tt} is finitary with respect to $(I, \prec)$ by a well-founded induction on the order. Given any $i \in I$, with

$$R_i = (\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow j),$$

let $\Theta = [\mathsf{M}_1^{\mathcal{B}_1}{:}\lfloor \mathcal{B}_1 \rfloor, \ldots, \mathsf{M}_n^{\mathcal{B}_n}{:}\lfloor \mathcal{B}_n \rfloor]$. We verifty that $(R_i)_{\mathrm{tt}} = (\Theta \Longrightarrow \lfloor j \rfloor)$ is finitary in $T' = ((R_j)_{j \prec i})_{\mathrm{tt}}$ as follows:

- $\vdash_{T'} \Theta\ \mathsf{mctx}$ holds by induction on $k = 1, \ldots, n$: assuming $\vdash_{T'} \Theta_{(k)}\ \mathsf{mctx}$ has been established, apply (2) to a cf-derivation of $\vdash_{(R_i)_{j \prec i}} \mathcal{B}_k$ and the suitable context $\Theta_{(k)}; [\,]$.

- $\vdash_{T'} \lfloor j \rfloor$ holds by application of (2) to a cf-derivation of $\vdash_{(R_i)_{j \prec i}} j$ and the suitable context $\Theta; [\,]$.

We next address statement (2), which we prove by structural induction on the derivation of $\vdash_T \mathcal{J}$.

Case CF-Var: A cf-derivation ending with the variable rule

$$\frac{}{\vdash_T \mathsf{a}^A : A}$$

is translated to an application of TT-Var

$$\frac{\mathsf{a}^A \in |\Gamma|}{\Theta;\Gamma \vdash_{T_{\mathrm{tt}}} \mathsf{a}^A : \lfloor A \rfloor}$$

By suitability of Γ the side-condition $\mathsf{a}^A \in |\Gamma|$ is satisfied, and $\Gamma(\mathsf{a}^A) = \lfloor A \rfloor$.

Case CF-Meta: Consider a cf-derivation ending in

$$\frac{\begin{array}{l}\vdash_T t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] \quad \text{for } i = 1, \ldots, n \\ \vdash_T \mathcal{b}[\vec{t}/\vec{x}]\end{array}}{\vdash (\mathcal{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}^{\mathcal{B}}(\vec{t})}}$$

Because erasure commutes with substitution we have

$$\begin{aligned}
\lfloor A_i[\vec{t}_{(i)}/\vec{x}_{(i)}] \rfloor &= \lfloor A_i \rfloor [\lfloor \vec{t}_{(i)} \rfloor / \vec{x}_{(i)}], \\
\lfloor \mathcal{b}[\vec{t}/\vec{x}] \rfloor &= \lfloor \mathcal{b} \rfloor [\lfloor \vec{t} \rfloor / \vec{x}], \\
\lfloor (\mathcal{b}\boxed{\mathsf{M}^{\{\vec{x}:\vec{A}\}\mathcal{b}}(\vec{x})})[\vec{t}/\vec{x}] \rfloor &= (\lfloor \mathcal{b} \rfloor \boxed{\lfloor \mathsf{M}^{\{\vec{x}:\vec{A}\}\mathcal{b}}(\vec{x}) \rfloor})[\lfloor \vec{t} \rfloor / \vec{x}].
\end{aligned}$$

Applying TT-Meta to the translation of the premises obtained by the induction hypothesis thus yields the desired result. Suitability of $\Theta;\Gamma$ is ensured because all premises are recorded in the conclusion.

Cases CF-Meta-Congr-Ty *and* CF-Meta-Congr-Tm: We spell out the translation of the latter rule, where $\mathcal{B} = \{x_1{:}A_1\} \cdots \{x_m{:}A_m\}\ \Box : B$:

$$\frac{\begin{array}{ll} \vdash s_k : A_k[\vec{s}_{(k)}/\vec{x}_{(k)}] & \text{for } k = 1,\ldots,m \\ \vdash t_k : A_k[\vec{t}_{(k)}/\vec{x}_{(k)}] & \text{for } k = 1,\ldots,m \\ \lfloor t_k \rfloor = \lfloor t'_k \rfloor & \text{for } k = 1,\ldots,m \\ \vdash s_k \equiv t'_k : A[\vec{s}_{(k)}/\vec{x}_{(k)}] \text{ by } \alpha_k & \text{for } k = 1,\ldots,m \end{array} \quad \vdash v : B[\vec{s}/\vec{x}] \qquad \lfloor \mathsf{M}^{\mathcal{B}}(\vec{t}) \rfloor = \lfloor v \rfloor \qquad \beta \text{ suitable}}{\vdash \mathsf{M}^{\mathcal{B}}(\vec{s}) \equiv v : B[\vec{s}/\vec{x}] \text{ by } \beta} \tag{3.8}$$

The context $\Theta;\Gamma$ is suitable for the premises because β is suitable. We apply TT-Meta-Congr as follows:

$$\frac{\begin{array}{ll} \Theta;\Gamma \vdash \lfloor s_k \rfloor : \lfloor A_k \rfloor[\lfloor \vec{s} \rfloor_{(k)}/\vec{x}_{(k)}] & \text{for } k = 1,\ldots,m \\ \Theta;\Gamma \vdash \lfloor t_k \rfloor : \lfloor A_k \rfloor[\lfloor \vec{t} \rfloor_{(k)}/\vec{x}_{(k)}] & \text{for } k = 1,\ldots,m \\ \Theta;\Gamma \vdash \lfloor s_k \rfloor \equiv \lfloor t_k \rfloor : \lfloor A_k \rfloor[\lfloor \vec{s} \rfloor_{(k)}/\vec{x}_{(k)}] & \text{for } k = 1,\ldots,m \\ \Theta;\Gamma \vdash \lfloor B \rfloor[\lfloor \vec{s} \rfloor/\vec{x}] \equiv \lfloor B \rfloor[\lfloor \vec{t} \rfloor/\vec{x}] & \end{array}}{\Theta;\Gamma \vdash \mathsf{M}_k^{\mathcal{B}}(\lfloor \vec{s} \rfloor) \equiv \mathsf{M}_k^{\mathcal{B}}(\lfloor \vec{t} \rfloor) : \lfloor B \rfloor[\lfloor \vec{s} \rfloor/\vec{x}]}$$

The first two rows of premises are secured by the induction hypotheses for the corresponding rows in (3.8), and the premises in the third row are derivable by the side conditions in the third row and induction hypotheses for the fourth row. The last premise follows by Theorem 2.2.8 applied to $\Theta;\Gamma \vdash_{T_{tt}} \lfloor B \rfloor$ type, which holds because we assumed $\vdash_{T_{tt}} \Theta$ mctx.

Case CF-Abstr: A cf-derivation ending with an abstraction

$$\frac{\vdash_T A \text{ type} \qquad \mathsf{a}^A \notin \mathsf{fv}(\mathcal{J}) \qquad \vdash_T \mathcal{J}[\mathsf{a}^A/x]}{\vdash_T \{x{:}A\}\ \mathcal{J}}$$

is translated to a tt-derivation ending with TT-Abstr

$$\frac{\Theta;\Gamma \vdash_{T_{tt}} \lfloor A \rfloor \text{ type} \qquad \mathsf{b}^A \notin |\Gamma| \qquad \Theta;\Gamma,\mathsf{b}^A{:}\lfloor A \rfloor \vdash_{T_{tt}} \lfloor \mathcal{J} \rfloor[\mathsf{b}^A/x]}{\Theta;\Gamma \vdash_{T_{tt}} \{x{:}\lfloor A \rfloor\}\ \lfloor \mathcal{J} \rfloor}$$

The premises get their derivations from induction hypotheses, where $\mathsf{b}^A \notin |\Gamma|$ ensures that $\Gamma, \mathsf{b}^A{:}\lfloor A \rfloor$ is suitable for $\mathcal{J}[\mathsf{b}^A/x]$.

Case of a specific rule: Consider a derivation ending with an instantiation $I = \langle \mathsf{M}_1^{\mathcal{B}_1} {\mapsto} e_1, \ldots, \mathsf{M}_n^{\mathcal{B}^n} {\mapsto} e_n \rangle$ of a raw cf-rule $R = (\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}^n} \Longrightarrow \mathcal{b}\boxed{e})$:

$$\frac{\begin{array}{ll} \vdash_T (I_{(i)*}\mathcal{B}_i)\boxed{e_i} & \text{for i} = 1,\ldots,\text{n} \\ \vdash_T I_*\mathcal{b} & \end{array}}{\vdash_T I_*(\mathcal{b}\boxed{e})}$$

Let $\lfloor I \rfloor = \mathsf{M}_1^{\mathcal{B}_1} \mapsto \lfloor e_1 \rfloor, \ldots, \mathsf{M}_n^{\mathcal{B}^n} \mapsto \lfloor e_n \rfloor$. Because erasure commutes with instantiation we have

$$\lfloor (I_{(i)*}\mathcal{B}_i)\boxed{e_i} \rfloor = (\lfloor I \rfloor_{(i)*} \lfloor \mathcal{B}_i \rfloor)\boxed{\lfloor e_i \rfloor}$$

and $\lfloor I_*(\mathscr{b}\boxed{e}) \rfloor = \lfloor I \rfloor_* \lfloor \mathscr{b}\boxed{e} \rfloor$. Thus we may appeal to the induction hypotheses for the premises and conclude by R_{tt}, so long as we remember to check that $\Theta; \Gamma$ is suitable for the premises, which it is because Definition 3.1.1 of raw cf-rules requires $\mathsf{mv}(\mathscr{b}\boxed{e}) = \{\!\{\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}^n}\}\!\}$.

Case of a congruence rule: Consider an application of the congruence rule associated with a cf-rule

$$R = (\mathsf{M}_1^{\mathcal{B}_1}, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \Longrightarrow t : A),$$

as in Definition 3.1.8:

$$\frac{\begin{array}{ll} \vdash_T (I_{(i)*}\mathcal{B}_i)\boxed{f_i} & \text{for } i = 1, \ldots, n \\ \vdash_T (J_{(i)*}\mathcal{B}_i)\boxed{g_i} & \text{for } i = 1, \ldots, n \\ \lfloor g_i' \rfloor = \lfloor g_i \rfloor & \text{for object boundary } \mathcal{B}_i \\ \vdash_T (I_{(i)*}\mathcal{B}_i)\boxed{f_i \equiv g_i' \text{ by } \alpha_i} & \text{for object boundary } \mathcal{B}_i \\ \vdash_T t' : I_* A & \lfloor t' \rfloor = \lfloor J_* t \rfloor \\ & \beta \text{ suitable} \end{array}}{\vdash_T I_* t \equiv t' : I_* A \text{ by } \beta} \tag{3.9}$$

The context $\Theta; \Gamma$ is suitable for the premises because β is suitable. We apply the corresponding congruence for R_{tt} (Definition 2.1.13):

$$\frac{\begin{array}{ll} \Theta; \Gamma \vdash_{T_{\mathrm{tt}}} (\lfloor I \rfloor_{(i)*} \lfloor \mathcal{B}_i \rfloor)\boxed{\lfloor f_i \rfloor} & \text{for } i = 1, \ldots, n \\ \Theta; \Gamma \vdash_{T_{\mathrm{tt}}} (\lfloor J \rfloor_{(i)*} \lfloor \mathcal{B}_i \rfloor)\boxed{\lfloor g_i \rfloor} & \text{for } i = 1, \ldots, n \\ \Theta; \Gamma \vdash_{T_{\mathrm{tt}}} (\lfloor I \rfloor_{(i)*} \lfloor \mathcal{B}_i \rfloor)\boxed{\lfloor f_i \rfloor \equiv \lfloor g_i \rfloor} & \text{for object boundary } \mathcal{B}_i \\ \Theta; \Gamma \vdash_{T_{\mathrm{tt}}} \lfloor I \rfloor_* \lfloor A \rfloor \equiv \lfloor J \rfloor_* \lfloor A \rfloor & \end{array}}{\Theta; \Gamma \vdash_{T_{\mathrm{tt}}} \lfloor I \rfloor_* \lfloor t \rfloor \equiv \lfloor J \rfloor_* \lfloor t \rfloor : \lfloor I \rfloor_* \lfloor A \rfloor}$$

The first and the second row of premises are derivable by induction hypotheses for the corresponding rows in (3.9), while the third row is derivable because of the side conditions on the third row and induction hypotheses for the fourth row. The last premise follows by Theorem 2.2.17 applied to $\Theta; \Gamma \vdash_{T_{\mathrm{tt}}} A \ \mathsf{type}$, which in turn follows by induction hypothesis applied to a derivation of $\vdash_T A \ \mathsf{type}$ witnessing the finitary character of R.

Case CF-CONV-TM: Consider a term conversion

$$\frac{\vdash_T t : A \qquad \vdash_T A \equiv B \text{ by } \alpha}{\vdash_T \kappa(t, \beta) : B}$$

The side condition $\mathsf{asm}(t, A, B, \alpha) = \mathsf{asm}(t, B, \beta)$ ensures that $\Theta; \Gamma$ is suitable for both premises, hence we may apply the induction hypotheses to the premisess and conclude by TT-Conv-Tm.

Case CF-Conv-EqTm: Consider an equality conversion

$$\frac{\vdash s \equiv t : A \text{ by } \alpha \qquad \vdash A \equiv B \text{ by } \beta}{\vdash \kappa(s, \gamma) \equiv \kappa(t, \delta) : B \text{ by } \alpha}$$

The side conditions

$$\mathsf{asm}(s, A, B, \beta) = \mathsf{asm}(s, B, \gamma) \qquad \text{and} \qquad \mathsf{asm}(t, A, B, \beta) = \mathsf{asm}(t, B, \delta)$$

ensure that $\Theta; \Gamma$ is suitable for both premises, hence we may apply the induction hypotheses to the premises and conclude by TT-Conv-EqTm. As in the preceding case all assumptions in the premises already appear in the conclusion, and suitability is preserved.

Cases CF-EqTy-Refl, CF-EqTy-Sym, CF-EqTy-Trans, CF-EqTm-Refl, CF-EqTm-Sym, CF-EqTm-Trans: These all proceed by application of induction hypotheses to the premises, followed by the corresponding tt-rule, where crucially we rely on recording metavariables in the assumption sets to make sure that Θ and Γ are suitable for the premises.

Finally, we address statement (2), which is proved by structural induction on $\vdash_T \mathcal{B}$. The base cases CF-Bdry-Ty, CF-Bdry-Tm, CF-Bdry-EqTy, CF-Bdry-EqTm reduce to translation of term and type judgements, while the induction step CF-Bdry-Abstr is similar to the case CF-Abstr above. □

With the theorem in hand, the loose ends are easily tied up.

Corollary 3.3.6. *The translation of a standard cf-theory is a standard tt-theory.*

Proof. The translation takes symbol rules to symbol rules, and equality rules to equality rules. □

Corollary 3.3.7. *If a finitary cf-theory T derives $\vdash_T \mathcal{J}$ and $\mathcal{J}$ has well-typed annotations then there exists a context $\Theta; \Gamma$ which is suitable for $\mathcal{J}$ such that $\vdash_{T_{\mathrm{tt}}} \Theta$ mctx and $\Theta \vdash_{T_{\mathrm{tt}}} \Gamma$ vctx.*

Proof. We may use the suitable context $\Theta; \Gamma$ with Θ and Γ constructed respectively from $\mathsf{mv}(\mathcal{J})$ and $\mathsf{fv}(\mathcal{J})$ as in Lemma 3.3.2. □

3.3.2 Translation from tt-theories to cf-theories

Transformation from tt-theories to cf-theories requires annotation of variables with typing information, insertion of conversions, and reconstruction of assumption sets. Unlike in the previous section, we cannot directly translate judgements, but must look at derivations in order to tell where conversions should be inserted and what

assumption sets used. We begin by defining auxiliary notions that help organize the translation.

Given a cf-expression e, let $\llbracket e \rrbracket$ be the ***double erasure*** of e, which is like erasure $\lfloor e \rfloor$, except that we also remove annotations: $\llbracket \mathsf{M}^{\mathcal{B}} \rrbracket = \mathsf{M}$ and $\llbracket \mathsf{a}^{A} \rrbracket = \mathsf{a}$. The following definition specifies when an assignment of annotations to variables, which we call a *labeling*, meets the syntactic criteria that makes it eligible for a translation.

Definition 3.3.8.

1. Consider a metavariable context

$$\Theta = [\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_m{:}\mathcal{B}_m].$$

   An ***eligible labeling for*** Θ is a map

$$\theta = \langle \mathsf{M}_1 {\mapsto} \mathcal{B}'_1, \ldots, \mathsf{M}_m {\mapsto} \mathcal{B}'_m \rangle$$

   which assigns to each M_i a cf-boundary $\mathcal{B}'_i$ such that $\llbracket \mathcal{B}'_i \rrbracket = \mathcal{B}_i$, and if $\mathsf{M}_j^{\mathcal{B}} \in \mathsf{mv}(\mathcal{B}'_i)$ then $\mathcal{B} = \mathcal{B}'_j$.

2. With Θ and θ as above, consider a variable context

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

   over Θ. An ***eligible labeling for*** Γ with respect to θ is a map

$$\gamma = \langle \mathsf{a}_1 {\mapsto} A'_1, \ldots, \mathsf{a}_n {\mapsto} A'_n \rangle$$

   which assigns to each a_i a cf-type A'_i such that $\llbracket A'_i \rrbracket = A_i$, if $\mathsf{M}_j^{\mathcal{B}} \in \mathsf{mv}(A_i)$ then $\mathcal{B} = \theta(\mathsf{M}_j)$, and if $\mathsf{a}_k^{A} \in \mathsf{fv}(A_i)$ then $A = \gamma(\mathsf{a}_k)$.

3. A (θ, γ) is an ***eligible labeling for*** $(\Gamma; \Theta)$ when θ is eligible for Θ and γ is eligible for Γ with respect to θ.

4. With (θ, γ) eligible for $\Theta; \Gamma$, an ***eligible cf-judgement*** $\mathcal{J}'$ for a tt-judgement $\mathcal{J}$ over $\Theta; \Gamma$ is one that satisfies $\llbracket \mathcal{J}' \rrbracket = \mathcal{J}$, if $\mathsf{M}_i^{\mathcal{B}} \in \mathsf{mv}(\mathcal{J}')$ then $\mathcal{B} = \theta(\mathsf{M}_i)$, and if $\mathsf{a}_k^{A} \in \mathsf{fv}(\mathcal{J}')$ then $A = \gamma(\mathsf{a}_k)$.

5. With (θ, γ) eligible for $\Theta; \Gamma$, an ***eligible cf-boundary*** $\mathcal{B}'$ for a tt-boundary $\mathcal{B}$ over $\Theta; \Gamma$ is one that satisfies $\llbracket \mathcal{B}' \rrbracket = \mathcal{B}$, if $\mathsf{M}_i^{\mathcal{B}''} \in \mathsf{mv}(\mathcal{B}')$ then $\mathcal{B}'' = \theta(\mathsf{M}_i)$, and if $\mathsf{a}_k^{A} \in \mathsf{fv}(\mathcal{B}')$ then $A = \gamma(\mathsf{a}_k)$.

We also postulate eligibility requirements for raw rules and theories.

Definition 3.3.9. Consider a raw tt-rule

$$R = (\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n \Longrightarrow j).$$

An ***eligible raw cf-rule*** for R is a raw cf-rule

$$R' = (\mathsf{M}_1^{\mathcal{B}'_1}, \ldots, \mathsf{M}_n^{\mathcal{B}'_n} \Longrightarrow j')$$

such that $\theta = \langle \mathsf{M}_1 \mapsto \mathcal{B}'_1, \ldots, \mathsf{M}_n \mapsto \mathcal{B}'_n \rangle$ is eligible for $[\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_n{:}\mathcal{B}_n]$, and j' is eligible for j with respect to θ (and the empty labeling for $[\,]$).

Let $T = \langle R_i \rangle_{i \in I}$ be a raw tt-theory over Σ. An ***eligible raw cf-theory*** for T is a raw cf-theory $T' = \langle R'_i \rangle_{i \in I}$ over Σ such that each R'_i is eligible for R_i.

Theorem 3.3.10 (Translation of standard tt- to cf-theories)**.**

1. *For any standard tt-theory T there exists a standard cf-theory T' eligible for T.*
2. *For any T, T' as above, if $\vdash_T \Theta$ mctx then there exists an eligible labeling θ for Θ such that $\vdash_{T'} \theta(\mathsf{M})$ for every $\mathsf{M} \in |\Theta|$.*
3. *For any T, T', Θ, θ as above, if $\Theta; [\,] \vdash_T \Gamma$ vctx then there exists an eligible labeling γ for Γ with respect to θ such that $\vdash_{T'} \gamma(\mathsf{a})$ type for every $\mathsf{a} \in |\Gamma|$.*
4. *For any T, T', Θ, θ, Γ, γ as above, if $\Theta; \Gamma \vdash_T \mathcal{B}$ then there exists an eligible cf-boundary $\mathcal{B}'$ for $\mathcal{B}$ with respect to θ, γ such that $\vdash_{T'} \mathcal{B}'$.*
5. *For any T, T', Θ, θ, Γ, γ, as above, if $\Theta; \Gamma \vdash_T \mathcal{J}$ then there exists an eligible cf-judgement $\mathcal{J}'$ for $\mathcal{J}$ with respect to θ, γ such that $\vdash_{T'} \mathcal{J}'$.*

Proof. We prove the above existence statements by explicit constructions, e.g., we prove (1)) by constructing a specific T' which meets the criteria, and similarly for the remaining parts. We proceed by simultaneous structural induction on all the parts.

Proof of part (1): We proceed by induction on a well-founded order $(I, \prec)$ witnessing the finitary character of $T = (R_i)_{i \in I}$. Consider any $i \in I$, with the corresponding specific rule

$$R_i = (\Theta \Longrightarrow \mathscr{b}\boxed{e}),$$

and let $T_i = (R_j)_{j \prec i}$. By induction hypothesis the tt-theory T'_i eligible for T_i has been constructred. Because $\vdash_{T_i} \Theta$ mctx, by (2) there is an eligible labeling $\theta = \langle \mathsf{M}_1 \mapsto \mathcal{B}'_1, \ldots, \mathsf{M}_n \mapsto \mathcal{B}'_n \rangle$ for Θ such that $\vdash_{T'_k} \mathcal{B}'_k$ for each $k = 1, \ldots, n$. The empty map $\gamma = \langle\rangle$ is an eligible labeling for the empty context $[\,]$. Because $\Theta; [\,] \vdash_{T_i} \mathscr{b}$, by (4) there is an eligible cf-boundary $\mathscr{b}'$ for $\mathscr{b}$ with respect to θ, γ such that $\vdash_{T'_i} \mathscr{b}'$. We now are in possession of the cf-rule-boundary

$$\mathsf{M}_1^{\mathcal{B}'_1}, \ldots, \mathsf{M}_n^{\mathcal{B}'_n} \Longrightarrow \mathscr{b}' \tag{3.10}$$

eligible for the tt-rule-boundary $\Theta \Longrightarrow \mathscr{b}$. Let

$$R'_i = (\mathsf{M}_1^{\mathcal{B}'_1}, \ldots, \mathsf{M}_n^{\mathcal{B}'_n} \Longrightarrow \mathscr{b}'\boxed{e'})$$

be the symbol or equality cf-rule induced by (3.10), as in Definitions 3.1.5 and 3.1.6. Comparison with Definitions 2.1.10 and 2.1.12 shows that $\llbracket e' \rrbracket = e$, as required.

Proof of part (2): We proceed by induction on the derivation of $\vdash_T \Theta\ \mathsf{mctx}$. The empty map is an eligible labeling for the empty metavariable context. If $\vdash_T \langle\Theta, \mathsf{M}{:}\mathcal{B}\rangle\ \mathsf{mctx}$ then by inversion $\vdash_T \Theta\ \mathsf{mctx}$ and $\Theta; [\,] \vdash_T \mathcal{B}$. By induction hypothesis there exists an eligible labeling θ for Θ, and by (4) applied to T, T', Θ, θ, $[\,]$, $\langle\rangle$ a cf-boundary $\mathcal{B}'$ eligible for $\mathcal{B}$ such that $\vdash_{T'} \mathcal{B}'$. The map $\theta' = \langle\theta, \mathsf{M}{\mapsto}\mathcal{B}'\rangle$ is eligible for $\langle\Theta, \mathsf{M}{:}\mathcal{B}\rangle$, and moreover $\vdash_{T'} \theta'(\mathsf{M}')$ for every $\mathsf{M}' \in |\theta'|$.

Proof of part (3) is analogous to part (2).

Proof of part (4): The non-abstracted boundaries reduce to instances of (5) by inversion, while the case of TT-Bdry-Abstr is analogous to the case TT-Abstr below.

Part (5): Let $T, T', \Theta, \theta, \Gamma, \gamma$ be as in (5) with

$$\begin{aligned}
\Theta &= [\mathsf{M}_1{:}\mathcal{B}_1, \ldots, \mathsf{M}_m{:}\mathcal{B}_p],\\
\theta &= \langle \mathsf{M}_1{\mapsto}\mathcal{B}'_1, \ldots, \mathsf{M}_p{\mapsto}\mathcal{B}'_p\rangle,\\
\Gamma &= [\mathsf{a}_1{:}A_1, \ldots, \mathsf{a}_p{:}A_r],\\
\gamma &= \langle \mathsf{a}_1{\mapsto}A'_1, \ldots, \mathsf{a}_r{\mapsto}A'_r\rangle.
\end{aligned}$$

We have the further assumption that each M_i has a cf-derivation D_{M_i} of $\vdash_{T'} \mathcal{B}'_i$, and each a_j a cf-derivation D_{a_j} of $\vdash_{T'} A'_j\ \mathsf{type}$. We proceed by structural induction on the derivation of $\Theta; \Gamma \vdash_T \mathcal{J}$. In each case we construct a cf-derivation concluding with $\vdash_{T'} \mathcal{J}'$ such that $\mathcal{J}'$ is eligible for $\mathcal{J}$.

Case TT-Var: Consider a tt-derivation ending with the variable rule

$$\frac{}{\Theta; \Gamma \vdash_T \mathsf{a}_j : A_j}$$

The corresponding cf-derivation is the application of CF-Var

$$\frac{}{\vdash_{T'} \mathsf{a}_j^{A'_j} : A'_j}$$

Case TT-Meta: Consider a tt-derivation ending with the metavariable rule, where $\mathcal{B}_k = \{x_1{:}B_1\} \cdots \{x_m{:}B_m\}\ \mathscr{b}$ and $\mathcal{B}'_k = \{x_1{:}B'_1\} \cdots \{x_m{:}B'_m\}\ \mathscr{b}'$:

$$\frac{\begin{array}{l}\Theta; \Gamma \vdash_T t_j : B_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m\\ \Theta; \Gamma \vdash_T \mathscr{b}[\vec{t}/\vec{x}]\end{array}}{\Theta; \Gamma \vdash_T (\mathscr{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_k(\vec{t})}}$$

The correspond cf-derivation ends with and application of CF-Meta,

$$\frac{\begin{array}{l}\vdash_{T'} t'_j : B'_j[\vec{t}'_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1, \ldots, m\\ \vdash_{T'} \mathscr{b}'[\vec{t}'/\vec{x}]\end{array}}{\vdash_{T'} \mathscr{b}'\boxed{\mathsf{M}^{\mathcal{B}'}(\vec{t}')}}$$

where the cf-terms $\vec{t}' = (t'_1, \ldots, t'_m)$ are constructed inductively as follows. Assuming we already have $\vec{t}'_{(j)}$, we apply the induction hypothesis to the j-th premise and obtain its eligible counterpart $\vdash_{T'} t''_j : B''_j$, so that $\llbracket t''_j \rrbracket = t_j$ and $\llbracket B''_j \rrbracket = B_j[\vec{t}_{(j)}/\vec{x}_{(j)}]$. It follows that $\lfloor B''_j \rfloor = \lfloor B'_j[\vec{t}'_{(j)}/\vec{x}_{(j)}] \rfloor$, therefore we may use Lemma 3.2.17 to modify t''_j to a term t'_j which fills $B'_j[\vec{t}'_{(j)}/\vec{x}_{(j)}]$.

Case TT-Meta-Congr: We consider a tt-derivation ending with a metavariable term congruence rule, where

$$\mathcal{B}_k = \{x_1 : B_1\} \cdots \{x_m : B_m\} \, \Box : C \quad \text{and} \quad \mathcal{B}'_k = \{x_1 : B'_1\} \cdots \{x_m : B'_m\} \, \Box : C'$$

$$\frac{\begin{array}{ll} \Theta; \Gamma \vdash_T s_j : B_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta; \Gamma \vdash_T t_j : B_j[\vec{t}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta; \Gamma \vdash_T s_j \equiv t_j : B_j[\vec{s}_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \Theta; \Gamma \vdash_T C[\vec{s}/\vec{x}] \equiv C[\vec{t}/\vec{x}] & \end{array}}{\Theta; \Gamma \vdash_T \mathsf{M}_k(\vec{s}) \equiv \mathsf{M}_k(\vec{t}) : C[\vec{s}/\vec{x}]} \tag{3.11}$$

The corresponding cf-derivation ends with CF-Meta-Congr-Tm

$$\frac{\begin{array}{ll} \vdash_{T'} s'_j : B'_j[\vec{s}'_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \vdash_{T'} t'_j : B'_j[\vec{t}'_{(j)}/\vec{x}_{(j)}] & \text{for } j = 1, \ldots, m \\ \lfloor t'_k \rfloor = \lfloor t''_j \rfloor & \text{for } j = 1, \ldots, m \\ \vdash_{T'} s'_j \equiv t''_j : B'_j[\vec{s}'_{(j)}/\vec{x}_{(j)}] \text{ by } \alpha_j & \text{for } j = 1, \ldots, m \\ \vdash_{T'} v : C'[\vec{s}'/\vec{x}] & \lfloor \mathsf{M}^{\mathcal{B}}(\vec{t}') \rfloor = \lfloor v \rfloor \end{array}}{\vdash_{T'} \mathsf{M}^{\mathcal{B}}(\vec{s}') \equiv v : C'[\vec{s}'/\vec{x}] \text{ by } \beta} \tag{3.12}$$

where suitable $\vec{s}', \vec{t}', \vec{t}'', \vec{\alpha}, v$, and β remain to be constructed. The terms $\vec{s}'$ and $\vec{t}'$ are obtained as in the previous case, using the first two rows of premises of (3.11). The induction hypotheses for the third row give us judgements, for $j = 1, \ldots, m$,

$$\vdash_{T'} s''_j \equiv t''_j : B''_j$$

such that $\lfloor B''_j \rfloor = \lfloor B_j[\vec{s}'_{(j)}/\vec{x}_{(j)}] \rfloor$. We convert the above equality along $\vdash_{T'} B''_j \equiv B_j[\vec{s}'_{(j)}/\vec{x}_{(j)}]$ to derive

$$\vdash_{T'} s'''_j \equiv t'_j : B_j[\vec{s}'_{(j)}/\vec{x}_{(j)}]$$

and since $\lfloor s'''_j \rfloor = \lfloor s'_j \rfloor$ by reflexivity and transitivity

$$\vdash_{T'} s'_j \equiv t'_j : B_j[\vec{s}'_{(j)}/\vec{x}_{(j)}].$$

It remains to construct v and β. For the former, we apply CF-Subst-EqTy to $\vdash_{T'} \{\vec{x} : \vec{B}'\} \, C' \text{ type}$ to derive

$$\vdash_{T'} C'[\vec{s}'/\vec{x}] \equiv C'[\vec{t}'/\vec{x}] \text{ by } \delta$$

ands use it to convert $\vdash_{T'} \mathsf{M}_k(\vec{t}') : C'[\vec{t}'/\vec{x}]$ to $\vdash_{T'} \kappa(\mathsf{M}_k(\vec{t}'), \epsilon) : C'[\vec{s}'/\vec{x}]$ for a suitable ϵ. We take $v = \kappa(\mathsf{M}_k(\vec{t}'), \epsilon)$ and the minimal suitable β.

Case TT-ABSTR: Consider a tt-derivation ending with an abstraction

$$\frac{\Theta;\Gamma \vdash_T A \text{ type} \qquad \mathsf{a} \notin |\Gamma| \qquad \Theta;\Gamma,\mathsf{a}{:}A \vdash_T \mathcal{J}[\mathsf{a}/x]}{\Theta;\Gamma \vdash_T \{x{:}A\}\ \mathcal{J}}$$

By induction hypothesis we obtain a derivation of $\vdash_{T'} A'$ type which is eligible for the first premise. The extended map $\langle\gamma, \mathsf{a}{\mapsto}A'\rangle$ is eligible for $\Gamma, \mathsf{a}{:}A$, and so by induction hypothesis we obtain a derivble $\vdash_{T'} \mathcal{J}'$ which is eligible for the second premise with respect to $(\theta, \langle\gamma, \mathsf{a}{\mapsto}A'\rangle)$. We form the desired abstraction by CF-ABSTR,

$$\frac{\vdash_{T'} A' \text{ type} \qquad \mathsf{a}^{A'} \notin \mathsf{fv}(\mathcal{J}') \qquad \vdash_{T'} \mathcal{J}'}{\vdash_{T'} \{x{:}A'\}\ \mathcal{J}'[x/\mathsf{a}^{A'}]}$$

Case of a specific rule: Consider a specific tt-rule

$$R = (\mathsf{N}_1{:}\mathcal{D}_1, \ldots, \mathsf{N}_m{:}\mathcal{D}_m \Longrightarrow j),$$

and the corresponding cf-rule

$$R' = (\mathsf{N}_1{}^{\mathcal{D}'_1}, \ldots, \mathsf{N}_k{}^{\mathcal{D}'_m} \Longrightarrow j')$$

Consider a tt-derivation ending with $I_* R$ where $I = \langle \mathsf{N}_1{\mapsto}e_1, \ldots, \mathsf{N}_m \to e_m\rangle$:

$$\frac{\begin{array}{l}\Theta;\Gamma \vdash_T (I_{(j)*}\mathcal{D}_j)\boxed{e_j} \quad \text{for } j = 1, \ldots, m \\ \Theta;\Gamma \vdash_T I_* \mathscr{b}\end{array}}{\Theta;\Gamma \vdash_T I_*(\mathscr{b}\boxed{e})} \tag{3.13}$$

The corresponding cf-derivation is obtained by an application of R' instantiated with

$$I' = \langle \mathsf{N}_1{}^{\mathcal{D}'_1}{\mapsto}e'_1, \ldots, \mathsf{N}_k{}^{\mathcal{D}'_k}{\mapsto}e'_m\rangle,$$

which is constructed inductively as follows. Suppose $\vec{e'}_{(j)}$ have already been constructed in such a way that $\llbracket e'_k \rrbracket = e_k$ and $\vdash_{T'} (I'_{(k)*}\mathcal{D}'_k)\boxed{e'_k}$ for all $k < j$. The induction hypothesis for the j-th premise of (3.13) yields $\vdash_{T'} \mathcal{D}''_j\boxed{e''_j}$ such that $\lfloor \mathcal{D}''_j \rfloor = \lfloor I'_{(j)}\mathcal{D}'_j \rfloor$. We apply Lemma 3.2.17 to modify e''_j to e'_j such that $\vdash_{T'} (I'_{(j)}\mathcal{D}'_j)\boxed{e'_j}$ and $\llbracket e'_j \rrbracket = \lfloor e''_j \rfloor$. Lastly, the premise $\vdash_{T'} I'_* \mathscr{b}'$ is derivable because R' is finitary.

Case of a congruence rule: Consider a term tt-rule

$$R = (\mathsf{N}_1{:}\mathcal{D}_1, \ldots, \mathsf{N}_m{:}\mathcal{D}_m \Longrightarrow t : C),$$

and the corresponding cf-rule

$$R' = (\mathsf{N}_1^{\mathcal{D}'_1}, \ldots, \mathsf{N}_m^{\mathcal{D}'_m} \Longrightarrow t' : C'),$$

Given instantiations

$$I = \langle \mathsf{N}_1 {\mapsto} f_1, \ldots, \mathsf{N}_m \to f_m \rangle \quad \text{and} \quad J = \langle \mathsf{N}_1 {\mapsto} g_1, \ldots, \mathsf{N}_m \to g_m \rangle,$$

suppose the tt-derivation ends with the congruence rule for R:

$$\frac{\begin{array}{ll} \Theta; \Gamma \vdash_T (I_{(j)*}\mathcal{D}_j)\boxed{f_j} & \text{for } i = 1, \ldots, m \\ \Theta; \Gamma \vdash_T (J_{(j)*}\mathcal{D}_j)\boxed{g_j} & \text{for } i = 1, \ldots, m \\ \Theta; \Gamma \vdash_T (I_{(j)*}\mathcal{D}_j)\boxed{f_j \equiv g_i} & \text{for object boundary } \mathcal{D}_j \\ \Theta; \Gamma \vdash_T I_* C \equiv J_* C & \end{array}}{\Theta; \Gamma \vdash I_* t \equiv J_* t : I_* C} \tag{3.14}$$

The corresponding cf-derivation ends with the congruence rule for R',

$$\frac{\begin{array}{ll} \vdash_{T'} (I'_{(i)*}\mathcal{D}'_i)\boxed{f'_i} & \text{for } i = 1, \ldots, m \\ \vdash_{T'} (J'_{(i)*}\mathcal{D}'_i)\boxed{g'_i} & \text{for } i = 1, \ldots, m \\ \lfloor g''_i \rfloor = \lfloor g'_i \rfloor & \text{for object boundary } \mathcal{D}'_i \\ \vdash_{T'} (I'_{(i)*}\mathcal{D}'_i)\boxed{f'_i \equiv g''_i \text{ by } \alpha_i} & \text{for object boundary } \mathcal{D}'_i \\ \vdash_{T'} t'' : I'_* C' & \lfloor t'' \rfloor = \lfloor J'_* t' \rfloor \\ & \beta \text{ suitable} \end{array}}{\vdash_{T'} I'_* t' \equiv t'' : I'_* C \text{ by } \beta}$$

where

$$I' = \langle \mathsf{N}_1^{\mathcal{D}'_1} {\mapsto} f'_1, \ldots, \mathsf{N}_m^{\mathcal{D}'_m} \to f'_m \rangle \quad \text{and} \quad J' = \langle \mathsf{N}_1^{\mathcal{D}'_1} {\mapsto} g'_1, \ldots, \mathsf{N}_m^{\mathcal{D}'_m} \to g'_m \rangle.$$

It remains to determine $\vec{f}'$, $\vec{g}'$, $\vec{g}''$, and t''.

The terms $\vec{f}'$ and $\vec{g}'$ are constructed from the first two rows of premises of the tt-derivation in the same way as $\vec{e}'$ in the previous case. The third row of premises yields equations, which after an application of Lemma 3.2.17, take the form

$$\vdash_{T'} (I'_{(i)*}\mathcal{D}'_i)\boxed{f''_i \equiv g''_i \text{ by } \beta_i}.$$

As $\lfloor f'_i \rfloor = \lfloor f''_i \rfloor$, these can be rectified by reflexivity and transitivity to the desired form

$$\vdash_{T'} (I'_{(i)*}\mathcal{D}'_i)\boxed{f'_i \equiv g''_i \text{ by } \alpha_i}.$$

Finally, we construct t'' by converting $\vdash_{T'} J_* t' : J_* C$ along $\vdash_{T'} J'_* C' \equiv I'_* C'$ by γ, which is derived as follows. The induction hypothesis for the last premise of (3.14) gives

$$\vdash_{T'} C_1 \equiv C_2$$

such that $\lfloor C_1 \rfloor = \lfloor I'_* C' \rfloor$ and $\lfloor C_2 \rfloor = \lfloor J'_* C' \rfloor$. Because $\vdash_{T'} I'_* C' \ \mathsf{type}$ and $\vdash_{T'} J'_* C' \ \mathsf{type}$, as well as $\vdash_{T'} C_1 \ \mathsf{type}$ and $\vdash_{T'} C_2 \ \mathsf{type}$ by Theorem 3.2.5, we may adjust the above equation to

$$\vdash_{T'} I'_* C' \equiv J'_* C',$$

which is only a symmetry away from the desired one.

The case of a type specific rule is simpler and dealt with in a similar fashion.

Cases TT-EqTy-Refl, TT-EqTy-Sym, TT-EqTm-Refl, TT-EqTm-Sym: each of these is taken care of by applying the induction hypotheses to the premises, followed by application of the corresponding cf-rule.

Cases TT-EqTy-Trans *and* TT-EqTm-Trans: Consider a derivation ending with term transitivity

$$\frac{\Theta; \Gamma \vdash_T s \equiv t : A \qquad \Theta; \Gamma \vdash_T t \equiv u : A}{\Theta; \Gamma \vdash_T s \equiv u : A}$$

The induction hypotheses for the premises produce eligible judgements

$$\vdash_{T'} s' \equiv t' : A' \ \mathsf{by}\ \alpha \qquad \text{and} \qquad \vdash_{T'} t'' \equiv u'' : A'' \ \mathsf{by}\ \beta$$

Because $\lfloor A' \rfloor = \lfloor A'' \rfloor$ and $\lfloor t' \rfloor = \lfloor t'' \rfloor$, we may convert the second judgement to A', and rectify the left-hand side, which results in

$$\vdash_{T'} t' \equiv u' : A' \ \mathsf{by}\ \gamma.$$

Now CF-EqTm-Trans applies. The case of transitivity of type equality similar and easier.

Case TT-Conv-Tm: Consider a conversion

$$\frac{\Theta; \Gamma \vdash_T t : A \qquad \Theta; \Gamma \vdash_T A \equiv B}{\Theta; \Gamma \vdash_T t : B}$$

The induction hypotheses for the premises produce eligible judgements

$$\vdash_{T'} t'' : A' \qquad \text{and} \qquad \vdash_{T'} A'' \equiv B' \ \mathsf{by}\ \alpha$$

Because $\lfloor A' \rfloor = \lfloor A'' \rfloor$, we obtain $A' \equiv B' \ \mathsf{by}\ \beta$, after which CF-Conv-Tm can be used to convert $\vdash_{T'} t'' : A'$ to a judgement $\vdash_{T'} t' : B' \ \mathsf{by}\ \beta$ which is eligible for the conclusion.

Case TT-Conv-EqTm: This case follows the same pattern as the previous one. □

Sharpest ever view of the Andromeda Galaxy.
Source: Hubble Space Telescope.

Chapter 4

An effectful metalanguage for type theories

We present in this chapter the Andromeda metalanguage (AML), an effectful metalanguage designed to support the user in the construction of judgements in a context-free standard type theory of their choice. AML assists the user in two ways, both of which are inspired by techniques that have a long tradition in proof assistants for type theory. First, given a context-free type theory[1], AML induces a corresponding algorithmic type system akin to bidirectional typing (Coquand 1996; Dunfield and Krishnaswami 2021) via its the operational semantics, which we will refer to as bidirectional evaluation. Second, AML supports effectful programming via *operations* and *runners* (Ahman and Bauer 2019). Proof assistants rely on effects for a wide range of proof development techniques, such as unification, backtracking, stateful hint databases, et cetera. Runners are akin to exception handlers and offer a general, mathematically sound methodology for working with user-definable effects. We will focus on those aspects of AML that pertain to its purpose as a language for context-free type theories, instead of distracting the reader with the addition of orthogonal, well-understood language features such as algebraic data types.

To construct a judgement, the user writes and evaluates an AML program. The computation of a result then amounts to the checking of a typing derivation. AML can roughly be divided into two parts. One part of the language provides an interface to the rules of type theory. We abstract over the implementation of the data types and rules presented in Chapter 3 and assume it is exposed via an interface that we call the *nucleus*. The nucleus provides the abstract data type of judgements, signatures, boundaries, and other constituent parts of context-free type theories, as well as algorithms for substitution and the effective metatheorems in Section 3.2. We exploit the fact that these constructions on context-free theories all work on *judgements*. Only the proofs of derivability of the results of the constructions proceed by induction on the derivations. Lemma 3.2.13 for instance relates the given type A of a term t in a judgement $\vdash t : A$ to the natural type $\tau(t)$ of t. The construction of the natural type, the conversion-residue,

[1] In the remainder of this chapter, all type theories will be assumed to be standard.

and thus the equality judgement $\vdash \tau(t) \equiv A$ by $\imath(t)$ relating the given and natural type is defined in terms of the judgement $\vdash t : A$. Only the subsequent proof of derivability of this judgement makes recourse to the derivation of $\vdash t : A$. Hence an implementation of a nucleus can provide effective metatheorems without having to store derivations. The bidirectional evaluation of computations allows contextual information to flow from a computation to its sub-computations. Before the definition of AML proper, we give the intuition behind bidirectional evaluation in Section 4.1.1.

The second part of AML supports general purpose programming features such as functions, definitions by pattern matching, and, crucially, operations and runners. These constructs can be used to program various techniques commonly found in proof assistants, such as elaboration of unannotated syntax to fully annotated syntax, simplification routines for expressions, or proof search. We present motivating examples of programs based on runners in Section 4.1.2. Operations interact virtuously with bidirectional evaluation through the Chk-Syn rule as we shall see in Section 4.3.2.

After this preliminary introduction to two central aspects of AML, we define the syntax of AML (§4.2) and present its operational semantics (§4.3). We define derived forms that extend AML with "syntactic sugar" for user convenience (§4.4). We discuss the soundness and completeness of AML with respect to context-free type theories (§4.5), and briefly present the implementation of AML in the Andromeda 2 prover (§4.6).

4.1 AML preliminaries

In this section, we will present two core concepts of AML, bidirectional evaluation and algebraic effect operations. Bidirectional evaluation is, to our knowledge, a new extension of existing ideas in the field, and we will thus spend more time on introducing it. The novelty of runners in AML lies in their use for proof development, which we will showcase before proceeding to the formal definition of AML.

4.1.1 Bidirectional evaluation

Before addressing AML in detail, we will sketch declarative, algorithmic, and bidirectional typing disciplines to prepare the ground for AML's operational semantics. For this purpose we will compare different presentations of a fragment of dependent type theory with dependent products. We deliberately omit rules that would be required to fully specify each system in order to focus our attention on the differences between each discipline.

Presentations of type systems via inference rules such as we saw in Section 2.1.3 and in Section 3.1.2 induce a derivability relation: if the premises are derivable, then so is the conclusion. This style of presentation is *declarative* in nature, i.e. it describes what is derivable, but gives no information about how such a derivation ought to be constructed. We take a declarative presentation of the variable, lambda and application rules from Figure 4.1 as starting point of our comparison. The rules are entirely standard, but let it be noted that Decl-Conv does not constrain the shape

of the term in the conclusion. As a result any of the terms may be derived either via an application of its associated rule or via DECL-CONV.

When implementing a type system, we usually fix an *algorithmic* presentation of these rules. At the very least, when applying a rule, we have to decide on the order in which we verify that the premises are of the correct form. In order to decide if a given term has a given type, the approach of algorithmic type checking is to make the type system syntax-directed, in the sense that by merely looking at the conclusion to be derived, one can decide which rule needs to be applied. If a type theory can be made syntax-directed, the user only has to provide the conclusion rather than writing derivations, and the algorithmic type system can determine the next rule to use. If all premises are recorded in the conclusion, those too can be checked recursively. As a result the user no longer has to write derivations, but only judgements. Once an algorithmic presentation has been chosen, one has to prove that the derivable judgements in both presentations coincide. The algorithmic rules in Figure 4.1 eliminate the non-determinism incurred by conversion by confining the use of type equality to ALGO-APP, but remain otherwise unchanged. This example is inspired by (Pierce 2005, Sec. 2.4) where further details such as the rules for type equality and the proof of equi-derivability of the two presentations can be found. Such a proof would only be a distraction at this point, and we shall proceed to the next system.

$$\textsc{Decl-Var}\ \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Decl-Lambda}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda(x{:}A.b) : \Pi(x{:}A.B)}$$

$$\textsc{Decl-App}\ \frac{\Gamma \vdash s : \Pi(x{:}A.B) \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B[t/x]} \qquad \textsc{Decl-Conv}\ \frac{\Gamma \vdash s : A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash s : B}$$

$$\textsc{Algo-Var}\ \frac{x{:}A \in \Gamma}{\Gamma \vdash x : A} \qquad \textsc{Algo-Lambda}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda(x{:}A.b) : \Pi(x{:}A.B)}$$

$$\textsc{Algo-App}\ \frac{\Gamma \vdash s : \Pi(x{:}A_1.B) \qquad \Gamma \vdash t : A_2 \qquad \Gamma \vdash A_1 \equiv A_2}{\Gamma \vdash s\ t : B[t/x]}$$

Figure 4.1: Declarative and algorithmic rules

In algorithmic type checking, the context, term, and type are given as input, and the rules establish the typing relation. In order to further assist the user, we can try to infer types when possible. Many algorithms for type inference exist, for example based on unification or constraint resolution (Pierce 2002; Pierce 2005), and these techniques have been scaled up to dependent type theory (Harper 1985), but careful analysis is required to show that the typings thus obtained are faithful to the declarative

system. We will consider the *bidirectional typing* approach to the problem, which is close to the algorithmic presentation and thus generalises to context-free type theories.

$$\textsc{Bidi-Var}\ \frac{x{:}A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \textsc{Bidi-Lambda}\ \frac{\Gamma, x{:}A \vdash b \Leftarrow B}{\Gamma \vdash \lambda(x.b) \Leftarrow \Pi(x{:}A.B)}$$

$$\textsc{Bidi-App}\ \frac{\Gamma \vdash s \Rightarrow \Pi(x{:}A.B) \qquad \Gamma \vdash t \Leftarrow A}{\Gamma \vdash s\ t \Rightarrow B[t/x]}$$

$$\textsc{Bidi-Chk-Syn}\ \frac{\Gamma \vdash s \Rightarrow A \qquad \Gamma \vdash A \equiv B}{\Gamma \vdash s \Leftarrow B} \qquad \textsc{Bidi-Ascribe}\ \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash s \Leftarrow A}{\Gamma \vdash s\ \mathsf{as}\ A \Rightarrow A}$$

$$\lambda\textsc{CF-Var}\ \frac{\vdash A\ \mathsf{type}}{\vdash \mathsf{a}^A \Rrightarrow \mathsf{a}^A : A} \qquad \lambda\textsc{CF-Lambda}\ \frac{\mathsf{a}^A\ \mathsf{fresh} \qquad \vdash b[\mathsf{a}^A/x] \Leftarrow B[\mathsf{a}^A/x] \gg b' : B'}{\vdash \lambda(x.b) \Leftarrow \Pi(x{:}A.B) \gg \lambda(x{:}A.b') : \Pi(x{:}A.B'[x/\mathsf{a}^A])}$$

$$\lambda\textsc{CF-App}\ \frac{\vdash s \Rrightarrow s' : \Pi(x{:}A.B) \qquad \vdash t \Leftarrow A \gg t' : A'}{\vdash s\ t \Rrightarrow B[t'/x]}$$

$$\lambda\textsc{CF-Chk-Syn}\ \frac{\vdash s \Rrightarrow s' : A \qquad \vdash A \equiv B}{\vdash s \Leftarrow B \gg s' : B} \qquad \lambda\textsc{CF-Ascribe}\ \frac{\vdash A\ \mathsf{type} \qquad \vdash s \Leftarrow A \gg s' : A'}{\vdash s\ \mathsf{as}\ A \Rrightarrow s' : A'}$$

Figure 4.2: Bidirectional typing and elaboration rules, with and without contexts

Bidirectional typing was first published in (Pierce and Turner 1998) and attributed by Pierce to John Reynolds (Dunfield and Krishnaswami 2021). See (Pfenning 2004; McBride 2018; Dunfield and Krishnaswami 2021) for a more complete introduction to the topic than can be presented here, and e.g. (Lennon-Bertrand 2021) for a recent work on the calculus of inductive constructions. The term "bidirectional" refers to the splitting of the typing relation between terms and their types in two. The first relation is "inference mode", which takes a context and a term as input and synthesises an appropriate type. The second is "checking mode", where a context, term, and type are inputs and the output is simply the success of the typing procedure, i.e. a value of unit type, witnessing that type checking did not fail. We can refine the notion of input as follows. The context and, in checking mode, type inputs are assumed to be well-formed, and each rule has to maintain this invariant. The term whose type is being synthesised or checked is the subject of the rule. The rules named Bidi-* in Figure 4.2 are a bidirectional reformulation of the algorithmic rules of Figure 4.1, where we write $\Gamma \vdash s \Rightarrow A$ for synthesis and $\Gamma \vdash s \Leftarrow A$ for checking. By exploiting contextual

information, the amount of annotations required can be reduced. The user does not need to give the type of terms that can be inferred. For instance the type of a variable need not be provided, as it can be looked up in the typing context Γ. The type of the application in BIDI-APP can be synthesised once the type of s is inferred. Checking propagates available information: The mode of use of t as argument to the dependent function s in BIDI-APP allows it to be checked against the type A. The binder in the conclusion of BIDI-LAMBDA does not need a typing annotation because the type of x can be read off the type $\Pi(x{:}A.B)$ under analysis. The type A is an input to the rule and can thus be assumed to be well-formed[2]. We exploit this fact by dropping the premise $\Gamma \vdash A$ type. The conversion rule takes the form of BIDI-CHK-SYN, and kicks in when an inferring term such as a variable is used in a checking position, say as the argument to a function. Finally, the switch from inference to checking can be stipulated via BIDI-ASCRIBE. It is needed when a checking term like $\lambda(x.x)$ is used in an inferring position. The first premise of BIDI-ASCRIBE requires that A be a well-formed type. In a fixed type theory with universes U_i one would customarily phrase this premise as $\Gamma \vdash A \Leftarrow U_i$. We will see how to check such premises in the absence of universes in PAML-ASCRIBE. Note that despite the presence of BIDI-CHK-SYN which does not impose syntactic restrictions on its subject, the system is again syntax directed, in the sense that each language construct is either synthesising or checking and only one rule is applicable at any point.

The bidirectional presentation is more user friendly than the previous algorithmic system because it allows the omission of typing annotations and the inference of certain terms. While type inference can be achieved via other means such as unification, the bidirectional approach has certain advantages. It is arguably conceptually simpler for the user to predict the behaviour of a bidirectional rule compared to the behaviour of the unification engine, leading to fewer surprises. Nothing precludes the combination of bidirectional typing with other powerful type inference methods based e.g. on unification (Norell 2007; The Coq development team 2021b). In case there is a mismatch between types, it can only occur in the BIDI-CHK-SYN rule, which means that error messages can tell the user where the problem arose. Finally, the simplicity of bidirectional typing allows straightforward proofs of equi-derivability with the declarative presentation.

The λCF rules in Figure 4.2 modify the preceding system in two ways. In the spirit of context-free type theories, contexts are dropped and each variable is instead annotated with its type. Furthermore, we switch from bidirectional typing to *bidirectional elaboration*, which has previously been studied in the context of the calculus of (co-) inductive constructions (Asperti et al. 2012). In the BIDI-* rules, successful synthesis produced a type and checking simply succeeded. In λCF, both synthesis $\vdash t \Rrightarrow t' : A$ and checking $\vdash t \Leftarrow A \gg t' : A'$ take a term (and a type in the case of checking) and produce a fully annotated judgement as output. The bidirectional discipline makes it easy to prove that for the simple fragment of type

[2]We rely here on the fact that well-formedness of $\Pi(x{:}A.B)$ implies well-formedness of A, which holds for standard type theories by inversion (Theorem 3.2.14).

$$\text{pAML-Var}\ \frac{\mathsf{a}^A \text{ fresh}}{\texttt{var}(A\ \mathsf{type}) \rightsquigarrow \mathsf{a}^A : A}$$

$$\text{pAML-Lambda}\ \frac{\begin{array}{rcl} c_A\ @\ \square\ \mathsf{type} & \checkmark & A\ \mathsf{type} \\ c_B\ @\ \{x{:}A\}\ \square\ \mathsf{type} & \checkmark & \{x{:}A\}\ B\ \mathsf{type} \\ c_b\ @\ \{x{:}A\}\ \square : B & \checkmark & \{x{:}A\}\ b : B \end{array}}{\texttt{lambda}(c_A,\ c_B,\ c_b) \rightsquigarrow \lambda(A, \{x\}B, \{x\}b) : \Pi(A, \{x\}B)}$$

$$\text{pAML-App}\ \frac{\begin{array}{rcl} c_A\ @\ \square\ \mathsf{type} & \checkmark & A\ \mathsf{type} \\ c_B\ @\ \{x{:}A\}\ \square\ \mathsf{type} & \checkmark & \{x{:}A\}\ B\ \mathsf{type} \\ c_s\ @\ \square : \Pi(A, \{x\}B) & \checkmark & s : \Pi(A, \{x\}B) \\ c_t\ @\ \square : A & \checkmark & t : A \end{array}}{\mathsf{app}(c_A,\ c_B,\ c_s,\ c_t) \rightsquigarrow \mathsf{app}(A,\ \{x\}B,\ s,\ t) : B[t/x]}$$

$$\text{pAML-Chk-Syn}\ \frac{c \rightsquigarrow \mathcal{J} \qquad r = \begin{cases} \mathcal{J} & \text{if } \mathcal{J} = \mathcal{B}\boxed{e} \\ \texttt{coerce-to}(\mathcal{J}, \mathcal{B}) & \text{otherwise} \end{cases}}{c\ @\ \mathcal{B} \checkmark r} \qquad \text{pAML-Ascribe}\ \frac{c\ @\ \mathcal{B} \checkmark \mathcal{J}}{c\ \mathsf{as}\ \mathcal{B} \rightsquigarrow \mathcal{J}}$$

Figure 4.3: Pseudo-AML rules

theory here presented, if $\vdash t \Rrightarrow t' : A$ then $t = t'$, and if $\vdash t \Leftarrow A \gg t' : A'$ then $t = t'$ and $A = A'$.

There is now a distinction between the input-language for subjects and the judgements produced as output. An expression of the input language still looks type theoretic, and the rules can be seen as a translation from an unannotated language to a fully annotated one. This is the perspective advocated by Pollack (1992), where the translation is used as a vehicle of equi-derivability between the two systems. The elaboration from implicit to explicit syntax does not use bidirectional typing but instead is left unspecified in (Pollack 1992), suggesting it could be instantiated for instance by a constraint solver in the style of (Harper and Pollack 1991). Another advantage of producing judgements besides the connection with translations is that the results we manipulate are always well-formed so long as the rules are sound. This perspective will become more important once we extend the input language to contain non-type-theoretic constructs. When implementing a metalanguage for type theory, the datatype for judgements can be left abstract in the runtime. Only the trusted nucleus which implements the rules needs to manipulate the constituent parts of a judgement. Contrary to the Bidi-* rules, we never need to assemble a judgement $\vdash s : A$ from checking s against A as checking elaboration will always produce a fully formed judgement.

In bidirectional evaluation, sketched in Figure 4.3 for a simplified pseudo-AML

language, the translation from an input language to judgements is taken seriously and interpreted as the evaluation of computations to results. These pAML rules are *not* part of AML but rather illustrate the idea behind AML's operational semantics on some fictitious mini-language restricted to a specific type theory involving lambda and app, whereas the operational semantics of AML will be defined for arbitrary standard type theories.

The moded discipline is retained, but the application of type-theoretic constructors such as λ or app is treated generically as synthesis, because the interplay between a term constructor and its type is specific to each type theory. In synthesis $c \rightsquigarrow \mathcal{J}$, the computation c is input and $\mathcal{J}$ is the result of evaluation. In checking mode $c \mathbin{@} \mathcal{B} \checkmark r$, the computation c and the boundary $\mathcal{B}$ are input, and r is the result of evaluation. Checking mode is generalised in the spirit of finitary type theories to allow the checking of a computation against arbitrary boundaries. This generalisation solves the aforementioned question "How do we check that A is a valid type without referring to universes?", by running the computation c_A in pAML-Lambda in checking mode against the boundary $\square$ type. The fact that `lambda` is a synthesising computation and no longer exploits information available from checking against a Π-type may seem like a shortcoming at first. After all, saving the user from writing typing annotations was one of the benefits of bidirectionalism. The great advantage of working with standard type theories, however, is that *all arguments are checking*, because a standard rule provides sufficient information to construct full boundaries for each subsequent argument from the preceding results. We will see in Section 4.4.3 how to recover the original checking semantics of λCF-Lambda in AML.

In the evaluation of c_B, abstractions make a reappearance, both on the checking boundary and on the resulting judgement. Ascription still switches from inferring to checking evaluation, but now takes a general boundary instead of a type.

The result of a successful computation is a CFTT judgement. Since we focused on bidirectional systems, the compatibility check between potential types for a term, "the type it wants to have" and "the type at which it is used" has been confined to the Chk-Syn rule. In pAML-Chk-Syn, we accordingly have to verify that the computation c synthesised a judgement with the required boundary. Foreshadowing the presence of effect operations and runners in AML, the pAML-Chk-Syn rule also describes what to do if the boundary of $\mathcal{J}$ does not match $\mathcal{B}$. In this case, rather than simply failing, a `coerce-to`$(\mathcal{J}, \mathcal{B})$ operation is triggered, which will give the program under evaluation a chance to rectify the situation and provide a judgement, probably based on $\mathcal{J}$, that does fit $\mathcal{B}$.

Consider for example the situation where $\mathcal{J} = s : A$ and $\mathcal{B} = \square : B$. We can recover the behaviour of previous Chk-Syn rules by providing a runner that computes a judgement $A \equiv B$ by α, performs the conversion of s to B, and yields the resulting judgement.

How will the method of bidirectional evaluation fare for context-free type theories? The rules for object judgements in standard context-free type theories are such that for any symbol there is exactly one rule by Definition 3.1.14 and each such rule further records all of the object premises as arguments to the symbol in question. We thus

stand a good chance to make the system syntax-directed, and we have seen how to generalise checking mode to boundaries. A possible complication arises from the fact that symbol rules are allowed to have equational premises. While context-free type theories are careful to record the assumptions used in such premises in the conclusion, they will not record any evidence that would allow us to reconstruct a derivation for the purposes of type-checking. As context-free type theories allow the formulation of theories with undecidable judgemental equality, we cannot provide a once-and-forall algorithm to check if a judgement $\mathcal{J}$ can be transformed into one with boundary $\mathcal{B}$. In our quest to offer a convenient method for working with context-free type theories we therefore have to consider the possibility that a conversion is required at any moment, and that equational premises need to be reconstructed. We shall see that operations and runners provide a natural solution to this problem, supporting the user in the implementation of proof assistant techniques.

4.1.2 Operations and runners

In this section, we describe the basic intuition behind operations and runners. We refer to (Pretnar 2015) and (Ahman and Bauer 2019) for background on algebraic effects and handlers and to runners respectively, and focus on their use for proof development here instead. For the sake of our purposes, a runner is best thought of as an effect handler that resumes its continuation exactly once, and in tail position. They further differ from the runners of Ahman and Bauer in that they do not carry state. We will illustrate this slightly cryptic sentence with two examples, one based on printing and one based on the `coerce-to` operation that we encountered in Figure 4.3 in the preceding section.

For the `debug-print` example we will refer to the beta-rule, presented here declaratively in context-free style. Context-free type theories can represent certain theories such as extensional type theory (Martin-Löf 1982) in which untyped beta-reduction is unsound, and as a consequence the following version of the beta-rule comes with full typing annotations.

$$\textsc{beta}\ \frac{A\ \mathsf{type} \qquad \{x : A\}\ B\ \mathsf{type} \qquad \{x : A\}\ b : B \qquad a : A}{\mathtt{app}(A, \{x : A\}\ B, \mathtt{lambda}(A, \{x : A\}\ B, \{x : A\}\ b), a) \equiv b[a/x] : B[a/x]}$$

The declarations

```
operation coerce-to :  judgement * boundary → judgement
operation debug-print :  string * judgement → unit
```

indicate that and `coerce-to` takes a pair of a judgement and a boundary as argument. A runner handling `coerce-to` is expected to produce a judgement to be returned to the point where the operation was invoked. Likewise, `debug-print` takes a message and a judgement, and a matching runner should produce a unit value. At first, an operation may seem a bit like a function call, but the resolution of an operation to a handling runner happens dynamically at runtime. This allows us to locally modify the behaviour

of a program that performs operations by wrapping it with a particular runner. The function `beta-reduce` that takes a term, matches it, and, if the head constructor is a beta redex, applies the beta rule to produce a judgemental equality between `t` and the reduced `b[a]`, and calls itself recursively.

```
let rec beta-reduce t =
  match t with
  | app(?A, ?B, lambda(_, _, ?b), ?a) →
    debug-print ("redex found:", t);
    let eq = beta A B b a in debug-print ("equation:", eq);
    (match eq with _ ≡ ?t' →
       let eq' = beta-reduce t' in
       eq-tm-transitivity eq eq' )
  | _ → debug-print ("not a redex:", t); eqtm-refl t
```

At several points, the `debug-print` operation is called. Suppose that the beta-reduce is used in the context of some larger program. Suppose further that for most of the program, `beta-reduce` works as intended, but at some point it fails because with an error about an incorrect use of the beta rule. It may be desirable to *locally* change the behaviour of `debug-print`, silencing all calls except for those in vicinity of the offending redex. This can be achieved by wrapping earlier calls in the `silence` runner, and using the `std-err` runner.

```
let silence = runner debug-print (msg, j) → ()
let std-err = runner debug-print (msg, j) →
    fprintf stderr "[Debug] %s: %j\n" msg j
let e₂ = let e₁ = with silence run
             let y = c₁ in
             beta-reduce y in
           with std-err run
             let z = c₂ in
             beta-reduce z
```

During the computation of `x`, calls to `beta-reduce` that may occur in c_1 and the call `beta-reduce y` will trigger the `debug-print` operation. As the computation is wrapped in a `with silence run` ... clause, the calls will jump to the `silence` runner, which does nothing and returns () to the call site in `beta-reduce`. The evaluation of c_2 and the subsequent call to `beta-reduce z` instead are handled by the `std-err` handler, which prints a debugging message to the `stderr` channel. This completes the first example. We have seen how a runner can be used *locally* to *dynamically* change the behaviour of a program.

The second example showcases how this mechanism can be exploited to allow a program to take advantage of local information.

```
let r = runner
  | coerce-to (J, bdry) →
    match (J, bdry) with
    | ((_ : ?A), (□ : ?B)) → let e = eqchk A B in convert J e
    | _ → coerce-to (J, bdry)
```

The runner `r` handles a `coerce-to` operation by matching its arguments to see if `J` and `bdry` are a term judgement and a term boundary respectively, and calling a function called `eqchk` that produces an equality judgement between `A` and `B`, which is then used to convert `J` to the desired type. In case the runner is called with a judgement or boundary of different shape, the operation is instead re-raised. The `eqchk` function here could be some specialised equality checking function for the type theory at hand, applying congruence rules, reducing redices, et cetera.

For the sake of the next example, we will assume that we work in a type theory with equality reflection. The reflection rule takes a type, two terms, and a proof of equality between them and returns the corresponding judgemental equality. The implementer of `eqchk` may have included an escape hatch. Say that `eqchk` raises an `equate` operation if it fails to prove a certain equality. If we are in a context where a certain equation is known to hold, we can then provide a local runner that exploits this assumption by handling `equate`.

```
operation equate : judgement * judgement → judgement
let X = {m : Nat} {n : nat} {v : Vec m} {e : Eq Nat m n}
          let by-e = runner equate (J₁, J₂) →
            match (J₁, J₂) with
            | ((?x : Nat), (?y : Nat)) →
              if x = n && y = m then
                reflect Nat m n e
              else
                equate (J₁, J₂)
            | (_, _) → equate (J₁, J₂)
          in with by-e run
               with r run
                 coerce-to(v, (□ : Vec n))
```

In the above example, we provide a runner using the local information provided in by the abstraction, in particular the variable `e`. The `by-e` runner handles `equate` by matching the arguments to ensure that the equation `m ≡ n : Nat` is requested. If this is the case, the runner uses the `reflect` rule that turns the term `e : Eq Nat m n` into the required equation. In all other cases, the runner re-raises the operation. We wrap the `coerce-to` call with the runner `r`, which will try to equate the type of `v`, `Vec m`, and the type on the boundary, `Vec n`. It can apply congruence, but has no way of proving the general equation `m ≡ n : Nat`. When `equate` is performed, `by-e` will return the required equation, and `r` can finish its work by appealing to conversion along `Vec m ≡ Vec n`.

These examples show an important application of operations and runners in AML, namely as a mechanism to inhabit equations without forcing the user to write down their derivations. Historically, this was the original purpose of operations in Andromeda, but the mechanism turns out to be very versatile. We can, for instance, use a runner for `coerce-to` to install "implicit coercions", which are frequently encountered in proof assistants.

```
let bool-of-nat = runner coerce-to (J, bdry) →
```

```
    match (J, bdry) with
    | ((?n : Nat), (□ : Bool)) → match n with 0 → false | S _ → true
    | (_, _) → coerce-to (J, bdry)

let monoid-of-group = runner coerce-to (J, bdry) →
    match (J, bdry) with
    | ((?g : Group), (□ : Monoid)) → g.monoid
    | (_, _) → coerce-to (J, bdry)
```

It takes little imagination to see how to use `coerce-to` to implement, for instance, typeclasses (Sozeau and Oury 2008) or canonical structures (Mahboubi and Tassi 2013).

This concludes our tour of operations and runners by example. The next section describes the operational semantics of AML, and in particular how AML combines operations and runners with bidirectional evaluation.

4.2 AML syntax

The expressions of the Andromeda metalanguage are classified as inert *values* and (potentially) effectful *computations*, which evaluate to *results*. This division is customary for effectful languages (Bauer and Pretnar 2015) because the isolation of effects simplifies the presentation of the operational semantics: with the notable exception of `let`, all language constructs only contain one computation and thus only one possible source of effects, and we can thus consider effects in the sequence dictated by successive let-bindings.

Concrete syntax	Abstract syntax	Meaning	Mode
Computation $C \ni c$ ::=			
`return` v	`return`(v)	value	$\rightsquigarrow$
`let` x = c_1 `in` c_2	`let`$(c_1,\ x.\, c_2)$	local definition	$\rightsquigarrow$, ✓
$v_1\ \ v_2$	`fun-apply`$(v_1,\ v_2)$	function application	$\rightsquigarrow$, ✓
`match` v `with` $(p \Rightarrow c)^*$	`match`$(v,\ (p.c)^*)$	case match	$\rightsquigarrow$, ✓
$op\ \ v$	`perform`$(op,\ v)$	perform operation	$\rightsquigarrow$, ✓
`with` v `run` c	`run`$(v,\ c)$	run with runner	$\rightsquigarrow$, ✓

Figure 4.4: Syntax of general AML computations

The *computations* in Figure 4.4 are familiar general-purpose programming constructs. We can embed values into computation via `return`, `let`-bind the result computations of c_1 in c_2, apply functions, and perform case analysis by pattern `match`ing. An operation *op* can be performed by applying it to an argument value. For simplicity's sake operations are presented in their generic form (Plotkin and Power 2003), where the user supplies no explicit continuation. A computation c can be `run` wrapped `with` a runner v, allowing v to handle operations raised by c.

For each construct we give a notation that we may use in examples, the corresponding abstract syntax with binding information, and the evaluation mode in the sense of bidirectional evaluation. Most of the general computations can be run in either synthesis $\rightsquigarrow$ or checking ✓ mode. As their nature is not type theoretic, they have no use for a checking boundary, and can thus be run in synthesis mode, but if available, they can propagate the checking information to sub-computations. Computations that can be evaluated in both modes are called neutral. The exception here is `return` v, as v is already fully evaluated and has nowhere to pass a checking boundary on to.

Concrete syntax	Abstract syntax	Meaning	Mode
Computation $C \ni c$ ::=			
c `as` v	`ascribe`$(c,\ v)$	boundary ascription	$\rightsquigarrow$
`tt-var` v	`tt-var`(v)	fresh TT variable	$\rightsquigarrow$
`tt-abstr` $v_1\ v_2$	`tt-abstr`$(v_1,\ v_2)$	abstraction	$\rightsquigarrow$
`tt-subst` $v_1\ v_2$	`tt-subst`$(v_1,\ v_2)$	substitution	$\rightsquigarrow$
`tt-mvar` v	`tt-mvar`(v)	fresh TT metavariable	$\rightsquigarrow$
`tt-derived` $v_1\ v_2$	`tt-derived`$(v_1,\ v_2)$	derivable rule	$\rightsquigarrow$
`tt-inst` $v_1\ v_2$	`tt-inst`$(v_1,\ v_2)$	rule application	$\rightsquigarrow$
`tt-congr`$(v_1,\ v_2,\ v^*)$	`tt-congr`$(v_1,\ v_2,\ v^*)$	congr. rule instance	$\rightsquigarrow$
$v_1 \overset{\alpha}{==} v_2$	`tt-alpha-equal`(v_1, v_2)	alpha equality	$\rightsquigarrow$
$v_1 \overset{\varepsilon}{==} v_2$	`tt-erasure-equal`(v_1, v_2)	erased syntactic eq.	$\rightsquigarrow$
`tt-refl` $v_1\ v_2$	`tt-refl`$(v_1,\ v_2)$	reflexivity of equality	$\rightsquigarrow$
`tt-convert` $v_1\ v_2$	`tt-convert`$(v_1,\ v_2)$	conversion	$\rightsquigarrow$
□ `type`	`tt-bdry-ty`	type boundary	$\rightsquigarrow$
`tt-bdry-tm` v	`tt-bdry-tm`(v)	term boundary	$\rightsquigarrow$
`tt-bdry-eqty` $v_1\ v_2$	`tt-bdry-eqty`$(v_1,\ v_2)$	type eq. boundary	$\rightsquigarrow$
`tt-bdry-eqtm` $v_1\ v_2\ v_3$	`tt-bdry-eqtm`(v_1, v_2, v_3)	term eq. boundary	$\rightsquigarrow$
∂v	`tt-bdry-of`(v)	bdry. of a judgement	$\rightsquigarrow$

Figure 4.5: Syntax of type theoretic AML computations

The computations in Figure 4.5 implement the rules of context-free type theories. In order to guarantee that only derivable judgements can be constructed, we never manipulate pieces of syntax such as raw terms directly, but always as part of a judgement. Boundary ascription changes the evaluation mode from synthesising to checking. The `tt-*` computations are smart constructors that directly call the nucleus. All of the nucleus computations are synthesising, but we shall see in Section 4.4 how derived constructs can be implemented on top of core AML to exploit checking information. The selection of `tt-`computations is deliberately minimal. Following the definition of context-free type theories (Definition 3.1.11), variables and metavariables can be created via `tt-var` and `tt-mvar`. In order to ensure the well-typedness of annotations (Definition 3.1.12), a type judgement (resp. boundary) has to be provided. An abstraction is formed via `tt-abstr`, and `tt-subst` implements the admissibility of substitution. The `tt-derived` and `tt-inst` computations allow the construction of derivable rules (Definition 2.1.17) and rule application via instantiation. Congruence

rules for symbols and metavariables are available through `tt-congr`. A boolean test for alpha equality of boundaries and judgements is provided via `tt-alpha-equal`. A boolean test for FTT alpha equality of the erasure of boundaries and judgements is provided via `tt-erasure-equal`. Judgemental equalities can be formed via `tt-refl` and eliminated via `tt-convert`, which implements conversion of both term and term-equality judgements. Symmetry and transitivity of judgemental equality are not included as primitives as they can be postulated as standard context-free equality rules via the `rule` toplevel command (Fig. 4.7). We do tacitly assume both rules to be part of any type theory in AML, as the metatheorems we implement rely on them, but do provide syntactic forms for them in order to simplify the presentation of syntax and operational semantics of AML. Finally there are four smart constructors for the creation of boundaries, and one that projects a judgement to its boundary.

Concrete syntax	Abstract syntax	Meaning
Value $V \ni v$ ::=		
x	`var`(x)	variable
`fun` $x \rightarrow c$	`fun`$(x.\ c)$	function
`runner` $(\mid op\text{-}case)^*$	`runner`$(op\text{-}case^*)$	runner
`Unbounded`	`Unbounded`	bound constructor
`Boundary` v	`Boundary`(v)	bound constructor
`false`	`false`	bool constructor
`true`	`true`	bool constructor
$\varnothing$	$\mathcal{J}$	judgement, see Fig. 3.1
$\varnothing$	$\mathcal{B}$	boundary, see Fig. 3.1
$\varnothing$	`rule` $M : \mathcal{B} \Longrightarrow v$	rule
where $op\text{-}case$ ::=		
$op\, x\, y \rightarrow c$	`case-op`$(op,\ x.y.c)$	operation case
Result $R \ni r$ ::=		
$\varnothing$	`val`(v)	value
$\varnothing$	`op`$(op,\ v_1,\ v_2,\ x.c_\kappa)$	operation
Identifier x, y, op, M, R, S ::=		
`x`, `y`, `op`, `M`, `R`, `S`, ...	x	identifiers

Figure 4.6: Syntax of AML values and results

The *values* of AML (Fig. 4.6) can again be classified into general programming and type theoretic. There are variables, functions, runners, and AML datatype constructors. A runner v can have a number of different operation cases, each of which binds two variables, say x and y, in the computation c which gets executed if the operation op is encountered. Upon evaluation of c, the operation argument gets substituted for x, and y gets replaced either by `Unbounded` if op is performed in synthesis mode, or by `Boundary` $\mathcal{B}$ if op is performed while checking against $\mathcal{B}$.

The type theoretic values are judgements, boundaries, and rules. While general purpose values can be provided directly by the user, type theoretic values can only be obtained as the result of evaluation of a type theoretic computation, and therefore no

concrete syntax is available for this class. The full grammar of the constituent parts of judgements and boundaries is given in Figure 3.1. Rules are presented in curried form, and can be applied via `tt-inst` to one premise at a time, resulting either in a judgement if no further metavariable abstraction is encountered or in a further rule where M has been instantiated with the given premise.

The *results* that evaluation produces are either values or operations. An operations is tagged with its name *op* and carries an argument value v_1, a checking bound v_2, and a continuation $x.c_\kappa$, which gets built up as the operation bubbles up from its origin to a runner with a clause for *op*, and gets invoked after the runner finishes its work.

Concrete syntax	Abstract syntax	Meaning
Top-runner stack $\mathcal{R}$ ::=		
$\mathcal{R}[c] =$ `with` v_1 `run` ... `with` v_n `run` c		
Theory $\mathbb{T}$ ::=		
$\{\ldots,\ R \mapsto (\texttt{rule}\ M_1 : \mathcal{B}_1 \Longrightarrow \ldots \Longrightarrow \texttt{rule}\ M_n : \mathcal{B}_n \Longrightarrow j)\ ,\ \ldots\}$		
Program *cmd** ::=		
$cmd_0 ; \ldots ; cmd_n$	$[\,cmd_0 ; \ldots ; cmd_n]$	list of commands
Toplevel command *cmd* ::=		
`let` x = c	`let`$(x,\ c)$	definition
`rule` $R\ v$	`rule`(R, v)	rule definition
`operation` *op* : *op-mlsig*	`operation`(*op*, *op-mlsig*)	operation declaration
`with` v `end`	`with`(v)	install top runner
op-mlsig ::=		
$mltype_1 \longrightarrow mltype_2$	$(mltype_1, mltype_2)$	operation ML signature
Pattern p ::=		
$?x$	`p-var`(x)	variable
_	`p-ignore`	ignore
$S(p^*)$	`p-sym`(S, p^*)	symbol application
`mvar` $p(p^*)$	`p-mvar`$(p,\ p^*)$	metavar. application
$\kappa(p_1, p_2)$	`p-convert`$(p_1,\ p_2)$	conversion term
□ `type`	`p-bdry-ty`	type boundry
$\square : p$	`p-bdry-tm`(p)	term boundry
$p_1 \equiv p_2$ `by` □	`p-bdry-eqty`$(p_1,\ p_2)$	type equality boundry
$p_1 \equiv p_2 : p_3$ `by` □	`p-bdry-eqtm`$(p_1,\ p_2,\ p_3)$	term equality boundry
{_ : p} _	`p-abstr`(p)	abstraction
`rule` (_ : p) _	`p-rule`(p)	rule

Figure 4.7: Syntax of AML toplevel commands and patterns

An AML program is composed of a sequence of *toplevel commands* (Fig. 4.7). A toplevel `let`-binding allows us to evaluate a computation and bind the resulting value in the rest of the program. A new type theoretic rule can be defined via `rule` $R\ v$, where v should be a rule boundary (Definition 3.1.4), i.e. a boundary $\mathscr{b}$, possibly with several metavariables $M_i : \mathcal{B}_i$ abstracted. A rule definition extends the

theory $\mathbb{T}$ with the symbol rule (Definition 3.1.5) or equality rule (Definition 3.1.6) associated to the rule boundary v. The declaration of an operation has no operational bearing, but is included here for the sake of readability of example code, and because it is required for the static AML typechecking that the Andromeda 2 implementation performs. Besides user declared operations, AML has one built-in operation `coerce` : `judgement` $\to$ `judgement`. Similar to the `coerce-to` operation example we encountered in Section 4.1.2, `coerce` is used to give the user the occasion to rectify the boundary of a judgement $\mathcal{J}$ with a runner that handles `coerce` $\mathcal{J}$. Installing a top runner wraps the rest of the program in the runner, by extending the current runner stack. The runner stack $\mathcal{R}$ is a simple evaluation environment consisting of nested runners that wrap around a hole, into which we can plug a computation.

The patterns in Figure 4.7, to be used in conjunction with a `match` statement, implement further type theoretic metatheorems. To distinguish pattern variables from constructors we require that they be prefixed with ?, i.e. `?x` binds the variable `x`, while `true` matches the constructor `true`. A symbol application pattern makes it possible to recover the premises used to derive a symbol rule by inversion, and metavariable patterns works analogously. Lemma 3.2.13, which relates the type of a term s to the natural type of s, is implemented as the `p-convert`(p_1, p_2) pattern. The `p-ty`, `p-tm`, `p-eqty`, and `p-eqtm` patterns expose the presuppositivity theorem, allowing to extract the presuppositions from the boundary of a judgement, and likewise for boundaries. The type of an abstraction can be recovered via `p-abstr`, and similarly for `rule`. To avoid redundancy in AML, we do not allow a pattern for the body of the abstraction, as the body can be recovered by substituting a fresh variable of the appropriate type.

4.3 AML operational semantics

The essence of AML's operational semantics is obtained by combining bidirectional evaluation with operations and runners. We adopt a big step, "fine-grained call-by-value" (Levy et al. 2003) style of presentation. We define two evaluation functions, synth-comp : $Th \times C \rightharpoonup R$, and check-comp : $Th \times C \times B \rightharpoonup R$. Here Th, C, B, and R stand for CFTT theories, computations, CFTT boundaries, and results respectively. We will use the notation $\mathbb{T} \mid c \rightsquigarrow r$ for "c synthesises r" and $\mathbb{T} \mid c \mathbin{@} \mathcal{B} \checkmark r$ for "c checked against $\mathcal{B}$ evaluates to r". The evaluation functions are recursively defined by inference rules, where premises are either further evaluations or side conditions, and premises are evaluated left to right, top to bottom. We will usually omit the theory argument $\mathbb{T}$ in the notation for the evaluation functions. It is understood that $\mathbb{T}$ is passed on unmodified during each recursive call to synth-comp or to check-comp in a premise; the only place where $\mathbb{T}$ is accessed is during pattern matching, the only place where the theory can be extended is at the toplevel.

Interlude: On AML typing The operational semantics presented in this section is partial and evaluation can fail at any moment when an ill-formed use of a type theoretic rule is encountered. It is desirable to rule out such failure at runtime, and the

standard technique to do so is to equip the language with a type system that ensures that *well-typed programs cannot go wrong* (Milner 1978). The reason why we do not present a type system for AML is that such a type system would have to be able to classify judgements of context-free type theories in order to ensure that all rule applications occurring in a program are well-formed. In other words, the system would have to be at least as expressive as the system AML is supposed to capture. While it is certainly reasonable to expect that CFTT can be expressed within a sufficiently strong dependent type theory, the effectful nature of AML complicates the situation. Effects in type theory are the topic of ongoing research (Swamy et al. 2016; Ahman, Ghani et al. 2016; Pédrot and Tabareau 2019) and designing a safe type system for AML would be a research problem in its own right, and is thus beyond the scope of the present thesis. In an effort to provide the user at least with safety for the non-type-theoretic part of the language, the implementation of AML in Andromeda 2 provide an off-the-shelf Hindley-Milner style simple type system with prenex polymorphism and abstract types for judgements, boundaries, and rules. It bears repeating that the static typing of the AML metalanguage is independent of the typing of the CFTT object language which happens via evaluation.

4.3.1 General programming

$$\textsc{Syn-return}\ \frac{}{\texttt{return}\,(v) \;\rightsquigarrow\; \texttt{val}\,(v)}$$

$$\textsc{Syn-op}\ \frac{}{\texttt{perform}(op, v) \;\rightsquigarrow\; \texttt{op}(op, v, \mathsf{Unbounded}, x.\texttt{return}\,(x))}$$

$$\textsc{Chk-op}\ \frac{}{\texttt{perform}(op, v)\ @\ \mathscr{B} \;\checkmark\; \texttt{op}(op, v, \mathsf{Boundary}(\mathscr{B}), x.\texttt{return}\,(x))}$$

Figure 4.8: Operational semantics of return and operations

By Syn-return in Figure 4.8, a value embedded via `return` (v) synthesises the result `val` (v). We use the explicit `val` tag to distinguish computed values from operations. Operations are neutral, and we thus give both a synthesis and a checking rule. The result of performing an operation is, of course, an operation, which we tag with `op` to distinguish them from values. The name of the operation and the argument v are included in the result. Furthermore, Chk-op stores the boundary that is currently being checked against as `Boundary`$(\mathscr{B})$ for use in a runner handling *op*, while Syn-op stores `Unbounded`. Operations propagate outwards by accumulating the continuation, i.e. the rest of the computation that needs to be performed after the operation gets handled. The base case, when an operation is performed, is to simply return whatever result the handling runner computes, i.e. the continuation $(x.\texttt{return}\,(x))$.

Two computations c_1, c_2 can be sequenced via a let-binding (Fig. 4.9). If c_1 produces a value v, the evaluation of `let` $c_1 = x$ `in` c_2 continues with $c_2[v/x]$, propagating the checking boundary if available (Chk-let-val). The case where c_1

$$\textsc{Syn-let-val}\ \frac{c_1 \rightsquigarrow \texttt{val}\,(v) \qquad c_2[v/x] \rightsquigarrow r}{\texttt{let}(c_1,\, x.c_2) \rightsquigarrow r}$$

$$\textsc{Syn-let-op}\ \frac{c_1 \rightsquigarrow \texttt{op}(op,\, v_1,\, v_2,\, y.c_\kappa)}{\texttt{let}(c_1,\, x.c_2) \rightsquigarrow \texttt{op}(op,\, v_1,\, v_2,\, y.\texttt{let}(c_\kappa,\, x.c_2))}$$

$$\textsc{Chk-let-val}\ \frac{c_1 \rightsquigarrow \texttt{val}\,(v) \qquad c_2[v/x]\ @\ \mathcal{B} \ \checkmark\ r}{\texttt{let}(c_1,\, x.c_2)\ @\ \mathcal{B}\ \checkmark\ r}$$

$$\textsc{Chk-let-op}\ \frac{c_1 \rightsquigarrow \texttt{op}(op,\, v_1,\, v_2,\, y.c_\kappa)}{\texttt{let}(c_1,\, x.c_2)\ @\ \mathcal{B}\ \checkmark\ \texttt{op}(op,\, v_1,\, v_2,\, y.\texttt{let}(c_\kappa,\, x.\texttt{ascribe}(c_2,\, \mathcal{B})))}$$

Figure 4.9: Operational semantics of let binding

synthesises an operation is more interesting. The continuation $y.c_\kappa$ represents the rest of c_1 that is awaiting the result of handling *op*. In Syn-let-op, the continuation gets extended to bind the result of c_κ to x and subsequently continue with the evaluation with c_2 once the operation gets handled by a runner. In Chk-let-op, we furthermore ascribe the current checking boundary to c_2, which will ensure that it will in turn be run in checking mode against $\mathcal{B}$.

Evaluating a computation c wrapped in a runner (Fig. 4.10) returns values transparently (Syn-run-val,Chk-run-val), and checking propagates $\mathcal{B}$ to c. If the runner v does not have a clause for the particular operation raised, it is propagated outward, but the continuation is re-wrapped with the v (Syn-run-op-propagate). In the case of Chk-run-op-propagate, the continuation is furthermore placed under boundary ascription. When c evaluates to an operation *op* and the handler v has a clause $x.y.c_{op}$ associated to *op*, the bubbling-up of *op* is stopped. The continuation c_κ expects the result of the runner to be bound to z. We hence evaluate a let statement which binds to z the value produced by the runner clause c_{op}. Recall from Chk-op that an operation which is performed in checking mode saves its checking boundary as `Boundary B_op`, whereas Syn-op will produce `Unbounded` for v_2. The runner clause c_{op} is evaluated in synthesis mode if v_2 = `Unbounded`. If v_2 = `Boundary B_op`, the operation was performed while checking against `B_op`, and accordingly c_{op} is wrapped with the boundary ascription `as B_op` in this case. The value produced by the runner case is bound to z.

In analogy with the terminology employed for handlers, our runners can be said to be *deep* and *forwarding*. Our runners are deep in the sense that the continuation c_κ is wrapped again in the runner v to handle further operations, and finally c_κ is resumed. Our runners are forwarding: If no clause matching *op* is found, the operation keeps propagating outward, which can be modelled by assuming each runner has an implicit default clause that re-raises an operation its arguments unchanged.

The fact that a runner invokes its continuation exactly once means that one of our motivating examples of effects, backtracking, is not expressible as a runner. On the other hand, this very restriction allows us to install handlers at toplevel. A more

$$\textsc{Syn-run-val}\ \frac{c \rightsquigarrow \mathtt{val}\,(v)}{\mathtt{with}\ v\ \mathtt{run}\ c \rightsquigarrow \mathtt{val}\,(v)} \qquad \textsc{Chk-run-val}\ \frac{c\ @\ \mathcal{B} \checkmark \mathtt{val}\,(v)}{\mathtt{with}\ v\ \mathtt{run}\ c\ @\ \mathcal{B} \checkmark \mathtt{val}\,(v)}$$

$$\textsc{Syn-run-op-propagate}\ \frac{c \rightsquigarrow \mathtt{op}(op, v_1, v_2, z.c_\kappa) \qquad \forall\, \mathtt{case\text{-}op}(op', x.y.c_{op'}) \in v,\ op' \neq op}{\mathtt{with}\ v\ \mathtt{run}\ c \rightsquigarrow \mathtt{op}(op, v_1, v_2, z.\mathtt{with}\ v\ \mathtt{run}\ c_\kappa)}$$

$$\textsc{Chk-run-op-propagate}\ \frac{c\ @\ \mathcal{B} \checkmark \mathtt{op}(op, v_1, v_2, z.c_\kappa) \qquad \forall\, \mathtt{case\text{-}op}(op', x.y.c_{op'}) \in v,\ op' \neq op}{\mathtt{with}\ v\ \mathtt{run}\ c\ @\ \mathcal{B} \checkmark \mathtt{op}(op, v_1, v_2, z.\mathtt{with}\ v\ \mathtt{run}\ \mathtt{ascribe}(c_\kappa, \mathcal{B}))}$$

$$\textsc{Syn-run-op-handle}\ \frac{\begin{array}{c} c \rightsquigarrow \mathtt{op}(op, v_1, v_2, z.c_\kappa) \\ v = \mathtt{runner}(\ldots, \mathtt{case\text{-}op}(op,\ x.y.c_{op}),\ \ldots) \\ \left(\begin{array}{l} \mathtt{let}\ z = \mathtt{match}\ v_2\ \mathtt{with} \\ \quad \mid\ \mathtt{Unbounded} \to c_{op}[v_1/x, v_2/y] \\ \quad \mid\ \mathtt{Boundary\ ?B_op} \to c_{op}[v_1/x, v_2/y]\ \mathtt{as}\ \mathtt{B_op} \\ \mathtt{in}\ \mathtt{with}\ v\ \mathtt{run}\ c_\kappa \end{array}\right) \rightsquigarrow r \end{array}}{\mathtt{with}\ v\ \mathtt{run}\ c \rightsquigarrow r}$$

$$\textsc{Chk-run-op-handle}\ \frac{\begin{array}{c} c\ @\ \mathcal{B} \checkmark \mathtt{op}(op, v_1, v_2, z.c_\kappa) \\ v = \mathtt{runner}(\ldots, \mathtt{case\text{-}op}(op,\ x.y.c_{op}),\ \ldots) \\ \left(\begin{array}{l} \mathtt{let}\ z = \mathtt{match}\ v_2\ \mathtt{with} \\ \quad \mid\ \mathtt{Unbounded} \to c_{op}[v_1/x, v_2/y] \\ \quad \mid\ \mathtt{Boundary\ ?B_op} \to c_{op}[v_1/x, v_2/y]\ \mathtt{as}\ \mathtt{B_op} \\ \mathtt{in}\ \mathtt{with}\ v\ \mathtt{run}\ c_\kappa \end{array}\right) @\ \mathcal{B} \checkmark r \end{array}}{\mathtt{with}\ v\ \mathtt{run}\ c\ @\ \mathcal{B} \checkmark r}$$

Figure 4.10: Operational semantics of runners

general mechanism such as full-blown effect handlers with first-class continuations would have to explain the meaning of backtracking at toplevel.

The operational semantics of runners is reminiscent of that of exception handlers or algebraic effect handlers. They behave virtually in the same way if the handled computation evaluates to a value by returning the result. An operation behaves differently from an exception in that it accumulates its continuation rather than discarding it. The reader familiar with the presentation of runners in (Ahman and Bauer 2019) may have noticed that in contrast to (Ahman and Bauer 2019), that the runners here presented are not stateful. The addition of state would needlessly clutter the presentation of the operational semantics and distract from the salient language features of AML. The addition of state to AML is standard, and ML-style references are in fact available in the Andromeda 2 implementation of AML.

The reader familiar with general handlers for algebraic effects may wonder *Why not general handlers?* There are several reason to prefer runners in our setup. General

handlers get access to the continuation of an operation as a first class value, and can invoke their continuation any number of times. In our setting, the resumption of the continuation is the last thing a runner does, and we can propagate checking information through a (`with v run c`) form. A handler may change the return type of an operation by wrapping the call to the continuation in a further computation, and no sensible way of propagating checking information is possible. As runners invoke their continuation exactly once, they can be implemented efficiently, without copying the call stack (Bruggeman et al. 1996; K. Sivaramakrishnan et al. 2021). While there is promising research on efficient implementations of general handlers, non-trivial optimisations are required to achieve comparable results (Karachalias et al. 2021; Schuster et al. 2020). Finally, the motivating example for runners in the context of AML is the `coerce` operation, and we found that most handlers for `coerce` that we implemented in Andromeda 1 (Bauer, Gilbert et al. 2018) were in fact expressible as runners.

$$\textsc{Syn-match}\quad \frac{v \sim p_i \rhd \sigma \qquad c_i[\sigma] \rightsquigarrow r}{\mathtt{match}(v, [p_1.c_1, \ldots, p_n.c_n]) \rightsquigarrow r} \qquad \textsc{Syn-fun-apply}\quad \frac{c[v/x] \rightsquigarrow r}{\mathtt{fun\text{-}apply}(\mathtt{fun}(x.c), v) \rightsquigarrow r}$$

$$\textsc{Chk-match}\quad \frac{v \sim p_i \rhd \sigma \qquad c_i[\sigma] \mathbin{@} \mathcal{B} \checkmark r}{\mathtt{match}(v, [p_1.c_1, \ldots, p_n.c_n]) \mathbin{@} \mathcal{B} \checkmark r} \qquad \textsc{Chk-fun-apply}\quad \frac{c[v/x] \mathbin{@} \mathcal{B} \checkmark r}{\mathtt{fun\text{-}apply}(\mathtt{fun}(x.c), v) \mathbin{@} \mathcal{B} \checkmark r}$$

Figure 4.11: Operational semantics of case matching and function application

The evaluation of pattern matching (Fig. 4.11) proceeds by comparing the value v with patterns p_1 through p_n until one of the patterns p_i succeeds in producing a substitution σ. The rules for computing substitutions will be presented in Figure 4.17. The substitution is applied to the associated computation c_i, and the checking boundary is propagated if available. Function application behaves similarly, propagating information checking if available.

4.3.2 Type theory

The evaluation of type theoretic forms of AML closely follows the rules of context-free type theories. The two rules in Figure 4.12 are particular to bidirectional evaluation.

$$\textsc{Syn-Ascr}\quad \frac{c \mathbin{@} \mathcal{B} \checkmark \mathcal{J}}{\mathtt{ascribe}(c, \mathcal{B}) \rightsquigarrow \mathcal{J}} \qquad \textsc{Chk-Syn}\quad \frac{\begin{array}{l}\mathtt{let}\ \mathtt{x} = c\ \mathtt{in} \\ \mathtt{let}\ \mathtt{B} = \partial\mathtt{x}\ \mathtt{in} \\ \mathtt{let}\ \mathtt{b} = \mathtt{B} \overset{\alpha}{==} \mathcal{B}\ \mathtt{in} \\ \mathtt{match}\ \mathtt{b}\ \mathtt{with} \\ \mid\ \mathtt{true} \to \mathtt{return}\ \mathtt{x} \\ \mid\ \mathtt{false} \to (\mathtt{coerce}\ \mathtt{x})\ \mathtt{as}\ \mathcal{B}\end{array} \rightsquigarrow r}{c \mathbin{@} \mathcal{B} \checkmark r}$$

Figure 4.12: Operational semantics of ascription and mode-switch

Ascription (SYN-ASCR) allows us to force the switch from inferring to checking mode. Take for instance the first computation in `let` $x = c_1$ `in` c_2. By Figure 4.9, c_1 will be run in inference mode. If we want to use an available boundary $\mathcal{B}$ to guide the evaluation of c_1, we can use the boundary ascription `let` $x = c_1$ `as` $\mathcal{B}$ `in` c_2.

The CHK-SYN rule is applied to evaluate an inferring computation c in checking mode. Many computations such as function application are neutral, and can be used in either mode. If c does not have a way of exploiting the available boundary information, as is the case for instance when c = `return` (v), the computation is inferred instead, and bound to `x`. If the checking boundary $\mathcal{B}$ is alpha-equal to inferred boundary ∂ `x` the result is returned. If a mismatch is detected, `coerce x` will be performed under boundary ascription $\mathcal{B}$. In the bidirectional type system presented in Figure 4.2, we saw how to confine conversion checking to the λCF-CHK-SYN rule. The pseudo-AML rule PAML-CHK-SYN in Figure 4.3 already suggested the idea of using an unspecified operation, without realising the technique of runners yet, to rectify a boundary mismatch. The present CHK-SYN combines these ideas with operations and runners, and allows us to implement runners handling coerce in the style of `coerce-to` runners in Section 4.1.2. The integration with bidirectional typing allows the evaluation of AML to trigger `coerce` automatically for the user at appropriate times, and the presentation of `coerce` as operation allows the user to exploit local information when handling a particular coercion problem.

$$\text{SYN-TT-VAR}\ \frac{\mathsf{a}^A\ \text{fresh}}{\texttt{tt-var}(A\ \mathsf{type})\ \rightsquigarrow\ \texttt{val}\ (\mathsf{a}^A : A)}$$

$$\text{SYN-TT-ABSTR}\ \frac{v_1 = \mathsf{a}^A : A \qquad v_2 = \mathcal{J}\ \text{or}\ \mathcal{B} \qquad \mathsf{a}^A \notin \mathsf{fvt}(v_2)}{\texttt{tt-abstr}(v_1,\ v_2)\ \rightsquigarrow\ \texttt{val}\ (\{x{:}A\}\ v_2[x/\mathsf{a}^A])}$$

$$\text{SYN-TT-SUBST}\ \frac{v = \mathcal{J}\ \text{or}\ \mathcal{B}}{\texttt{tt-subst}((\{x{:}A\}\ v),\ (t : A))\ \rightsquigarrow\ \texttt{val}\ (v[t/x])}$$

Figure 4.13: Operational semantics of TT variables

We now arrive at the smart constructors for type theory. The treatment of type-theoretic variables (Fig. 4.13) is straightforward. A fresh variable can be created at a given (evaluated) type per SYN-TT-VAR, which ensures that judgements will always have well-typed annotations (Def. 3.1.12). Abstraction is modelled after CF-ABSTR-FWD. The side condition $\mathsf{a}^A \notin \mathsf{fvt}(v_2)$ ensures that no other variable occurring in v_2 can depend on a^A. In presentations of type theory with contexts as lists this corresponds to the restriction that only the last free variable can be bound. As context-free type theories have no explicit context lists, we instead refer to the partial order induced by dependency, similarly to the construction of suitable contexts in Section 3.3.1. Admissibility of substitution is implemented via SYN-TT-SUBST.

One may wonder why all the type theoretic rules are inferring. Creation of a

variable, for instance, could reasonably be expected to be checking, which would relieve the user from the requirement to explicitly provide the type. As we will see in Section 4.4, many type theoretic *derived forms* can be defined to exploit checking information for user convenience. AML is sufficiently expressive that these forms do not need to be treated specially in the core language.

$$\textsc{Syn-tt-mvar}\ \frac{\mathsf{M}^{\mathcal{B}}\ \text{fresh}}{\texttt{tt-mvar}(\mathcal{B}) \ \rightsquigarrow\ \texttt{val}\ (\mathcal{B}\boxed{\widehat{\mathsf{M}^{\mathcal{B}}}})}$$

$$\textsc{Syn-tt-derived}\ \frac{\begin{array}{c} v_1 = \mathcal{B}\boxed{\widehat{\mathsf{M}^{\mathcal{B}}}} \qquad v_2 = \mathcal{J}\ \text{or}\ \mathcal{B}\ \text{or}\ (\texttt{rule}\ N : \mathcal{B}' \Longrightarrow v) \\ \forall \mathsf{N}^{\mathcal{B}'} \in \mathsf{mv}(v_2), \mathsf{N}^{\mathcal{B}'} = \mathsf{M}^{\mathcal{B}} \vee \mathsf{M}^{\mathcal{B}} \notin \mathsf{asm}(\mathcal{B}') \qquad \mathsf{fv}(v_2) = \emptyset \end{array}}{\texttt{tt-derived}(v_1, v_2) \ \rightsquigarrow\ \texttt{val}\ (\texttt{rule}\ M : \mathcal{B} \Longrightarrow v_2[M/\mathsf{M}^{\mathcal{B}}])}$$

$$\textsc{Syn-tt-inst}\ \frac{v = \mathcal{J}\ \text{or}\ \mathcal{B}'\ \text{or}\ (\texttt{rule}\ N : \mathcal{B}' \Longrightarrow v')}{\texttt{tt-inst}((\texttt{rule}\ M : \mathcal{B} \Longrightarrow v), \mathcal{B}\boxed{e}) \ \rightsquigarrow\ \texttt{val}\ (v[e/M])}$$

$$\textsc{Syn-tt-congr}\ \frac{\begin{array}{c} v_l = \mathcal{b}\boxed{\mathsf{S}(e_1, \ldots, e_n)},\quad v_r = \mathcal{b}\boxed{\mathsf{S}(e'_1, \ldots, e'_n)} \\ \text{or}\ \ v_l = \mathcal{b}\boxed{\mathsf{M}^{\mathcal{B}}(e_1, \ldots, e_n)},\quad v_r = \mathcal{b}\boxed{\mathsf{M}^{\mathcal{B}}(e'_1, \ldots, e'_n)} \\ \text{Apply the congruence rule for } \mathsf{S}\ /\ \mathsf{M}^{\mathcal{B}} \text{ to premises } v_1, \ldots, v_n \text{ to obtain } j, \\ \text{checking that } v_1, \ldots, v_n \text{ are the correct premises (Def. 3.1.8, 3.1.10)} \end{array}}{\texttt{tt-congr}(v_l, v_r, v_1, \ldots, v_n) \ \rightsquigarrow\ \texttt{val}\ (j)}$$

Figure 4.14: Operational semantics of metavariables and congruence rules

The treatment of metavariables is very similar to that of variables (Fig. 4.14). Fresh metavariables can be created from well-formed boundaries, and a metavariable can be abstracted to form a derivable rule. The first argument to $\texttt{tt-derived}(v_1, v_2)$ must be a metavariable (judgement), which gets abstracted in v_2, creating a derived rule in the sense of Definition 2.1.17. The abstraction of metavariables includes a side-condition analogous to the abstractability condition for variables, saying that in order to abstract $\mathsf{M}^{\mathcal{B}}$, no other metavariable should depend on $\mathsf{M}^{\mathcal{B}}$. A rule, whether primitive or derivable, can be applied via Syn-tt-inst. As the application of rules proceeds one premise at a time, the result of an instantiation is either a further (derivable) rule, a judgement, or a boundary. Congruence rules do not need to be manually postulated. Instead, $\texttt{tt-congr}(v_l, v_r, \vec{v})$ computes the congruence rule associated to the head symbol of v_l and v_r. If the arguments $\vec{v}$ match the premises of the corresponding context-free congruence rule, the judgement j which equates v_l and v_r is produced.

Two type theoretic values can be tested for syntactic equality via `tt-alpha-equal`. Likewise, `tt-erasure-equal` provides a test for the equality after erasure of conversion terms and assumption sets. Reflexivity of judgemental equality, for types and terms, is implemented modulo erasure from context-free to finitary type theory, in keeping with CF-EqTy-Refl and CF-EqTm-Refl. Conversion has to compute a suitable assumption set. By Theorem 3.2.14, a derivable term judgement never needs to apply

$$\text{Syn-tt-alpha-equal}\ \frac{b = \begin{cases}\texttt{true} & \text{if } v_1 = v_2 \text{ syntactically (per §3.1.1.4)}\\ \texttt{false} & \text{otherwise}\end{cases}}{\texttt{tt-alpha-equal}(v_1,\ v_2) \quad\rightsquigarrow\quad \texttt{val}\ (b)}$$

$$\text{Syn-tt-erasure-equal}\ \frac{b = \begin{cases}\texttt{true} & \text{if } \lfloor v_1 \rfloor = \lfloor v_2 \rfloor \text{ syntactically (per §2.1.1.3)}\\ \texttt{false} & \text{otherwise}\end{cases}}{\texttt{tt-erasure-equal}(v_1,\ v_2) \quad\rightsquigarrow\quad \texttt{val}\ (b)}$$

$$\text{Syn-tt-refl}\ \frac{j = \begin{cases}A_1 \equiv A_2 \text{ by } \{\!\!\}\!\!\} & \text{if } v_1 = (A_1 \text{ type}),\ v_2 = (A_2 \text{ type}), \text{ and } \lfloor A_1 \rfloor = \lfloor A_2 \rfloor\\ t_1 \equiv t_2 : A \text{ by } \{\!\!\}\!\!\} & \text{if } v_1 = (t_1 : A),\ v_2 = (t_2 : A), \text{ and } \lfloor t_1 \rfloor = \lfloor t_2 \rfloor\end{cases}}{\texttt{tt-refl}(v_1,\ v_2) \quad\rightsquigarrow\quad \texttt{val}\ (j)}$$

$$\text{Syn-tt-convert-tm}\ \frac{(t, \gamma) = \begin{cases}(s', \alpha \cup \mathsf{asm}(A) \cup \beta \setminus \mathsf{asm}(s', B)) & \text{if } s = \kappa(s', \beta)\\ (s, \alpha \cup \mathsf{asm}(A) \setminus \mathsf{asm}(s, B)) & \text{otherwise}\end{cases}}{\texttt{tt-convert}((s : A),\ (A \equiv B \text{ by } \alpha)) \quad\rightsquigarrow\quad \texttt{val}\ (\kappa(t, \gamma) : B)}$$

$$\text{Syn-tt-convert-eqtm}\ \frac{\begin{array}{l}(s', \gamma) = \begin{cases}(s'', \beta \cup \mathsf{asm}(A) \cup \zeta \setminus \mathsf{asm}(s'', B)) & \text{if } s = \kappa(s'', \zeta)\\ (s, \beta \cup \mathsf{asm}(A) \setminus \mathsf{asm}(s, B)) & \text{otherwise}\end{cases}\\ (t', \delta) = \begin{cases}(t'', \beta \cup \mathsf{asm}(A) \cup \xi \setminus \mathsf{asm}(t'', B)) & \text{if } t = \kappa(t'', \xi)\\ (t, \beta \cup \mathsf{asm}(A) \setminus \mathsf{asm}(t, B)) & \text{otherwise}\end{cases}\end{array}}{\texttt{tt-convert}((s \equiv t : A \text{ by } \alpha),\ (A \equiv B \text{ by } \beta)) \quad\rightsquigarrow\quad \texttt{val}\ (\kappa(s', \gamma) \equiv \kappa(t', \delta) : B \text{ by } \alpha)}$$

Figure 4.15: Operational semantics of TT equality forms

conversion twice in a row. Successive conversions can be compressed by chaining the successive type equations via transitivity. Therefore two cases are distinguished in Syn-tt-convert-tm and Syn-tt-convert-eqtm. If the term to be converted is already a conversion term, we strip the inner conversion tag and take the residual assumption set into account while computing γ. Otherwise, if the term is an applied symbol, metavariable, or a variable, we leave it unchanged and compute γ accordingly. This "optimisation" is not necessary from a type theoretic point of view, but it ensures that the size of a term computed in AML is comparable to the size of its erased FTT term.

The constructors for boundaries (Fig. 4.16) are simply the translation of the corresponding CFTT rules (Fig. 3.8) to AML. A given judgement can be projected to its boundary via Syn-tt-bdry-of. This primitive realises Theorem 3.2.5, context free presuppositivity.

Pattern matching is the AML implementation of context-free inversion, Theorem 3.2.14. Inversion allows us to reconstruct the premises of a symbol rule. In Pat-sym, each of the argument patterns p_i is matched against the judgement obtained by inverting v and decomposing the head into the heads obtained from the premises. By the inversion theorem, filling $\mathcal{B}_i$ with the head e_i is a derivable judgement, and pattern matching can proceed on $\mathcal{B}_i\boxed{e_i}$. The boundaries $\mathcal{B}_i$ are provided by the current

$$\textsc{Syn-tt-bdry-ty}\ \frac{}{\texttt{tt-bdry-ty} \ \rightsquigarrow\ \texttt{val}\ (\Box\ \mathsf{type})} \qquad \textsc{Syn-tt-bdry-tm}\ \frac{}{\texttt{tt-bdry-tm}(A\ \mathsf{type})\ \rightsquigarrow\ \texttt{val}\ (\Box : A)}$$

$$\textsc{Syn-tt-bdry-eqty}\ \frac{}{\texttt{tt-bdry-eqty}(A\ \mathsf{type},\ B\ \mathsf{type})\ \rightsquigarrow\ \texttt{val}\ (A \equiv B\ \mathsf{by}\ \Box)}$$

$$\textsc{Syn-tt-bdry-eqtm}\ \frac{}{\texttt{tt-bdry-eqtm}(A\ \mathsf{type},\ s : A,\ t : A)\ \rightsquigarrow\ \texttt{val}\ (s \equiv t : A\ \mathsf{by}\ \Box)}$$

$$\textsc{Syn-tt-bdry-of}\ \frac{\mathcal{J} = \mathcal{B}\boxed{e}}{\texttt{tt-bdry-of}(\mathcal{J})\ \rightsquigarrow\ \texttt{val}\ (\mathcal{B})}$$

Figure 4.16: Operational semantics of TT boundaries

CFTT theory $\mathbb{T}$ which is implicitly passed on through evaluation of computations. We require all variables occurring in patterns to be different from one another, and can thus simply take the union of the substitutions obtained from recursive calls. The pattern for metavariable application, Pat-mvar works much the same way, except that the boundary is obtained from the annotation rather than by consulting $\mathbb{T}$. The second case of Theorem 3.2.14 treats the case where a conversion was required to derive a term judgement at the given type, and is implemented in Pat-convert. The first pattern is matched against the stripping of t at its natural type, the second gives access to the equation connecting the given type to the natural type, implementing Lemma 3.2.13. The AML conversion rule Syn-tt-convert-tm compresses successive conversions via transitivity. The stripping of the converted term against which p_1 is matched can thus be obtained in a single step, i.e. $\mathscr{s}(t) = t'$ and $\mathscr{z}(t) = \alpha$. The boundary patterns, Pat-abstr, and Pat-rule are likewise justified by inversion.

4.3.3 Toplevel

We now turn our attention to the evaluation of programs, i.e. lists of toplevel commands defined in Figure 4.18. We give a small step operational semantics for programs working on a runner stack $\mathcal{R}$, a CFTT theory $\mathbb{T}$, and a list of top-commands cs. When we defined the evaluation functions synth-comp and check-comp at the beginning of this section, we mentioned that they pass the current theory $\mathbb{T}$ on to recursive calls, and in Pat-sym we saw how $\mathbb{T}$ is used for the purposes of inversion, but we have yet to discuss where $\mathbb{T}$ originates.

Evaluation of a program starts with the empty theory $\mathbb{T}_0 = \{\}$, and new rules can be added via Top-rule. In order to maintain the invariant that $\mathbb{T}$ is a standard context-free type theory (Def. 3.1.14), we have to ensure that there is exactly one rule per symbol. This is taken care of by the first side condition. The derivability of the boundaries $\mathcal{B}_i$ and $\mathscr{b}$ is guaranteed by requiring v to be a derivable rule boundary. The `rule` command extends $\mathbb{T}$ with the economic rule v' obtained by filling the rule boundary v with the symbol R applied to the heads of the premises if $\mathscr{b}$ is an object boundary, or with the appropriate assumption set in case $\mathscr{b}$ is an equation. Relying on

Proposition 3.2.9 and Proposition 3.2.10, we use the economic rules, which leave out the premise requiring the well-formedness of the boundary of the conclusion.

A toplevel let-binding is evaluated by wrapping c in the current runner stack $\mathcal{R}$, evaluating it to a value v', and substituting v' in the rest of the top-commands.

A typing declaration for an operation has no operational meaning and is simply skipped over during evaluation.

The runner stack is extended in TOP-RUNNER by appending v as the innermost runner that gets wrapped around an argument computation c.

$$\textsc{Pat-var}\ \frac{}{v \ \sim\ \mathtt{p\text{-}var}(x)\ \rhd\ \{x \mapsto v\}}$$

$$\textsc{Pat-ignore}\ \frac{}{v \ \sim\ \mathtt{p\text{-}ignore}\ \rhd\ \{\}}$$

$$\textsc{Pat-sym}\ \frac{\begin{array}{c} v = \mathcal{b}\boxed{e} \quad \text{and} \quad e = S(e_1, \ldots, e_n) \\ \mathbb{T}(S) = (\mathtt{rule}\ M_1 : \mathcal{B}_1 \Longrightarrow \ldots \Longrightarrow \mathtt{rule}\ M_n : \mathcal{B}_n \Longrightarrow j) \\ \text{let } I = [\mathsf{M}_1^{\mathcal{B}_1} \mapsto e_1, \ldots, \mathsf{M}_n^{\mathcal{B}_n} \mapsto e_n] \\ (I_{(i)*}\mathcal{B}_i\boxed{e_i}) \ \sim\ p_i \ \rhd\ \sigma_i \quad \text{for } i = 1, \ldots, n \end{array}}{v \ \sim\ \mathtt{p\text{-}sym}(S, p_1, \ldots, p_n)\ \rhd\ \sigma_1 \cup \ldots \cup \sigma_n}$$

$$\textsc{Pat-mvar}\ \frac{\begin{array}{c} v = \mathcal{b}\boxed{e} \quad \text{and} \quad e = \mathsf{M}^{\mathcal{B}}(t_1, \ldots, t_n) \\ \mathcal{B} = \{x_1 : A_1\} \cdots \{x_n : A_n\}\ \mathcal{b}' \\ \mathcal{b}\boxed{\widehat{\mathsf{M}^{\mathcal{B}}}} \ \sim\ p_M \ \rhd\ \sigma_M \qquad (t_i : A_i[\vec{t}_{(i)}/\vec{x}_{(i)}]) \ \sim\ p_i \ \rhd\ \sigma_i \quad \text{for } i = 1, \ldots, n \end{array}}{v \ \sim\ \mathtt{p\text{-}mvar}(p_M, p_1, \ldots, p_n)\ \rhd\ \sigma_M \cup \sigma_1 \cup \ldots \cup \sigma_n}$$

$$\textsc{Pat-convert}\ \frac{\begin{array}{c} t = \kappa(t', \alpha) \\ (t' : \tau(t)) \ \sim\ p_1 \ \rhd\ \sigma_1 \qquad (A \equiv \tau(t) \text{ by } \alpha) \ \sim\ p_2 \ \rhd\ \sigma_2 \end{array}}{(t : A) \ \sim\ \mathtt{p\text{-}convert}(p_1, p_2)\ \rhd\ \sigma_1 \cup \sigma_2}$$

$$\textsc{Pat-bdry-ty}\ \frac{}{(\square \text{ type}) \ \sim\ \mathtt{p\text{-}bdry\text{-}ty}\ \rhd\ \{\}}$$

$$\textsc{Pat-bdry-tm}\ \frac{(A \text{ type}) \ \sim\ p \ \rhd\ \sigma}{(\square : A) \ \sim\ \mathtt{p\text{-}bdry\text{-}tm}(p)\ \rhd\ \sigma}$$

$$\textsc{Pat-bdry-eqty}\ \frac{(A \text{ type}) \ \sim\ p_1 \ \rhd\ \sigma_1 \qquad (B \text{ type}) \ \sim\ p_2 \ \rhd\ \sigma_2}{(A \equiv B \text{ by } \square) \ \sim\ \mathtt{p\text{-}bdry\text{-}eqty}(p_1, p_2)\ \rhd\ \sigma_1 \cup \sigma_2}$$

$$\textsc{Pat-bdry-eqtm}\ \frac{\begin{array}{c} (s : A) \ \sim\ p_1 \ \rhd\ \sigma_1 \\ (t : A) \ \sim\ p_2 \ \rhd\ \sigma_2 \qquad (A \text{ type}) \ \sim\ p_3 \ \rhd\ \sigma_3 \end{array}}{(s \equiv t : A \text{ by } \square) \ \sim\ \mathtt{p\text{-}bdry\text{-}eqtm}(p_1, p_2, p_3)\ \rhd\ \sigma_1 \cup \sigma_2 \cup \sigma_3}$$

$$\textsc{Pat-abstr}\ \frac{(A \text{ type}) \ \sim\ p \ \rhd\ \sigma}{(\{x : A\}\ v) \ \sim\ \mathtt{p\text{-}abstr}(p)\ \rhd\ \sigma}$$

$$\textsc{Pat-rule}\ \frac{\mathcal{B} \ \sim\ p \ \rhd\ \sigma}{(\mathtt{rule}\ M : \mathcal{B} \Longrightarrow v) \ \sim\ \mathtt{p\text{-}rule}(p)\ \rhd\ \sigma}$$

Figure 4.17: Pattern matching

Top-rule

$$\frac{R \notin |\mathbb{T}| \qquad v = (\texttt{rule}\ M_1 : \mathscr{B}_1 \Longrightarrow \ldots \Longrightarrow \texttt{rule}\ M_n : \mathscr{B}_n \Longrightarrow \mathscr{b}) \quad \text{let } v' \text{ be the economic rule associated to the rule boundary } v \text{ for } R \text{ (Def. 3.1.5, 3.1.6)}}{\mathcal{R}, \mathbb{T} \mid \texttt{rule}(R, v)\ ;\ cmds \quad \longrightarrow \quad \mathcal{R}, \mathbb{T} \cup \{R \mapsto v'\} \mid cmds[v'/R]}$$

Top-let

$$\frac{\mathcal{R}[c] \quad \rightsquigarrow \quad \texttt{val}\ (v)}{\mathcal{R}, \mathbb{T} \mid \texttt{let}(x,\ c)\ ;\ cmds \quad \longrightarrow \quad \mathcal{R}, \mathbb{T} \mid cmds[v/x]}$$

Top-operation

$$\frac{}{\mathcal{R}, \mathbb{T} \mid \texttt{operation}(op,\ op\text{-}mlsig)\ ;\ cmds \quad \longrightarrow \quad \mathcal{R}, \mathbb{T} \mid cmds}$$

Top-runner

$$\frac{\text{if } \mathcal{R}[c] = \texttt{with}\ v_1\ \texttt{run} \ldots \texttt{with}\ v_n\ \texttt{run}\ c \quad \text{let } \mathcal{R}'[c] = \texttt{with}\ v_1\ \texttt{run} \ldots \texttt{with}\ v_n\ \texttt{run}\ \texttt{with}\ v\ \texttt{run}\ c}{\mathcal{R}, \mathbb{T} \mid \texttt{with}(v)\ ;\ cmds \quad \longrightarrow \quad \mathcal{R}', \mathbb{T} \mid cmds}$$

Figure 4.18: Operational semantics of toplevel commands

4.4 Standard derived forms

The purpose of AML as defined in Section 4.2 and Section 4.3 is to give a small core language that can express the mechanisms that were empirically found to be useful for working with context-free type theories in the Andromeda prover. In this section we present a number of derived forms that offer further user convenience as "syntactic sugar"[3]. Each derived form is given an equivalent form in AML. The latter defines the meaning of the former and induces its operational semantics. In some cases, we will illustrate the behaviour of the sugared syntax by describing its induced operational semantics in the form of a rule. We will take the liberty to omit certain cases, as the formal semantics is always the induced one. Some derived forms require to set up a preamble with toplevel commands, such as operation declarations or rule definitions. In such cases, it is understood that the preamble is only included once, and the definition of the derived computation follows the preamble after a comment line `(* ---- *)`. As the derived syntax sometimes considerably reduces the amount of code required to express an idea, we will use earlier derived forms in the definition of later derived forms. This section will thus also serve us as a source of examples of AML code.

Derived forms for boundary formation (Fig. 4.19) can beneficially use available information to guide computations in checking mode via boundary ascriptions. The derived form $\square : c$ for term boundaries for instance requires c to evaluate to a type, so we run c under the constraint that it has to fill the boundary $\square$ `type`. Substitution requires its first argument to be an abstraction, and the second a term of matching type. The derived form $v\{c\}$ recovers the type on the abstraction v by pattern matching, and runs c with the resulting boundary ascription before performing the actual substitution via `tt-subst`. Similarly, conversion of a term t along an equality e requires the type of t to match the left hand side of e.

The induced operational semantics (Fig. 4.20) of boundaries and substitution only display the value case for all involved computations to avoid burdening the reader with an exponential number of rules. They intuitively capture the expected checking behaviour of boundary formation and substitution, but by presenting them as derived forms we avoid having to consider operation cases for each sub-computation. Instead, we rely on the behaviour of Syn-let-op if any of the let-bound computations performs an operation. The evaluation of substitution is reminiscent of Bidi-App, the usual application rule in bidirectional systems: to infer the type of an application, first infer the function, then check the argument, finally construct the result type.

The previous examples of derived forms were all synthesising, with information flowing internally to the construction, from one argument to another. Figure 4.21 demonstrates how to use operations and runners to define derived forms that can be run in checking mode. Derived checking forms are defined with the help of operations, as operations run in checking mode allow programs to access the current checking

[3]The terminology *derived form* is standard (Milner et al. 1990), but to avoid confusion with type theoretic derivable rules we will use the term "sugared forms" when such confusion could arise.

Derived form	Equivalent form	Meaning	Mode
Derived computation $C \ni c$ ::=			
$\square : c$	`let x = c as (□ type) in` `tt-bdry-tm x`	term boundary	$\rightsquigarrow$

$c_1 \equiv c_2$ `by` $\square$	`let x = c₁ as (□ type) in`
	`let y = c₂ as (□ type) in`
	`tt-bdry-eqty x y`
		type eq. boundary	$\rightsquigarrow$

$c_1 \equiv c_2 : c_3$ `by` $\square$	`let A = c₃ as (□ type) in`
	`let b-A = tt-bdry-tm A in`
	`let z₁ = c₁ as b-A in`
	`let z₂ = c₂ as b-A in`
	`tt-bdry-eqtm z₁ z₂ A`
		term eq. boundary	$\rightsquigarrow$

$v\{c\}$	`match v with {_ : ?A} _ →`
	`  let b-A = tt-bdry-tm A in`
	`  let y = c as b-A in`
	`  tt-subst v y`
		substitution	$\rightsquigarrow$

`convert-tm` $c\ v$	`let B = ∂v in`
	`match B with ?A ≡ _ by □ →`
	`  let b-A = tt-bdry-tm A in`
	`  let x = c as b-A in`
	`  tt-convert x v`
		conversion	$\rightsquigarrow$

Figure 4.19: Syntax of derived boundary, substitution, and conversion computations

Syn-Bdry-Tm-val
$$\frac{c \ @\ \square\ \mathsf{type} \ \checkmark\ \mathtt{val}\ (A\ \mathsf{type})}{\square : c \ \rightsquigarrow\ \mathtt{val}\ (\square : A)}$$

Syn-Bdry-EqTy-val-val
$$\frac{c_1 \ @\ \square\ \mathsf{type} \ \checkmark\ \mathtt{val}\ (A\ \mathsf{type}) \qquad c_2 \ @\ \square\ \mathsf{type} \ \checkmark\ \mathtt{val}\ (B\ \mathsf{type})}{c_1 \equiv c_2\ \mathtt{by}\ \square \ \rightsquigarrow\ \mathtt{val}\ (A \equiv B\ \mathsf{by}\ \square)}$$

Syn-Bdry-EqTm-val-val-val
$$\frac{\begin{array}{c} c_3 \ @\ \square\ \mathsf{type} \ \checkmark\ \mathtt{val}\ (A\ \mathsf{type}) \\ c_1 \ @\ \square : A \ \checkmark\ \mathtt{val}\ (s : A) \qquad c_2 \ @\ \square : A \ \checkmark\ \mathtt{val}\ (t : A) \end{array}}{c_1 \equiv c_2 : c_3\ \mathtt{by}\ \square \ \rightsquigarrow\ \mathtt{val}\ (s \equiv t : A\ \mathsf{by}\ \square)}$$

Syn-Subst-val
$$\frac{v = \mathcal{B} \text{ or } v = \mathcal{J} \qquad c \ @\ \square : A \ \checkmark\ \mathtt{val}\ (t : A)}{(\{x{:}A\}\ v)\{c\} \ \rightsquigarrow\ \mathtt{val}\ (v[t/x])}$$

Figure 4.20: Induced operational semantics of boundaries and substitution

Creation of a fresh variable:

`fresh`

```
operation genfresh : unit → judgement
with runner genfresh () bdry_opt →
    match bdry_opt with
    | Boundary (□ : ?A) → tt-var A
end
(* ---- definition of fresh : ---- *)
genfresh ()
```

✓

$$\textsc{Chk-fresh}\ \frac{\mathsf{a}^A \text{ fresh}}{\mathtt{fresh} \ @\ \square : A \ \checkmark\ \mathtt{val}\,(\mathsf{a}^A : A)}$$

Figure 4.21: Syntax and induced operational semantics of derived fresh computation

boundary. The derived form `fresh` creates a new variable with the help of a `genfresh` operation. The runner for `genfresh` extracts the type from the checking boundary and returns a new variable created via `tt-var`. After this preamble is set up, the form `fresh` is defined as `genfresh ()`.

The construction of abstractions in AML happens in a somewhat roundabout way: a fresh variable has to be created and let-bound, a construction involving this variable is performed and let-bound, and finally the variable is abstracted. AML provides the necessary primitives to perform these steps, but the pattern is so common that we provide additional syntactic sugar for it in Figure 4.22. Untyped abstraction, written as $\{x\}\ c$, is checking and propagates typing from the binder of the checking boundary to the computation, using an auxiliary operation much like `fresh` did. Typed abstraction $\{x : c_1\}\ c_2$ is our first example of a neutral derived form. It immediately triggers the operation `abstrT` to see whether a checking boundary if present. In synthesis mode, it simply creates a fresh variable at the provided type and synthesises c_2. In checking mode, the boundary is decomposed to read off the expected abstraction type `A`. As c_2 expects a variable of type `C`, which was provided on the abstraction annotation as c_1, we try to ascribe the type `C` to `a`. If the two types coincide, the ascription will succeed, otherwise a a coercion is performed. Finally we can run c_2 in checking mode, further propagating the information, and abstract the resulting value. The induced dynamic behaviour is summarised in Figure 4.23.

4.4.1 Rule application and formation

The propagation of boundaries is particularly useful when dealing with rules specific to a given type theory.

The derived form for rule application (Fig. 4.24) allows us to write several iterated metavariable instantiations by juxtaposition. Typically c_R is either a further rule application or a rule. In this situation each of the subsequent rule applications in c_R will in turn evaluate its argument in checking mode against the boundary of the corresponding rule premise instantiated with the preceding arguments.

Meaning: Derived form	Equivalent form	Mode

Derived computation $C \ni c$ ::=

untyped abstr.: $\{x\}\ c$ — Mode: ✓

```
operation abstrU
    : (judgement → judgement) → judgement
with runner abstrU x_c bdry_opt →
  match bdry_opt with
  | Boundary ?B → match bdry with {_ : ?A} _ →
    let a = tt-var A in
    let z = (x_c a) as bdry{return a} in
    tt-abstr a z
end
(* ---- definition of {x} c : ---- *)
abstrU (fun x → c)
```

typed abstr.: $\{x : c_1\}\ c_2$ — Mode: ⇝, ✓

```
operation abstrT
    : (unit → judgement) ∗ (judgement → judgement)
    → judgement
with runner abstrT (ty, body) bdry_opt →
  let C = ty () as (□ type) in
  match bdry_opt with
  | Boundary ?bdry → (match bdry with {_: ?A} _ →
    let a = tt-var A in
    let c = a as C in
    let y = body c as bdry{return a} in
    tt-abstr a y)
  | Unbounded →
    let c = tt-var C in
    let y = body c in
    tt-abstr c y
end
(* ---- definition of {x:c1} c2 : ---- *)
abstrT ((fun _ → c1), (fun x → c2))
```

Figure 4.22: Syntax of derived abstraction computations

$$\dfrac{\mathsf{a}^A \text{ fresh} \qquad c[(\mathsf{a}^A : A)/x] \ @\ \mathcal{B}[\mathsf{a}^A/x] \quad \checkmark \quad \mathtt{val}\,(\mathcal{J})}{(\{x\}c)\ @\ (\{x{:}A\}\ \mathcal{B}) \quad \checkmark \quad \{x{:}A\}\ \mathcal{J}[x/\mathsf{a}^A]}\ \textsc{Chk-abstr-untyped}$$

$$\dfrac{\mathsf{a}^A \text{ fresh} \qquad \begin{array}{l} c_1\ @\ \square\ \mathsf{type} \quad \checkmark \quad \mathtt{val}\,(A\ \mathsf{type}) \\ c_2[(\mathsf{a}^A : A)/x] \quad \rightsquigarrow \quad \mathtt{val}\,(v) \qquad v = \mathcal{B} \text{ or } \mathcal{J} \end{array}}{\{x{:}c_1\}\ c_2 \quad \rightsquigarrow \quad \mathtt{val}\,(\{x{:}A\}\ v[x/\mathsf{a}^A])}\ \textsc{Syn-abstr-typed}$$

$$\dfrac{\begin{array}{c} c_1\ @\ \square\ \mathsf{type} \quad \checkmark \quad \mathtt{val}\,(C\ \mathsf{type}) \qquad \mathsf{a}^A \text{ fresh} \\ (\mathsf{a}^A : B)\ @\ \square : C \quad \checkmark \quad \mathtt{val}\,(t : C) \qquad c_2[(t : C)/x]\ @\ \mathcal{B}[\mathsf{a}^A/x] \quad \checkmark \quad \mathtt{val}\,(\mathcal{J}) \end{array}}{(\{x{:}c_1\}\ c_2)\ @\ \{x{:}A\}\ \mathcal{B} \quad \checkmark \quad \{x{:}A\}\ \mathcal{J}[x/\mathsf{a}^A]}\ \textsc{Chk-abstr-typed}$$

Figure 4.23: Induced operational semantics of abstraction

$c_R\ c_{arg}$

```
let R = c_R in
match R with rule (_ : ?bdry) _ →
  let a = c_arg as bdry in
  tt-inst R a
```

$\rightsquigarrow$

$$\dfrac{c_R \quad \rightsquigarrow \quad \mathtt{val}\,(\mathtt{rule}\,(M : \mathcal{B}) \Longrightarrow v) \qquad c_{arg}\ @\ \mathcal{B} \quad \checkmark \quad \mathtt{val}\,(\mathcal{B}\boxed{e})}{c_R\ c_{arg} \quad \rightsquigarrow \quad \mathtt{val}\,(v[e/M])}\ \textsc{Syn-TT-apply-val-val}$$

Figure 4.24: Syntax and induced operational semantics of rule application

$\mathtt{derive}(M : c_{bdry})\ c$

```
let bdry = c_bdry in
let M = tt-mvar bdry in
let x = c in
tt-derived M x
```

$\rightsquigarrow$

$$\dfrac{c_{bdry} \quad \rightsquigarrow \quad \mathtt{val}\,(\mathcal{B}) \qquad \mathsf{M}^{\mathcal{B}} \text{ fresh} \qquad c[\mathcal{B}\boxed{\widehat{\mathsf{M}^{\mathcal{B}}}}/M] \quad \rightsquigarrow \quad \mathtt{val}\,(v)}{\mathtt{derive}(M : c_{bdry})\ c \quad \rightsquigarrow \quad \mathtt{val}\,(\mathtt{rule}\ M : \mathcal{B} \Longrightarrow v[M/\mathsf{M}^{\mathcal{B}}])}\ \textsc{Syn-derivable-rule-val-val}$$

Figure 4.25: Syntax and induced operational semantics of derivable rules

Formation of derivable rules (Fig. 4.25) is similar to the synthesis of a typed abstraction, coupling the creation and abstraction of metavariables.

4.4.2 Handling syntactic equality

The presence of conversion terms $\kappa(s, \alpha)$ in context-free type theories causes syntactic equality to be a slightly stricter notion than syntactic equality of finitary type theories. Lemma 3.2.17 allows us to use modify the head of a judgement $\mathcal{J}$ to fit a prescribed boundary $\mathcal{B}$, so long as the boundary of $\mathcal{J}$ matches $\mathcal{B}$ up to erasure. The AML code in Appendix A implements Lemma 3.2.17 as `boundary-convert`[4], and installs a corresponding toplevel runner `boundary-converter`. The derived forms for substitution, conversion, and boundary formation all run their arguments with suitable ascriptions so that a boundary mismatch will trigger a coercion that can be handled by `boundary-converter`. This fully relieves the user of the burden of manually having to adjust conversion terms, and allows us to work transparently as if we were using finitary type theory.

4.4.3 Recovering λCF-Lambda

In the introduction to bidirectional typing (§4.1.1), we encountered the bidirectional typing and elaboration rules for lambda abstraction, Bidi-Lambda and λCF-Lambda, which we recall here.

$$\textsc{Bidi-Lambda}\ \frac{\Gamma, x{:}A \vdash b \ \Leftarrow B}{\Gamma \vdash \lambda(x.b) \ \Leftarrow \Pi(x{:}A.B)}$$

$$\lambda\textsc{CF-Lambda}\ \frac{\mathsf{a}^A \text{ fresh} \qquad \vdash b[\mathsf{a}^A/x] \ \Leftarrow B[\mathsf{a}^A/x] \gg b' : B[\mathsf{a}^A/x]}{\vdash \lambda(x.b) \ \Leftarrow \Pi(x{:}A.B) \gg \lambda(x{:}A.b') : \Pi(x{:}A.B)}$$

To represent dependent products and lambda in AML, we first have to declare the corresponding rules, displayed in Figure 4.26. We can then recover the checking behaviour of Bidi-Lambda in a couple of lines of AML code. We follow the by-now-familiar recipe of using an operation to access the checking boundary, providing the body of the lambda wrapped in a closure as argument. Once the boundary Π-type is deconstructed, we use the derived for rule application (Fig. 4.24) to apply the λ rule to arguments `A`, `absB`, and `clos ()`. This will run all arguments in checking mode. In particular, the closure containing the body of the lambda will be evaluated in checking mode against the correct boundary. Following the recipe laid out by Bidi-Lambda we thus recover the elaboration behaviour of λCF-Lambda for free as AML evaluates `lambda`c_b to the fully annotated judgement.

[4]The implementation uses `let rec`, which can easily be defined formally in AML, and is available in the Andromeda 2 implementation of AML.

lambda(c_b)

```
let pi-rule-boundary =
    derive (A : □ type)
    derive (B : {x : return A} □ type)
    □ type
rule Π pi-rule-boundary

let lambda-rule-boundary =
    derive (A : □ type)
    derive (B : {x : return A} □ type)
    □ : Π (return A) (return B)
rule λ lambda-rule-boundary

operation lam-chk : (unit → judgement) → judgement
with runner lam-chk clos bdry_opt →
  match bdry_opt with Boundary (□ : Π(?A, ?absB)) →
    (return λ ) (return A) (return absB) (clos ())
end
(* ---- definition of lambda(c_b) : ---- *)
lam-chk (fun _ → c_b)
```

✓

$$\textsc{Chk-Lambda-val}\ \frac{c_b \ @\ \{x{:}A\}\ \square : B \quad \checkmark \quad \{x{:}A\}\ b : B}{\texttt{lambda}(c_b)\ @\ \Pi(A,\{x\}B) \quad \checkmark \quad \lambda(A,\{x\}B,\{x\}b) : \Pi(A,\{x\}B)}$$

Figure 4.26: Syntax and induced operational semantics of checking lambda

4.5 On soundness & completeness

AML is conceived as a metalanguage for the development of context-free type theories. The operational semantics implicitly assume an abstract implementation of the infrastructure of context-free type theories, such as a datatype for syntax, judgements, rules et cetera. If such an implementation is provided via a nucleus, can we then say that in good faith that judgements produced by AML *are* context-free judgements? Such a statement amounts to a soundness theorem for AML with respect to derivable context-free judgements. A vacuous way to achieve soundness is to implement the empty language, that does nothing. While this is clearly not the case for AML, it indicates that besides soundness, we also want a guarantee of completeness with respect to derivable context-free judgements. In this section, we give a brief sketch of soundness and completeness of AML for context-free type theories.

Claim 4.5.1 (AML Soundness)**.** *Let* $\mathbb{T}$ *be a standard context-free type theory. If* $\mathbb{T} \mid c \rightsquigarrow \texttt{val}\,(\mathcal{J})$*, then* $\vdash \mathcal{J}$ *is derivable in* $\mathbb{T}$*.*

A proof of this claim would proceed by an appropriate computational induction (Pretnar 2010) for operations and runners over the AML evaluation relation. We would have to generalise the statement to roughly state that judgement and boundary values are derivable, and for an operation $\texttt{op}(op, v_1, v_2, x.c_\kappa)$, v_1 and v_2 are derivable, and

for any derivable value v, $c_\kappa[v/x]$ evaluates to a derivable value, where we extend the notion of derivability to other values in the obvious way.

Claim 4.5.2 (AML Completeness)**.** *Given a standard context-free type theory $\mathbb{T}$, if $\mathcal{J}$ has well-typed annotations and $\vdash_{\mathbb{T}} \mathcal{J}$, then there exists an AML program c such that $\mathbb{T} \mid c \rightsquigarrow \mathtt{val}\,(\mathcal{J}')$ and $\lfloor \mathcal{J} \rfloor = \lfloor \mathcal{J}' \rfloor$.*

We would prove this statement by hoisting out the free (meta-) variables occurring in $\mathcal{J}$ into let-bindings while respecting their dependency order, using the evidence provided by the well-typed annotations to create variables via `tt-var`, and constructing the AML program by induction over the typing derivation of $\mathcal{J}$. The reason we can only recover $\mathcal{J}$ up to erasure is that the conversion rule of AML applies transitivity on the fly in order to compress successive conversions.

We expect the proof of soundness to be a standard application of proof techniques for algebraic effects and handlers (Pretnar 2010; Bauer and Pretnar 2014; Ahman and Bauer 2019; Lukšič 2020), and the proof of completeness to be straightforward. Detailed proofs of both claims are left as future work.

4.6 AML in Andromeda 2

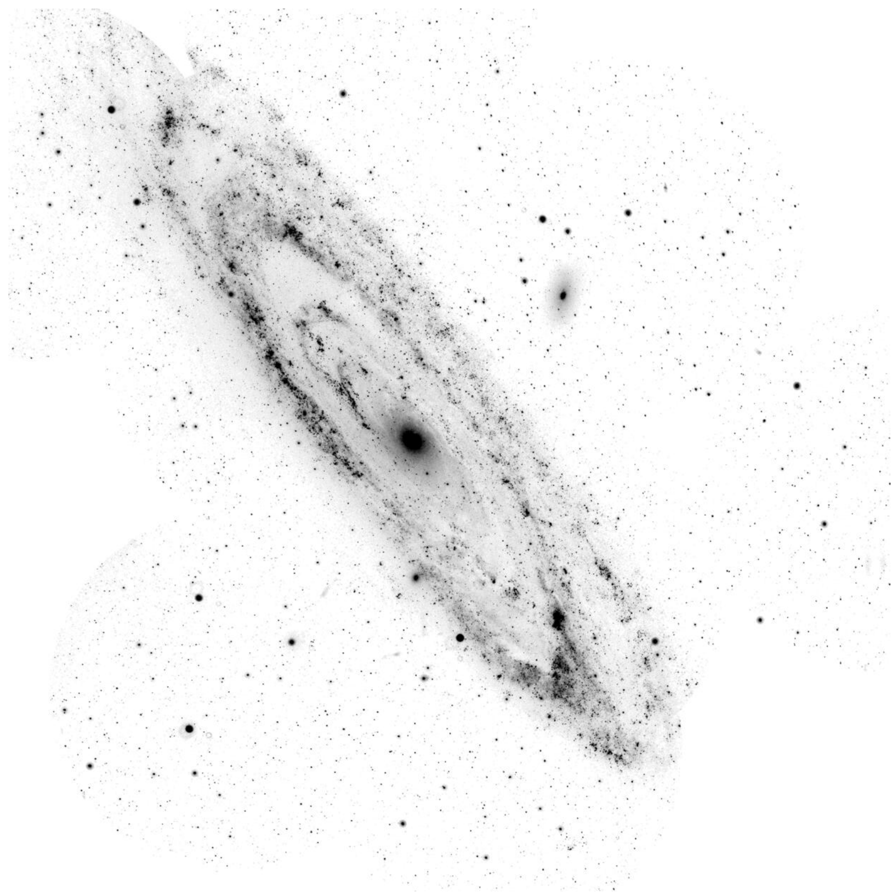
The Andromeda 2 prover (Bauer, Haselwarter and Petković Komel 2021) is an implementation of a variant of AML. The nucleus, which provides the interface for the evaluation of the type theoretic smart constructors, is written in 3000 lines of OCaml code, which closely corresponds to the rules of context-free type theories. The exact implementation of the nucleus is best appreciated by consulting the source code[5]. Essentially, it amounts to a transcription of the datatypes, functions, and algorithms corresponding to the proofs of Chapter 3. As the syntax for context-free type theories is rather verbose compared to that of finitary type theories, Andromeda 2 applies the cf-to-tt erasure of judgements and reconstruction of a suitable context on the fly when printing a type theoretic value.

Andromeda 2 extends core AML with user definable data types, general recursive definitions, references, and exceptions. It comes equipped with Hindley-Milner style static type system helps the user avoid programming errors, and represents values constructed by the nucleus as the abstract types of judgements, boundaries, and rules. The surface language syntax resembles that of AML with the standard derived forms, but is more permissive in places, allowing the application of functions to computations and similar standard relaxations of syntax for fine-grained call-by-value.

Andromeda 2 evolved from Andromeda 1, an experimental implementation of extensional type theory with algebraic effects and handlers. We draw a more detailed comparison in Section 5.1.2.

An example of a formalisation using Andromeda 2 is included in Appendix B. Further examples, such as a definition of the calculus of constructions can be found in the `theories/` subfolder of the Andromeda source code.

[5] https://github.com/Andromedans/andromeda



The Andromeda galaxy, M31. Source: NASA.

Chapter 5

Conclusion

Our quest for an effective metatheory for type theory has lead us to present and study three languages. In Chapter 2, we gave a general definition of a broad class of finitary type theories and proven that it satisfies the expected desirable type theoretic metatheorems. In Chapter 3, we introduced a context-free formulation of type theories and demonstrated that this definition satisfies further metatheorems that were previously lost, notably strengthening and good inversion principles. Context-free type theories were defined with an eye towards implementation, as the annotation discipline for variables allows for the use of an effectful metalanguage. In Chapter 4 we saw the Andromeda metalanguage, an effectful language with bidirectional evaluation for context-free type theories.

5.1 Related work

We will now discuss work related to finitary type theories and to AML, as the related work for context free type theories, most importantly (Geuvers et al. 2010), has been addressed in Chapter 3.

5.1.1 Finitary type theories

Finitary type theories (FTT) were developed concurrently with several other general frameworks for type theory. There are different approaches to the study of formal systems such as logics and type theories, ranging from syntactic (Cartmell 1978; Harper, Honsell et al. 1993) to semantic (Fiore and Mahmoud 2014; Isaev 2016) characterisations. To reasonably delimit the scope of this discussion we shall focus on those that (i) are sufficiently expressive to faithfully represent type theories, and (ii) are sufficiently restrictive to provide useful expected results about type theories.

General dependent type theories The closest relative are general dependent type theories (Bauer, Haselwarter and Lumsdaine 2020), which we proposed together with Bauer and Lumsdaine. Finitary and general dependent type theories (GDTT) have

more in common than divides them. FTT can be seen as a bridge from GDTT to context-free type theories (CFTT). As context-free type theories in turn are intended as the theoretical underpinning of AML, the choice was made to restrict arities of rules and symbols to be finite, which allows for a direct representation as syntax. An arity of a symbol or a rule in GDTT on the other hand can be indexed by an arbitrary set (or type, actually). The finitary restriction is thus somewhat coincidental, and we expect that it should be possible to generalise much of the treatment of FTT and possibly CFTT to arbitrary arities. The treatment of variables in FTT follows a locally-nameless discipline, while GDTT uses shape systems both for free and bound variables. Metavariables represent a separate syntactic category in FTT while they are treated as particularly simple symbols in GDTT. The GDTT approach offers a streamlined theoretical development, as the same specific and congruence rules are used both for symbols and metavariables, and can be viewed as providing a semantic account of FTT's metavariable contexts as theory extensions. The rôle of metavariables in proof assistants however is rather distinct from that of rules, and we chose not to conflate the two in FTT. Finally, the levels of well-formedness of the two formalisms are different. GDTT places fewer restrictions on the rules of raw type theories, while a raw FTT already satisfies presuppositivity.

Logical frameworks Perhaps the most prominent family of systems for representing logics are logical frameworks (Harper, Honsell et al. 1993; Pfenning 2001). Logical frameworks have spawned a remarkably fruitful line of work (Cervesato and Pfenning 2002; Watkins et al. 2003; Cousineau and Dowek 2007) and several implementations exist (Pfenning and Schürmann 1999; Pientka and Dunfield 2010), and have been used to study the metatheory of deductive systems and programming languages (Dunfield 2014; Pientka 2015). In concurrent work to the development of GDTTs and FTTs, Uemura (Uemura 2019) and Harper (Harper 2021) recently proposed frameworks with the purpose of representing type theories. We will thus focus on these two systems.

Both Uemura's LF (*ULF* for short), and Harper's Equational LF (henceforth *EqLF*) extend previous frameworks by the addition of an equality type satisfying reflection to judgemental equality at the framework level, and Uemura includes a substantial development of a general categorical semantics. Uemura's account of type theory can be compared to FTT along several axes. In one way, ULF is more expressive than FTT. While FTT allows only one judgement form for types, terms, and their equalities, ULF can capture theories with different judgement forms, such as the fibrancy judgement of the homotopy type system or two-level type theory (Voevodsky 2013; Annenkov et al. 2019), or the face formulas of cubical type theory (Cohen et al. 2016). While it may be possible to reconstruct some type theories expressible in ULF via the use of universes in FTT, a careful analysis would be required to show that the account is faithful, for instance by showing that it is sound and complete for derivability. Conversely, every standard finitary type theory is expressible in ULF. The translation is straightforward, and we take this as a sign that both ULF and FTT achieve their goal of giving a "natural" account of type theory.

Raw finitary type theories on the other hand are not directly expressible in ULF or in EqLF. Frequently, accounts of type theory present rules that are not standard, most often because a symbol does not record all of the metavariables introduced by its premises as arguments. But it is also standard practice to have only one notation for say dependent products which may occur at more than one sort, as is done in (Martin-Löf 1982; Harper 2021), or give a general cumulativity rule allowing the silent inclusion of types from one sort into another (Luo 1990; Uemura 2019). One may of course take the view that such presentations are not *really* type theories and should be read with full annotations inserted. It is usually understood that such an annotated presentation can be given, and by including the right set of equations the original calculus can be recovered (Harper and Pollack 1991). Proofs that an unannotated theory is equivalent to a fully annotated one are hard labour (Streicher 1991, Theorem 4.13). Finitary type theories can thus serve to study the elaboration of such unannotated to a standard FTT or ULF presentation. One such useful general result can already be found in (Bauer, Haselwarter and Lumsdaine 2020), where it is shown that every raw type theory, possibly containing cyclic dependencies between rules, is equivalent to a well-founded one. The assumption of well-founded stratification is hardwired in ULF through the definition of a signature and in EqLF trough the inductive construction of a context serving as signature, so that such a theorem could not even be stated in ULF or EqLF.

An alternative perspective is offered by the observation that Harper's Equational LF *almost* forms a standard finitary type theory. In Appendix B.1 we suggest a standard CFTT presentation of EqLF that we formalised in Andromeda 2. As an implementation of standard type theories, Andromeda 2 is well suited for working with a type theory with equality reflection such as EqLF. Appendix B.2 develops most of the examples from (Harper 2021), and with a few simple local definitions and runners we can transcribe the examples almost verbatim.

In our opinion, an open ended concept such as type theory can only benefit from the fact that several formalisms have been proposed. We do not intend the definitions proposed in this thesis to be definitive or prescriptive. Finitary type theories are well-suited for our needs, and we hope that other practitioners of type theory too can find them useful.

5.1.2 Andromeda metalanguage

The landscape of proof assistants based on type theory is vast (Constable et al. 1986; The Coq development team 2021a; Norell 2009; de Moura et al. 2015; The Isabelle development team 2016), and all of these systems include a form of vernacular in which proofs can be developed rather than forcing the user to write plain terms in type theory. AML has three distinguishing characteristics along which we can compare it to other languages: it features user-definable effects, it supports user-definable type theories, and it employs bidirectional evaluation.

Effectful metalanguages, direct style The original metalanguage of Edinburgh LCF (Gordon, Milner, Morris et al. 1978), namesake of the ML family of programming

languages, featured several effects natively, notably state and exceptions ("escape and escape trapping", (Gordon, Milner and Wadsworth 1979)). ML languages have since been extended with further effects, such as support for concurrency (Reppy 1991) or first-class continuations (Harper, Duba et al. 1993). Likewise, most modern proof assistants come with such a fixed set of effects built into their metalanguage. The Coq proof assistant (The Coq development team 2021a) implements a variant of the calculus of inductive constructions (The Coq development team 2021c). It is written in OCaml (Leroy et al. 2021), and can be extended via plug-ins. OCaml is itself a descendant of ML, and has built-in support for a number of effects which can be used when programming plug-ins. Modifying a prover's implementation language however constitutes a high hurdle, and is not expected of regular users of the system, as can be seen from the fact that the manual does not explain how to do so. Recent work on OCaml aims to bring user-definable algebraic effects and handlers to the language (K. Sivaramakrishnan et al. 2021), which could in principle be used in plugins. We are not aware of any work in this direction. Ultimately, OCaml remains a low-level language from the point of view of proof development and offers the user little support. More commonly, users interact with Coq through one of its tactic languages, which in turn rely on the core tactic engine (Spiwack 2010). The tactic engine provides access to the current proof state, the *proof view*, during the refinement of a partial term, and supports backtracking for proof search. $\mathcal{L}_{tac}$ (Delahaye 2000) is Coq's de facto standard tactic language. It comes with built-in support for backtracking, and allows for easier proof development than working in bare OCaml. But $\mathcal{L}_{tac}$ fares rather poorly in terms of language design compared to members of the ML family, due to its lack of a formal semantics or static type system. While the presentation of AML in Chapter 4 does not include the discussion of a type system, the Andromeda 2 implementation does feature a static Hindley-Milner style type system ruling out certain programming errors that sometimes plague $\mathcal{L}_{tac}$ scripts. The available effects in $\mathcal{L}_{tac}$ are very limited. Stateful tactics for instance have to be programmed in OCaml, and the backtracking behaviour can have undesirable consequences.

Ltac2 (Pédrot 2019) is a member of the ML family that amends many of these shortcomings, and is poised to eventually succeed $\mathcal{L}_{tac}$. It features a carefully chosen selection of effects useful for proof development, such as manipulating Coq's proof state, IO, mutable state, backtracking, and exceptions. Ltac2 is a closer relative to AML, but the selection of effects remains predetermined. But it is built on a firmer foundation than its predecessor, and an extension to runners could be possible, but their interaction with the built-in proof view would have to be carefully considered.

Effectful metalanguages, monadically Yet a different approach is taken by Mtac2 (Kaiser et al. 2018), which instead provides a reflection of Coq's type theory into a monad for meta-programming. The main benefit that Mtac2 provides is thus an expressive type system that can be used to ensure the correctness of Mtac2 tactics. The monad itself is rather limited from the point of view of effectful programming though, allowing only unbounded recursion and exceptions, while manipulation of the proof

view is limited. In principle other effects could be presented monadically in Mtac2, but the difficulty of combining monadic effects was one of the original motivations for the adoption of algebraic effects in their stead (Bauer and Pretnar 2015).

Similar mechanisms for reflection-based monadic meta-programming exist in Agda (van der Walt and Swierstra 2013), Idris (Christiansen and Brady 2016), and Lean (Ebner et al. 2017). The Lean implementation of reflection exposes full access to the proof state, and comes with a library of further predefined standard monads. Approaches based on monadic reflection can be both powerful and efficiently implemented, but suffer from the aforementioned lack of modularity regarding the combination of effects. It is also worth noting that this approach presupposes that the type theory which the reflection targets is a essentially a dependently typed programming language with a well-behaved notion of computation, which need not apply to finitary type theories in general.

User-definable theories We can further compare provers with an eye to the second distinguishing feature of AML, its support for user-definable type theories. There are roughly two categories we can draw. On the one hand, there are provers based on a specific type theory that allow certain extensions. The extension of judgemental equality with rewriting rules has been implemented in Agda and proposed for Coq (Cockx, Tabareau et al. 2020). This mechanism allows the addition of new computation rules and can make it practical to work with axiomatic extensions of type theory that would otherwise interact badly with equality checking. On the other hand, the method is not suited to handle for instance equality reflection or extensionality principles, which can be added to AML by providing suitable runners (Bauer, Gilbert et al. 2018; Bauer, Haselwarter and Petković 2020; Bauer and Petković Komel 2021). Furthermore, the method can only extend an existing calculus, defining a different theory, say Martin-Löf type theory without an impredicative universe in Coq, is not possible.

On the other hand, we can consider fully generic provers that allow the user to define their own theory. As we mentioned before, logical frameworks were introduced with the purpose of representing deductive systems. Existing tools based on LF (Pfenning and Schürmann 1999; Pientka and Dunfield 2010) do not, to the best of our knowledge, allow the user to postulate new judgemental equalities. Instead, the equality judgement of the object theory is represented as a type, following the "judgements as types" idea. This ensures that terms in LF correspond to derivations in the object theory, and permits the metatheoretic analysis of the object theory by conducting proofs by induction over the derivations. The Dedukti proof checker (Assaf et al. n.d.) constitutes an exception to this rule. Based on λΠ-modulo (Cousineau and Dowek 2007), it allows to postulate certain equations in the form of rewriting rules. Unlike Twelf or Beluga its purpose is not the study of metatheory but rather the independent re-checking of proofs exported from another prover to an embedding of its theory into λΠ-modulo. The same remarks about the limitations regarding rewriting in Agda or Coq apply. In contrast, the goal of AML is the construction of judgements in the

object theory rather than its metatheoretic study or their mere re-checking. AML may, however, constitute a suitable metalanguage for equational logical frameworks, and we included a first step into this direction in Appendix B.

Bidirectionalism Bidirectional evaluation is, to the best of our knowledge, a novel contribution of this thesis. A related idea is bidirectional elaboration, pioneered by Saïbi for implementation in Coq (Saïbi 1997) and further developed for Matita (Asperti et al. 2012), which turns incomplete input terms, possibly containing unsolved metavariables, into fully annotated terms. The algorithm in (Asperti et al. 2012) tackles bidirectional elaboration for the full calculus of (co-) inductive constructions. As the algorithm is tailored carefully to the type theory at hand, it can perform a more complete reconstruction than a naïve presentation of the same theory in AML could. In comparison, AML makes no assumptions about the type theory, but allows arbitrary computations as the input language to bidirectional evaluation. This enables the construction of bidirectional computations as derived forms as we have seen in Section 4.4.3.

Andromeda 1 Finally we should mention our own work related to AML. The first incarnation of Andromeda (Bauer, Gilbert et al. 2018) was based on a similar ML-like language. Both the type theory and the metalanguage have undergone some changes in the second version. Andromeda 1 was based on a fixed extensional type theory with the inconsistent but (in terms of universe management) convenient rule $\mathsf{Type} : \mathsf{Type}$. In (Bauer, Gilbert et al. 2018), the convenience of extensional type theory as a logical framework is very visible. Rather than replacing the rule with a particular fixed system of universes, we were lead to develop finitary type theories. The representation of variables in the Andromeda 1 variant of extensional type theory is halfway between that of finitary and context-free type theories. It features (untyped) assumption sets, recovering strengthening, but still represents judgements with explicit contexts. In order to enable forward reasoning, contexts were stored as acyclic graphs rather than lists. Contexts were endowed with a join operation, combining contexts with compatible dependency structures. The context-free presentation is slicker in this respect, as it never requires explicit manipulation of contexts, which are left implicit in the dependency order induced by the assumption sets in a judgement. The Andromeda metalanguage has retained many of its characteristics such as the implementation of inversion principles through pattern matching and the automatic dispatch of coercion operations in the Chk-Syn rule. Andromeda 1 featured arbitrary handlers instead of runners. As discussed in Section 4.3.1, we found runners to be more suitable for our purposes, and the examples of (Bauer, Gilbert et al. 2018) remain expressible in the new setting. In a line of work we initiated jointly with Bauer and Petković and brought to completion by Bauer and Petković, the equality checking algorithm from Andromeda 1 has undergone a substantial generalisation to finitary type theories (Bauer, Haselwarter and Petković 2020; Bauer and Petković Komel

2021). Its extensible design allows familiar type theories to be used without overhead in Andromeda 2.

5.2 Future work

Finitary and context-free type theories The recent emergence of several general definitions of type theories offers an opportunity for further study. Many definitions of open-ended mathematical concepts such as the concept of a space can coexist, and contemporary research in mathematics continues to propose new perspectives (Anel and Catren 2021; Scholze 2019). Mathematics teaches us that we should embrace this pluralism. New definitions are usually introduced with a purpose in mind, and using them within that context is a natural direction for further work. Finitary type theories are used in ongoing research regarding the connection of raw and standard type theories (Petković Komel 2021). To avoid that pluralism leads to fragmentation, we also have to take the step of clarifying the connection of a new definition to other definitions of the same concept. The first step for finitary type theories should be a crisp theorem relating them to general dependent type theories (Bauer, Haselwarter and Lumsdaine 2020), their closest kin. As mentioned in Section 5.1.1, the logical framework of Uemura (Uemura 2019) is more liberal in the class of theories that can express than standard finitary type theories, but more restrictive in what it accepts compared to raw type theories. It would be useful to prove a general adequacy theorem of Uemura's or Harper's (Harper 2021) logical framework for finitary type theories. Conversely, the extension of finitary and context-free type theories to other judgement forms in the style of Uemura's LF seems within reach and would allow the expression of new type theories such as those based on cubical sets (Cohen et al. 2016; Angiuli et al. 2018; Cavallo et al. 2020). Another active domain of current research are modal type theories (Schreiber and Michael Shulman 2014; Birkedal et al. 2021). Multimodal type theory does not readily fit into our setup or the framework of Uemura (Gratzer 2021), and the development of modal finitary type theories is an exciting possibility for further work.

Andromeda metalanguage Programming languages, much like type theories, are a multifaceted concept. With the definition of AML in hand, we can pursue its theoretical study, analyse its relation to other languages, and of course consider further extensions. The obvious first step that should be taken is the proof of the soundness and completeness theorems for AML with respect to standard context-free type theories. As mentioned in Section 4.5, we expect that standard proof techniques for languages with algebraic effects should apply (Lukšič 2020). The mathematical theory of algebraic effects and handlers, and operations and runners, importantly includes equations that operations should satisfy (Plotkin and Pretnar 2013). Until recently (Lukšič 2020), programming languages for effects have mostly ignored equations, despite the well-known potential of optimisations based on effect theories (Kammar 2014). The equations respected by an effect theory on a set of operations depends of course

on the intended interpretation of the generators. We can consider, for instance, the theory obtained by interpreting the $\mathtt{coerce}(\mathcal{J}, \mathcal{B})$ operation as conversion to recover the traditional bidirectional conversion rule from Chk-Syn. Under this assumption, we would expect the equations

$$\begin{aligned}
\mathtt{coerce}(s : A,\ \square : A) &= \mathtt{return}(s : A) \\
\mathtt{coerce}(s : A,\ \square : B) &= \mathtt{coerce}(s : B,\ \square : A) \\
\mathtt{coerce}(\mathtt{coerce}(s : A,\ \square : B),\ \square : C) &= \mathtt{coerce}(s : A,\ \square : C)
\end{aligned}$$

to hold for all type theories by reflexivity, symmetry, and transitivity of judgemental equality.

Translating AML to an exist language with support for effects and handlers (Kiselyov and K. C. Sivaramakrishnan 2018) could open up a path to an efficient execution of AML programs. The current implementation of AML in Andromeda 2 interprets the code, and while this is suitable for experiments we do not expect it to scale to serious proof development. For example, multicore OCaml (K. Sivaramakrishnan et al. 2021) and web assembly with delimited continuation (Pinckney et al. 2020) are sufficiently expressive to implement runners.

The restriction that a runner must use its continuation once, in tail position, has the advantage that runners can be used at toplevel, while general handlers cannot. But it also rules out one of the motivating examples for handlers, namely implementing backtracking. We plan to investigate the addition of local, i.e. non top-level, effect handlers to AML. It remains to be seen if general handlers interact as nicely with bidirectional evaluation as runners do.

The possible extensions of finitary and context free type theories discussed in Section 5.2 would be natural candidates for further extensions of AML. But until the theoretical ground work is laid, the combination of an implementation of modal type theory (Gratzer et al. 2019) with AML remains in the realm of speculation. Beyond the borders of Andromeda, we believe that AML could beneficially be used in other proof assistants. We mentioned the development of Ltac2 in Section 5.1.2 in the context of Coq. The AML definition is modular in terms of its expectations towards the object type theory, and one could imagine the combination of AML with other nuclei implementing different type theories.

Appendix A

AML implementation of the boundary conversion lemma

This section contains the AML implementation of Lemma 3.2.17. As the proof of Lemma 3.2.17 relies on transitivity of judgemental equality, we include the definition of rules corresponding to CF-EqTy-Trans and CF-EqTm-Trans.

```
let eqty-trans-bdry =
    derive (A : return (□ type))
    derive (B : return (□ type))
    derive (C : return (□ type))
    derive (eqAB : return (A ≡ B by □))
    derive (eqBC : return (B ≡ C by □))
    A ≡ C by □

rule EqTy-Trans = eqty-trans-bdry

let eqtm-trans-bdry =
    derive (A : return (□ type))
    derive (t₁ : (□ : (return A)))
    derive (t₂ : (□ : (return A)))
    derive (t₃ : (□ : (return A)))
    derive (eq₁₂ : return (t₁ ≡ t₂ : A by □))
    derive (eq₂₃ : return (t₂ ≡ t₃ : A by □))
    t₁ ≡ t₃ : A by □

rule EqTm-Trans = eqtm-trans-bdry

let eqtm-trans eq₁ eq₂ =
    let b₁ = ∂ eq₁ in
    let b₂ = ∂ eq₂ in
    match (b₁, b₂) with
    | (?t₁ ≡ ?t₂ : ?A by □), (_ ≡ ?t₃ : _ by □) →
        EqTm-Trans (return A) (return t₁) (return t₂) (return t₃)
                   (return eq₁) (return eq₂)
```

```
let rec boundary-convert j bdry₂ =
    let bdry₁ = ∂ j in
    let b = bdry₁ =α= bdry₂ in
    match b with true → j | false →
    (match (bdry₁, bdry₂) with
    | □ : ?A₁, □ ?A₂ →
        let eq = tt-refl A₁ A₂ in
        convert j eq
    | (? A₁ ≡ ?B₁ by □), (? A₂ ≡ ?B₂ by □) →
        let eqA = tt-refl A₂ A₁ in
        let eqB = tt-refl B₁ B₂ in
        let eq₁ = EqTy-Trans (return A₂) (return A₁) (return B₁)
                             (return eqA) (return j) in
        let eq₂ = EqTy-Trans (return A₂) (return B₁) (return B₂)
                             (return eq₁) (return eqB) in
        eq₂
    | (? s₁ ≡ ?t₁ : ?A₁ by □), (? s₂ ≡ ?t₂ : ?A₂ by □) →
        let eqA = tt-refl A₁ A₂ in
        let j' = tt-convert j eqA in
        (match ∂ j' with
         | s₁' ≡ t₁' : A₂ \by □ →
             let eq-s = tt-refl A₂ s₂ s₁' in
             let eq-t = tt-refl A₂ t₁' t₂ in
             let eq₁ = eqtm-trans eq-s j in
             let eq₂ = eqtm-trans eq₁ eq-t in
             eq₂)
    | ({_ : ?A₁} _, {_ : ?A₂} _) →
        let a₂ = tt-var A₂ in
        let eqA = tt-refl A₂ A₁ in
        let a₁ = tt-convert a₂ eqA in
        let j' = tt-subst j a₁ in
        let bdry₂' = tt-subst bdry₂ a₂ in
        let j" = boundary-convert j' bdry₂' in
        tt-abstr a₂ j")

let boundary-converter = runner coerce j bdry_opt →
    match bdry_opt with
    | Unbounded → coerce j
    | Boundary ?bdry₂ →
      let bdry₁ = ∂ j in
      let b = bdry₁ =ε= bdry₂ in
      match b with
      | false → coerce j as bdry₂
      | true → boundary-convert j bdry₂

with boundary-converter end
```

Appendix B

Equational LF in Andromeda 2

The code presented in this appendix attempts to implement Harper's *Equational Logical Framework* (Harper 2021) in Andromeda 2. The rules in (Harper 2021) do not directly constitute a standard type theory in the sense of Definition 3.1.14 for three reasons. In (Harper 2021), several object rules omit premises introducing metavariables, and the symbols that each rule introduces do not record all metavariables. For instance, the LAM rule does not include a premise for K_2. The resulting system is nonetheless presuppositive (Harper 2021, Lemma 1). As Andromeda 2 only accepts standard type theories, our presentation deviates slightly from (Harper 2021) and add the omitted premises. As a result, the elided metavariables are automatically added as arguments to the symbols. In (Harper 2021), there are two introduction rules for dependent products, PI-CLS and PI-SORT, both introducing the same symbol. As a standard type theory is allowed to introduce a symbol only through one rule, we instead create two kinds of dependent product, one corresponding to each rule, and add a coherence equation PI_COH governing their interaction.

The surface language of Andromeda 2 does not match exactly the syntax presented in Section 4.2, but the differences should be sufficiently minor not to impede readability. Appendix B.1 defines the rules given in (Harper 2021). Appendix B.2 contains most of the examples of (Harper 2021). Especially in the latter section, we frequently use the notation untyped abstractions and substitution that was introduced defined in Section 4.2.

B.1 Equational LF rules

```
(*
   ______________________________________________________________
                  Andromeda formalisation of
     "An Equational Logical Framework for Type Theories"
                         Robert Harper
                         June 4, 2021
                     (arxiv:2106.01484v1)
   ______________________________________________________________

   Standard CFTT presentation & Andromeda formalisation by
   Philipp G. Haselwarter.
 *)



(* We load the eqchk library and libraries postulating
   transitivity and symmetry of type and term equality, and define
   some auxiliary functions. *)
require eq
require judgemental_equality_type
require judgemental_equality_term

let sym_eq h = match h with
  | ?A ≡ ?B → judgemental_equality_type.eq_type_sym A B h
  | ?a ≡ ?b : ?A → judgemental_equality_term.eq_term_sym A a b h
  end
let rec refl_eq j = match j with
  | _ type → judgemental_equality_type.eq_type_refl j
  | _ : ?A → judgemental_equality_term.eq_term_refl A j
  | ({_ : ?A} _ :> judgement) → {y : A} (refl_eq j{y} :> judgement)
  end


(* ____________________________________________
   Figure 4: Formation Judgements (begin)
   ____________________________________________ *)

(* Interpreting "cls" as "type". *)
rule Sort type
rule incl (S : Sort) type

with operation ML.coerce (?j : Sort) (□ type) → incl j end

rule Pi_cls (S1 : Sort) ({X : S1} K2 type) type

rule Eq_cls (S : Sort) (O1 : S) (O2 : S) type
```

```
(* In [Ha21] the same notation is used for dependent products with
   codomain in cls or in sort. In a standard CFTT, each symbol can
   only be introduced by a single rule. We therefore introduce a
   separate dependent products for sorts. *)
rule Pi_sort (S1 : Sort) ({ X : S1} S2 : Sort) : Sort

(* In order to use application, we need to be able to convert a
   term at a sort-valued product to a class-valued product. *)
rule Pi_coh (S1 : Sort) ({ X : S1} S2 : Sort)
  : Pi_sort S1 S2
    ≡ Pi_cls S1 ({ X : S1} S2{X})

(* Install the Pi_coh rule as equality hint. *)
let _ = eq.add_rule Pi_coh

(* To give a standard presentation of equational LF, we have to
   modify the lam rule to include a premise specifying the type of
   K2. Besides requiring the presence of the premise, CFTT also
   records the argument when the symbol is applied. *)
rule lam (S1 : Sort) ({ X : S1} K2 type)
  ({ X : S1} O2 : K2{X})
  : Pi_cls S1 K2

(* Again, we add missing premises S1, K2. *)
rule app (S1 : Sort) ({ X : S1} K2 type)
         (O : Pi_cls S1 K2)
         (O1 : S1)
         : K2{O1}

(* application for f : Pi_sort A B, using Pi_coh *)
let app_sort = derive (A : Sort) ({ X : incl A} B : Sort)
     (f : incl (Pi_sort A B)) (a : incl A)
     → (* convert f along Pi_coh, then use app *)
     ( let f' = let h = Pi_coh A B in convert f h in
       app A ({ X : incl A} incl B{X}) f' a )

(* We add a missing premise introducing S.
   We will now stop mentioning such missing premises. *)
rule self (S : Sort) (O : S) : (Eq_cls S O O)

(* ────────────────────────────────────────
   Figure 4: Formation Judgements (end)
   ──────────────────────────────────────── *)
```

```
(* ───────────────────────────────────────────
   Figure 5: Structural Judgements (begin)
   ─────────────────────────────────────────── *)

(* Rules for context formation and variable projection omitted. *)

rule cls_rfl (K type) : K ≡ K
rule cls_st (K type) (K' type) (K'' type)
  (K ≡ K') (K'' ≡ K')
  : K ≡ K''
rule obj_rfl (K type) (O : K) : O ≡ O : K
rule obj_st (K type) (O : K) (O' : K) (O'' : K)
  (O ≡ O' : K) (O'' ≡ O' : K)
  : O ≡ O'' : K

(* Term conversion is implemented as a derived rule by appealing
   to Andromeda's built-in eliminator for judgemental equality,
   `convert`. *)
let obj_cls = derive (K type) (K' type)
  (O : K) (K ≡ K' by eq)
  → (convert O eq)

(* Term equality conversion, similarly using `convert` *)
let obj_eq_cls = derive (K type) (K' type)
  (O : K) (O' : K)
  (O ≡ O' : K by eq_O) (K ≡ K' by eq_K)
  → (convert eq_O eq_K)

(* ─────────────────────────────────────────
   Figure 5: Structural Judgements (end)
   ───────────────────────────────────────── *)


(* ─────────────────────────────────────────
   Figure 6: Equality Judgements (begin)
   ───────────────────────────────────────── *)

(* Andromeda provides congruence rules for symbols via the
   `congruence` keyword. For the sake of completeness, we present
   the congruence rules in [Ha21] as derived rules, but we could
   well omit them and use `congruence` when needed. *)
let incl_eq = derive (S : Sort) (S' : Sort)
  (S ≡ S' : Sort by α)
  →
  congruence (incl S) (incl S') α
```

```
(* install incl_eq as toplevel runner *)
with operation ML.coerce
  (( _ ≡ _ : Sort) as ?j) (( incl ?S ≡ incl ?S' by □) as ?bdry)
  → incl_eq S S' j
end


let pi_class_eq = derive (S1 : Sort) (S1' : Sort)
  ({ X : S1} K2 type) ({ X : S1' } K2' type)
  (S1 ≡ S1' : Sort by α)
  ({ X : S1} K2{X} ≡ K2' {convert X (incl_eq S1 S1' α)} by β)
  →
  congruence (Pi_cls S1 K2) (Pi_cls S1' K2' ) α β


let eq_class_eq = derive (S : Sort) (S' : Sort)
  (O1 : S) (O1' : S) (O2 : S' ) (O2' : S' )
  (S ≡ S' : Sort by α) (O1 ≡ O1' : S by β) (O2 ≡ O2' : S' by γ)
  →
  (eq.add_locally (derive → α) (fun () →
  let h = eq.prove (incl S' ≡ incl S by □) in
  let γ' = convert γ h in
  congruence (Eq_cls S O1 O2) (Eq_cls S' O1' O2' ) α β γ' ))


let pi_sort_eq = derive (S1 : Sort) (S1' : Sort)
  ({ X : S1} S2 : Sort) ({ X : S1' } S2' : Sort)
  (S1 ≡ S1' : Sort by α)
  ({ X : S1} S2{X} ≡ ( eq.add_locally (derive → α) (fun () →
                       S2' {X})) : Sort by β)
  →
  congruence (Pi_sort S1 S2) (Pi_sort S1' S2' ) α β


(* This rule seems needed for lam_eq because of the phrasing of
   the first equation " Γ ⊢ S₁ = S₁' cls " . It would seem more
   natural to require the equation " Γ ⊢ S₁ = S₁' : Sort "
   instead. *)

rule El_incl_injectivity (S1 : Sort) (S1' : Sort)
  (incl S1 ≡ incl S1' by α)
  : S1 ≡ S1' : Sort

let lam_eq =
  derive (S1 : Sort) (S1' : Sort)
    ({ X : S1} K2 type) ({ X : S1' } K2' type)
    ({ X : S1} O2 : K2{X}) ({ X : S1' } O2' : K2' {X})
```

```
    (incl S1 ≡ incl S1'  by α)
    ({X : S1}  K2{X}  ≡  K2'{convert X α}  by β)
    ({X : S1}  O2{X}  ≡  ( eq.add_locally (derive → α) (fun () →
                             convert O2' {X} (sym_eq β{X}))  )
                        : K2{X} by γ)
    →
    congruence (lam S1 K2 O2) (lam S1' K2' O2')
               (El_incl_injectivity S1 S1' α) β γ

(* Congruence for application. Note that no equations for S₁ and
   K₂ are given. No generality is lost, because O₁' can be
   converted to ΠS₁K₂ via pi-sort-eq if necessary, and similarly
   for O'. *)
let app_eq = derive (S1 : Sort) ({X : S1} K2 type)
    (O : Pi_cls S1 K2) (O' : Pi_cls S1 K2)
    (O1 : S1) (O1' : S1)
    (O ≡ O' : Pi_cls S1 K2 by α)  (O1 ≡ O1' : S1 by β)
    →
    congruence (app S1 K2 O O1) (app S1 K2 O' O1')
               (refl_eq S1) (refl_eq K2) α β

rule app_lam (S1 : Sort) ({X : S1} K2 type)
     ({X : S1} O2 : K2{X})  (O1 : S1)
     : app S1 K2 (lam S1 K2 O2) O1 ≡ O2{O1} : K2{O1}

rule lam_app (S1 : Sort) ({X : S1} K2 type)
     (O : Pi_cls S1 K2)
     : O ≡ lam S1 K2 ({X : S1} app S1 K2 O X) : Pi_cls S1 K2

rule reflection (S : Sort) (O1 : S) (O2 : S)
  (O : Eq_cls S O1 O2)
  : O1 ≡ O2 : S

rule unicity (S : Sort) (O1 : S) (O2 : S)
  (O : Eq_cls S O1 O2) (O' : Eq_cls S O1 O2)
  : O ≡ O' : Eq_cls S O1 O2

(* ____________________________________
   Figure 6: Equality Judgements (end)
   ____________________________________ *)



(* Full congruence for application, with the possibility to vary
   S₁ and K₂. As claimed, we can derive it from app_eq using
   pi_class_eq to convert O' and obj_cls to convert O1'. *)
let app_eq' = derive
    (S1 : Sort) (S1' : Sort)
```

```
({X : S1} K2 type)  ({X : S1'} K2' type)
(O : Pi_cls S1 K2)  (O' : Pi_cls S1' K2')
(O1 : S1)  (O1' : S1')

(S1 ≡ S1' : Sort by α)
({X : S1} K2{X} ≡ (K2'{convert X (incl_eq S1 S1' α)}) by β)

(O ≡
   ( let h_π = pi_class_eq S1 S1' K2 K2' α β in
     convert O' (sym_eq h_π) )
   : Pi_cls S1 K2 by γ)
(O1 ≡
    (eq.add_locally (derive → (sym_eq α)) (fun () → O1' : S1))
    : S1 by δ)
→
let O'' = ( let h_π = pi_class_eq S1 S1' K2 K2' α β in
            convert O' (sym_eq h_π) ) in
let O1'' = convert O1' (sym_eq (incl_eq S1 S1' α)) in
app_eq S1 K2 O O'' O1 O1'' γ δ
```

B.2 Equational LF examples

```
(* Large and small dependent LF product. *)
let Π = Pi_cls
let π = Pi_sort

(* Infix notations for non-dependent functions. A ML-typing
   annotation disambiguates (b :> judgement) from a boundary. *)
let ( »--> ) a b = Π a ({_ : incl a} (b :> judgement))
let ( »-> ) a b = π a ({_ : incl a} (b :> judgement))

(* Infix notation for application. *)
let ( ` ) f a = match f with
  | _ : (Pi_cls ?S1 ?K2) → app S1 K2 f a
  | _ : incl (Pi_sort ?S1 ?S2) → app_sort S1 S2 f a
  end

(* The `decoder` function allows us to omit applications of `el`
   in the definitions of Gödel's T and the dependent version of T.
   As the runner it defines (misleadingly called "handler" for
   historical reasons) is parametrised by `tp` and `el`, we can
   use `decoder` in both definitions. *)
let decoder tp el =

    (* check that decoding from tp via el is requested *)
    let guard j tp' bdry c_true = match base.( = ) tp' tp with
      | ML.true → c_true ()
      | ML.false → ML.coerce j bdry
      end in

    handler
    | ML.coerce (?j : (incl ?tp')) ((□ : Sort) as ?bdry) →
      guard j tp' bdry (fun () → el ` j)
    | ML.coerce (?j : (incl ?tp')) ((□ type) as ?bdry) →
      guard j tp' bdry (fun () → incl (el ` j))
    end


let Gödel's_T =
  {tp : Sort}
  {el : tp »--> Sort}
  (with decoder tp el try       (* handle decoding with `el` *)
  {nat : tp}
  {arr : tp »-> tp »-> tp}
  {zero : nat}
  {succ : nat »-> nat}
```

```
  {nat_rec : π tp ({A} (A »-> (nat »-> A »-> A) »-> nat »-> A))}

  {nat_β_z : Π tp ({A} (Π A ({b} (Π (nat »-> A »-> A) ({s}
                (Eq_cls A (nat_rec `A `b `s `zero) b ))))))}

  {nat_β_s : Π tp ({A} (Π A ({b} (Π (nat »-> A »-> A) ({s} (Π nat ({n}
                (let lhs = nat_rec `A `b `s `(succ `n) in
                 let rest = nat_rec `A `b `s `n in
                 Eq_cls A lhs (s `n `rest) ))))))))}

  {lambda : π tp ({A1} π tp ({A2} (A1 »-> A2) »-> (arr `A1 `A2)))}

  {apply : π tp ({A1} π tp ({A2} (arr `A1 `A2) »-> A1 »-> A2))}

  {arr_β : Π tp ({A1} Π tp ({A2} Π (A1 »-> A2) ({F} Π A1 ({M1}
             (let lhs = apply `A1 `A2 `(lambda `A1 `A2 `F) `M1
              in Eq_cls A2 lhs (F `M1))))))}

  {arr_η : Π tp ({A1} Π tp ({A2} Π (arr `A1 `A2) ({M}
             (let body = {x : A1} apply `A1 `A2 `M `x in
              let l = lam A1 ({_ : A1} A2) body in
              let rhs = lambda `A1 `A2 `l in
              Eq_cls (arr `A1 `A2) M rhs))))}

(* ________________________________
   Figure 7: Signature of Gödel's T
   ________________________________ *)

(* The context is set up, returning a dummy value. *)
   Sort)



let Dependent_Gödel's_T_Equality_and_Identity =
  {tp : Sort}
  {el : tp »--> Sort}
  (with decoder tp el try       (* handle decoding with `el` *)
  {nat : tp}
  {arr : tp »-> tp »-> tp}
  {pi : π tp ({A} (A »-> tp) »-> tp)}
  {zero : nat}
  {succ : nat »-> nat}

  {nat_rec : π (nat »-> tp) ({A}
             π (A `zero) ({b}
             π (π nat ({m} ((A `m) »-> (A `(succ `m)))))   ({s}
             π nat ({n} A `n))))}
```

```
{nat_β_z : Π (nat »-> tp) ({ A}
           Π (A `zero) ({ b}
           Π (π nat ({ m} (( A `m) »-> (A `(succ `m))))   ({ s}
           (Eq_cls (A `zero) (nat_rec `A `b `s `zero) b) )))}

{nat_β_s : Π (nat »-> tp) ({ A}
           Π (A `zero) ({ b}
           Π (π nat ({ m} (( A `m) »-> (A `(succ `m))))   ({ s}
           Π nat ({ n}
           (Eq_cls (A `(succ `n))
                   (nat_rec `A `b `s `(succ `n))
                   (s `n `(nat_rec `A `b `s `n)))))))}

{lambda : π tp ({ A1} π (A1 »-> tp) ({ A2} (π A1 ({ x} A2 `x))
           »-> (pi `A1 `A2)))}

{apply : π tp ({ A1} π (A1 »-> tp) ({ A2}
         pi `A1 `A2 »-> π A1 ({ x} A2 `x)))}

{pi_β : Π tp ({ A1} Π (A1 »-> tp) ({ A2}
        Π (π A1 ({ x} A2 `x)) ({ F} Π A1 ({ M1}
          (let lhs = apply `A1 `A2 `(lambda `A1 `A2 `F) `M1
           in Eq_cls (A2 `M1) lhs (F `M1))))))}

{pi_η : Π tp ({ A1} Π (A1 »-> tp) ({ A2} Π (pi `A1 `A2) ({ M}
            (let body = {x : A1} apply `A1 `A2 `M `x in
             let l = lam A1 ({ x : A1} A2 `x) body in
             let rhs = lambda `A1 `A2 `l in
             Eq_cls (pi `A1 `A2) M rhs))))}

(* ____________________________________________
   Figure 8: Signature of Dependent Gödel's T
   ____________________________________________ *)



{eq : π tp ({ A} A »-> A »-> tp)}
{self : π tp ({ A} π A ({ M} (eq `A `M `M)))}
{eqref : Π tp ({ A} Π A ({ M1} Π A ({ M2}
         eq `A `M1 `M2 »--> Eq_cls A M1 M2)))}
{equni : Π tp ({ A} Π A ({ M1} Π A ({ M2} (let eqM = (eq `A `M1 `M2) in
         Π eqM ({ M} Π eqM ({ M' }
           (Eq_cls eqM M M' )))))))}

(* ___________________________________
   Figure 9: Dependent Equality Type
   ___________________________________ *)
```

```
  {id : π tp ({A} A »-> A »-> tp)}
  {refl : π tp ({A} π A ({M} (id `A `M `M)))}
  {j : π tp ({A}
       π (π A ({m1} π A ({m2} (id `A `m1 `m2) »-> tp))) ({B}
       π (π A ({x} B `x `x `(refl `A `x))) ({r}
       π A ({m} π A ({m' }
       π (id `A `m `m' ) ({p}
       B `m `m' `p )))))))}
  {id_β : Π tp ({A}
       Π (π A ({m1} π A ({m2} (id `A `m1 `m2) »-> tp))) ({B}
       Π (π A ({x} B `x `x `(refl `A `x))) ({r}
       Π A ({m}
           (let ty = (B `m `m `(refl `A `m)) in
            let lhs = (j `A `B `r `m `m `(refl `A `m)) in
            let rhs = (r `m) in
            Eq_cls ty lhs rhs)))))}

(* ____________________________________
   Figure 10: Dependent Identity Type
   ____________________________________ *)


(* The context is set up, returning a dummy value. *)
   Sort)
```

Appendix C

Razširjeni povzetek v slovenščini

This appendix contains a summary of the thesis in Slovene.

C.1 Poglavje 1: Uvod

V disertaciji razvijemo splošno matematično teorijo za teorije odvisnih tipov in v učinkovni metateoriji opišemo, kako z njimi računamo. V literaturi se pojavljajo študije mnogih primerov teorij odvisnih tipov, kot na primer Martin-Löfova teorija tipov (Martin-Löf 1998) ali račun konstrukcij (Coquand in Huet 1988). Vendar pa ni splošno sprejete definicije, ki bi zajela širok nabor primerov in omogočila razvoj splošne metateorije. S tem namenom predlagamo definicijo v obliki končnih teorij tipov in dokažemo osnovne metateoretične rezultate. Nato reformuliramo končne teorije tipov, da omogočajo praktičen razvoj dokazov, s čimer pridemo do kontekstno neodvisnih teorij tipov. Te nam omogočajo razvoj splošnega metajezika za standardne končne teorije tipov.

Radi bi preučevali širok razred teorij tipov, tudi tiste, ki ne omogočajo odločljivega preverjanja tipov. S tem namenom bomo posvojili Martin-Löfov stil prezentacije teorij tipov. Martin-Löf predstavi teorije tipov preko štirih sodb:

Pod predpostavkami Γ sodba

- $\Gamma \vdash A\ \mathsf{type}$ trdi, da je A dobro oblikovan tip,
- $\Gamma \vdash A \equiv B$ trdi, da sta tipa A in B enaka,
- $\Gamma \vdash s : A$ trdi, da je s term tipa A,
- $\Gamma \vdash s \equiv t : A$ trdi, da sta terma s in t enaka pri tipu A.

Teorija tipov je podana z množico pravil, ki jih induktivno beremo kot definicijo deduktivnega sistema za izpeljevanje sodb.

Pojasnilo. Zaradi prostorskih omejitev v ta povzetek ne moremo vključiti polnih definicij. Pogosto bomo opustili dele definicij in izrekov, ter vključili le informacije, ki so potrebne za predstavitev splošne ideje.

C.2 Poglavje 2: Končne teorije tipov

V poglavju 2 predlagamo *končno teorijo tipov* kot elementarno definicijo za širok razred teorij tipov v stilu Martina-Löfa. Teorija tipov mora imeti nekatere metateoretične lastnosti: sestavni deli izpeljive sodbe morajo biti dobro oblikovani, dopustiti mora pravila za substitucijo in vsak term mora imeti enoličen tip.

Definicija teorije tipov je zgrajena po stopnjah. Vsaka stopnja izboljša pojem *pravila* in *teorije tipov*, saj določi pogoje za dobro oblikovanje. Začnemo s surovo sintakso (§2.1.1) izrazov in formalnih metaspremenjivk, iz katerih zgradimo kontekste, substitucije in sodbe.

Izraz za tip ali kar preprosto ***tip*** dobimo z uporabo simbola za tip na argumentih $\mathsf{S}(e_1, \ldots, e_n)$ ali z uporabo metaspremenljivke za tip na izrazih za term $\mathsf{M}(t_1, \ldots, t_n)$. ***Izraz za term*** ali kar preprosto ***term*** je lahko prosta spremenljivka a, vezana spremenljivka x, uporaba simbola za term na argumentih $\mathsf{S}(e_1, \ldots, e_n)$, ali uporaba metaspremenljivke za term na izrazih za term $\mathsf{M}(t_1, \ldots, t_n)$.

Oblika za ***(hipotetično) sodbo*** je

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

Nato vpeljemo *surova pravila* (§2.1.3), formalen pojem, ki mu pogosto rečemo "shematično pravilo sklepanja".

Definicija C.2.1. ***Surovo pravilo*** nad signaturo Σ je hipotetična sodba nad Σ oblike $\Theta; [\,] \vdash j$. Takšno pravilo označimo z

$$\Theta \Longrightarrow j.$$

Elementi Θ so ***premise*** in j je ***zaključek***. Pravimo, da je pravilo ***objektno pravilo***, kadar je j sodba za tip ali term, in ***pravilo enakosti***, kadar je j sodba enakosti.

Primer C.2.2. Surova pravila lahko prevedemo nazaj v njihovo tradicionalno obliko tako, da zapolnimo glave z generično uporabljenimi metaspremenjivkami. Na primer, bralec lahko preveri, da je surovo pravilo

$$\mathsf{A}{:}(\Box\ \mathsf{type}),\ \mathsf{s}{:}(\Box : \mathsf{A}),\ \mathsf{t}{:}(\Box : \mathsf{A}),\ \mathsf{p}{:}(\Box : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})) \Longrightarrow \mathsf{s} \equiv \mathsf{t} : \mathsf{A}\ \mathsf{by}\ \star$$

v korespondenci z *refleksijo enakosti*, pravilom ekstenzionalne teorije tipov, ki ga tradicionalno zapišemo kot

$$\textsc{Eq-Reflect}\quad \frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A} \qquad \vdash \mathsf{p} : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A}}$$

Za lažjo berljivost bomo surova pravila prikazovali v tradicionalni obliki, a bomo uporabili definicijo 2.1.5, ko je to zaradi nivoja formalnosti potrebno.

TT-META

$$\dfrac{\Theta(\mathsf{M}_k) = \{x_1{:}A_1\}\cdots\{x_m{:}A_m\}\ \mathcal{b} \qquad \Theta;\Gamma \vdash t_j : A_j[\vec{t}_{(j)}/\vec{x}_{(j)}] \quad \text{for } j = 1,\ldots,m \qquad \Theta;\Gamma \vdash \mathcal{b}[\vec{t}/\vec{x}]}{\Theta;\Gamma \vdash (\mathcal{b}[\vec{t}/\vec{x}])\boxed{\mathsf{M}_k(\vec{t})}}$$

TT-VAR

$$\dfrac{\mathsf{a} \in |\Gamma|}{\Theta;\Gamma \vdash \mathsf{a} : \Gamma(\mathsf{a})}$$

TT-ABSTR

$$\dfrac{\Theta;\Gamma \vdash A\ \mathsf{type} \qquad \mathsf{a} \notin |\Gamma| \qquad \Theta;\Gamma,\mathsf{a}{:}A \vdash \mathcal{J}[\mathsf{a}/x]}{\Theta;\Gamma \vdash \{x{:}A\}\ \mathcal{J}}$$

Slika C.1: Pravila zaprtja za spremenljivke, metaspremenljivke in abstrakcijo.

Definicija C.2.3. Naj bo

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

surovo objektno pravilo za mejo nad Σ. ***Pridruženo simbolno pravilo*** za $\mathsf{S} \notin |\Sigma|$ je surovo pravilo

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

nad razširjeno signaturo $\langle\Sigma, \mathsf{S}{\mapsto}(c, [\mathsf{ar}(\mathcal{B}_1),\ldots,\mathsf{ar}(\mathcal{B}_n)])\rangle$, kjer je $\widehat{\mathsf{M}}$ ***generična uporaba*** metaspremenljivke M s pridruženo mejo $\mathcal{B}$, definirana kot

1. $\widehat{M} = \{x_1\}\cdots\{x_k\}\mathsf{M}(x_1,\ldots,x_k)$, če je $\mathsf{ar}(\mathcal{B}) = (c,k)$ in $c \in \{\mathsf{Ty},\mathsf{Tm}\}$,
2. $\widehat{M} = \{x_1\}\cdots\{x_k\}\star$, če je $\mathsf{ar}(\mathcal{B}) = (c,k)$ in $c \in \{\mathsf{EqTy},\mathsf{EqTm}\}$.

Primer C.2.4. V skladu z definicijo 2.1.10 je simbolno pravilo za Π generirano s pravilom za mejo

$$\dfrac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \{x{:}\mathsf{A}\}\ \mathsf{B}(x)\ \mathsf{type}}{\vdash \Box\ \mathsf{type}}$$

Res, pridruženo simbolno pravilo za Π je

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

Nato uvedemo *strukturna pravila* (slike 2.4, 2.5, 2.6), ki pripadajo vsaki teoriji tipov, in definiramo *pravila kongruence* (definicija 2.1.13).

Ta pravila so potem zbrana v surovih teorijah tipov (definicja 2.1.16).

Definicija C.2.5. ***Surova teorija tipov*** T nad signaturo Σ je družina surovih pravil nad Σ, ki jim pravimo ***specifična pravila*** teorije T. ***Pridružen deduktivni sistem*** teorije T je sestavljen iz:

1. ***strukturnih pravil*** nad Σ:

   a) pravila zaprtja za *spremenljivke*, *metaspremenljivke*, *kongruence metaspremenljivk* in *abstrakcije* (slika 2.4),
   b) pravila zaprtja za *enakost* (slika 2.5),
   c) pravila zaprtja za *meje* (slika 2.6);

2. instanciacij specifičnih pravil teorije T (definicija 2.1.8);

3. za vsako specifično objektno pravilo teorije T imamo instanciacijo pridruženega pravila kongruence (definicija 2.1.13).

Pišemo $\Gamma \vdash_T \mathcal{J}$, ko je $\Gamma \vdash \mathcal{J}$ izpeljiva glede na deduktivni sistem, ki je pridružen teoriji T, in podobno za $\Gamma \vdash_T \mathcal{B}$.

Da bi preprečili cikle v izpeljavah dobre tipiziranosti in podali princip indukcije za končne teorije tipov, uvedemo *končna pravila* in *končne teorije tipov* (§2.1.4).

Definicija C.2.6. Naj bo T teorija nad signaturo Σ. Surovo pravilo $\Theta \Longrightarrow \mathscr{b}\boxed{e}$ nad Σ je ***končno*** nad T, ko velja $\vdash_T \Theta\ \mathsf{mctx}$ in $\Theta; [\,] \vdash_T \mathscr{b}$. Podobno je surovo pravilo za mejo $\Theta \Longrightarrow \mathscr{b}$ končno, ko velja $\vdash_T \Theta\ \mathsf{mctx}$ in $\Theta; [\,] \vdash_T \mathscr{b}$.

Končna teorija tipov je surova teorija tipov $(R_i)_{i\in I}$, za katero obstaja dobro osnovana urejenost $(I, \prec)$, za katero velja, da je vsako pravilo R_i končno nad $(R_j)_{j\prec i}$.

Na koncu so uvedemo *standardne* teorije tipov (definicija 2.1.20), pri katerih je vsakemu simbolu pridruženo enolično pravilo.

Definicija C.2.7. Končna teorija tipov je ***standardna*** če so njena specifična objektna pravila simbolna pravila in ima vsak simbol natanko eno pridruženo pravilo.

Dokažemo metaizreke o surovih (§2.2.1), končnih (§2.2.2), in standardnih teorijah tipov (§2.2.3).

Prispevki. Predlagamo matematično natančno definicijo teorije tipov. Vse konstrukcije v tem poglavju so konstruktivne s ciljem uporabe zgolj elementarnih pojmov, ki jih je mogoče interpretirati v naboru različnih formalizmov. Da povzamemo:

- definiramo pojem mestnosti in signature, ki ustrezata strukturi vezanja spremenljivk, ki jo pogosto najdemo v teorijah tipov.
- definiramo splošen pojem surove sintakse,
- podamo način, kako formalno ravnamo z metaspremenljivkami,
- uvedemo uporabno dekompozicijo sodbe na *glavo* in *mejo*,
- definiramo pravila, ki se ujemajo s pogosto prakso v teoriji tipov,
- razložimo lastnosti, ki naredijo teorijo tipov *končno* in *standardno*,
- dokažemo naslednje metaizreke:
  – dopustnost substitucije in enakosti substitucij (izrek 2.2.8),
  – dopustnost instanciacije metaspremenljivk (izrek 2.2.13) in enakosti instanciacij (izrek 2.2.17),

- izpeljivost predpostavk (trditev 2.2.18),
- dopustnost "ekonomičnih" pravil (trditev 2.2.19, 2.2.20)
- principi inverzije (izrek 2.2.22),
- enoličnost tipiziranja (izrek 2.2.24).

C.3 Poglavje 3: Kontekstno neodvisne teorije tipov

Cilj tega poglavja je razviti prezentacijo končne kontekstno neodvisne teorije tipov, ki lahko služi kot osnova za implementacijo v dokazovalnem pomočniku. Definicija končne teorije tipov v poglavju 2 je primerna za študijo metateoretičnih lastnosti teorije tipov, vendar pa ne naslavlja problemov pri implementaciji. To zlasti velja za kontekste, ki so v tradicionalnih prezentacijah teorij tipov eksplicitno predstavljeni s seznami.

Radi bi zamenjali sodbe oblike "$\Theta; \Gamma \vdash \mathcal{J}$" s kratko obliko "$\mathcal{J}$". V tradicionalnih prezentacijah logike, kot tudi v Γ_∞ (Geuvers in sod. 2010), to dosežemo tako, da proste spremenljivke eksplicitno označimo s tipi: namesto, da bi imeli $\mathsf{a} : A$ v kontekstu spremenljivk, vsako pojavitev a označimo s svojim tipom a^A.

Isto idejo uporabimo tudi pri kontekstno neodvisnih teorijah tipov, vendar pa moramo premagati številne tehnične zaplete. Največji izziv je pomanjkanje krepitve (strengthening), ki pravi, da če je $\Theta; \Gamma, \mathsf{a}{:}A, \Delta \vdash \mathcal{J}$ izpeljiva sodba in se a ne pojavi v Δ in $\mathcal{J}$, potem je $\Theta; \Gamma, \Delta \vdash \mathcal{J}$ tudi izpeljiva sodba. Primer pravila, ki krši krepitev, je refleksija enakosti iz primera 2.1.7,

$$\frac{\vdash \mathsf{A}\ \mathsf{type} \qquad \vdash \mathsf{s} : \mathsf{A} \qquad \vdash \mathsf{t} : \mathsf{A} \qquad \vdash \mathsf{p} : \mathsf{Id}(\mathsf{A}, \mathsf{s}, \mathsf{t})}{\vdash \mathsf{s} \equiv \mathsf{t} : \mathsf{A}}$$

Ker se v zaključku ne pojavi metaspremenljivka p, ne bomo zabeležili dejstva, da smo morda uporabili spremenljivko pri izpeljavi četrte premise. Posledično samo iz sodbe ne moremo ugotoviti, katere spremenljivke se morajo pojaviti v kontekstu. Izkaže se, da so srž problema spremenljivke, ki jih izpustimo v izpeljavah enačb. Situacijo lahko popravimo tako, da spremenimo sodbe enakosti, ki po novem nosijo dodatno informacijo o tem, katere spremenljivke smo uporabili pri izpeljavi.

V *kontekstno neodvisnih teorijah tipov* spremenimo sintakso izrazov (§3.1.1) tako, da je vsaka prosta spremenljivka označena s svojim tipom a^A, namesto da bi ji tip dodelil kontekst. Ker so spremenljivke, ki se pojavljajo v oznaki s tipom A, tudi označene s svojimi tipi, se zabeleži odvisnost med spremenljivkami. Torej sodbe v kontekstno neodvisnih teorijah tipov ne nosijo s seboj eksplicitnega konteksta. Z metaspremenljivkami ravnamo podobno. Upoštevati moramo tudi pravila, ki ne nosijo informacije o dokazu (proof-irrelevant rules). Tak primer je refleksija enakosti, kjer se v zaključku pravila ne pojavljajo vse spremenljivke, ki nastopajo v izpeljavi premis. Zato sodbe enakosti oplemenitimo z *množicami predpostavk* (§3.1.1.5). Za intuicijo si lahko predstavljamo, da v sodbi $\vdash A \equiv B$ by α množica predpostavk α vsebuje (označene) spremenljivke, ki smo jih uporabili pri izpeljavi enačbe, vendar ne nastopajo med prostimi spremenljivkami tipov A in B.

Izraz za term s, t	$::=$	a^A	prosta spremenljivka
	$\mid$	x	vezana spremenljivka
	$\mid$	$\mathsf{S}(e_1, \ldots, e_n)$	uporaba simbola za term S
	$\mid$	$\mathsf{M}^{\mathcal{B}}(t_1, \ldots, t_n)$	uporaba metaspremenjivke za term $\mathsf{M}^{\mathcal{B}}$
	$\mid$	$\kappa(t, \alpha)$	pretvorba
Množica predpostavk α, β	$::=$	$\{\!\!\{ \ldots, a_i^A, \ldots, x_j, \ldots, \mathsf{M}_k^{\mathcal{B}}, \ldots \}\!\!\}$	
Sodba j	$::=$	$A\ \mathsf{type}$	A je tip
	$\mid$	$t : A$	t ima tip T
	$\mid$	$A \equiv B\ \mathsf{by}\ \alpha$	A in B sta enaka tipa
	$\mid$	$s \equiv t : A\ \mathsf{by}\ \alpha$	s in t sta enaka terma pri tipu A
		$\vdots$	

Slika C.2: Surova sintaksa končnih kontekstno neodvisnih teorij tipov.

CF-VAR
$$\frac{}{\vdash \mathsf{a}^A : A}$$

CF-ABSTR
$$\frac{\vdash A\ \mathsf{type} \qquad \mathsf{a}^A \notin \mathsf{fv}(\mathcal{J}) \qquad \vdash \mathcal{J}[\mathsf{a}^A/x]}{\vdash \{x{:}A\}\ \mathcal{J}}$$

CF-CONV-TM
$$\frac{\vdash t : A \qquad \vdash A \equiv B\ \mathsf{by}\ \alpha \qquad \mathsf{asm}(t, A, B, \alpha) = \mathsf{asm}(t, B, \beta)}{\vdash \kappa(t, \beta) : B}$$

Slika C.3: Izvleček kontekstno neodvisnih pravil.

Pravilo za pretvorbo v teoriji tipov dovoljuje uporabo sodbene enakosti za konstrukcijo sodbe za term. Da bi zagotovili, da se množice predpostavk ne izgubijo v rezultatu pretvorbe, v sintakso vključimo *pretvorjene terme* (slika 3.1).

Po vzoru razvoja končnih teorij tipov vpeljemo surova kontekstno neodvisna pravila in surove kontekstno neodvisne teorije tipov (§3.1.2). Nadaljujemo z definicijo *končnih kontekstno neodvisnih pravil* in končnih kontekstno neodvisnih teorij tipov, za katere lahko dobro oblikovanost izpeljemo glede na dobro urejenost (definicija 3.1.13). Na koncu definiramo še *standardne* kontekstno neodvisne teorije (definicija 3.1.14). Verzija končnih kontekstno neodvisnih pravil in teorij tipov je zelo podobna originalni s konteksti.

Nato dokažemo metaizreke o surovi (§3.2.1), končni (§3.2.2) in standardni (§3.2.3) kontekstno neodvisni teoriji tipov. V razdelku 3.2.4 dokažemo, da v nasprotju s končnimi teorijami tipov surove kontekstno neodvisne teorije tipov zadoščajo krepitvi (izrek 3.2.16).

Izrek C.3.1 (Krepitev)**.** *Če surova kontekstno neodvisna teorija tipov izpelje*

$$\vdash \{\vec{y}{:}\vec{B}\}\{x{:}A\}\ \mathcal{J}$$

in $x \notin \mathsf{bv}(\mathcal{J})$*, potem izpelje tudi* $\vdash \{\vec{y}{:}\vec{B}\}\ \mathcal{J}$*.*

Konstrukcije, ki so podlaga za te metaizreke, so definirane na sodbah in ne na izpeljavah. Zato jih lahko *učinkovito* implementiramo v dokazovalnem pomočniku za kontekstno neodvisne teorije tipov in pri tem ne shranjujemo dreves izpeljav.

Na koncu pokažemo korespondenco med teorijami tipov in kontekstno neodvisnimi teorijami tipov ter konstruiramo prevedbi v obe smeri (§3.3). Da ločimo med obema verzijama teorije tipov uporabljamo predponi "tt" (za "tradicionalne tipe") in "cf" (za "kontekstno neodvisne tipe" (context-free).

Najprej pokažemo, kako prevedemo posamezne dele cf-teorij v ustrezne dele tt-teorij. Načrt je preprost: premaknemo oznake s spremenljivk v kontekste, pobrišemo pretvorbe in zamenjamo množice predpostavk z vrednostjo $\star$.

Izrek C.3.2 (Prevedba iz končnih cf- v tt-teorije)**.** *Naj bo T končna cf-teorija, katere prevedba T_{tt} je prav tako končna. Naj bo $\Theta;\Gamma$ tak tt-kontekst, da velja $\vdash_{T_{\mathrm{tt}}} \Theta\ \mathsf{mctx}$ in $\Theta \vdash_{T_{\mathrm{tt}}} \Gamma\ \mathsf{vctx}$. Če velja $\vdash_T \mathcal{J}$ in je $\Theta;\Gamma$ primeren za $\mathcal{J}$, potem velja $\Theta;\Gamma \vdash_{T_{\mathrm{tt}}} \lfloor \mathcal{J} \rfloor$.*

Posledica C.3.3. *Prevedba standardne cf-teorije je standardna tt-teorija.*

Prevedba iz tt-teorije v cf-teorije zahteva, da označimo spremenljivke z informacijami o tipih, vstavimo pretvorbe in rekonstruiramo množice predpostavk.

Izrek C.3.4 (Prevedba iz standardne tt- v cf-teorijo)**.** *Za vsako standardno tt-teorijo T, standardno cf-teorijo T', ki je primerna za T, in Θ, θ, Γ, γ kot zgoraj, če velja $\Theta;\Gamma \vdash_T \mathcal{J}$, potem obstaja primerna cf-sodba $\mathcal{J}'$ za $\mathcal{J}$ glede na θ, γ da velja $\vdash_{T'} \mathcal{J}'$.*

Prispevki. Predlagamo definicijo kontekstno neodvisne teorije tipov. Pokažemo, da kontekstno neodvisne teorije tipov zadostujejo uporabnim mataizrekom. Priskrbimo natančno povezavo s končnimi teorijami tipov. Da povzamemo:

- definiramo sintakso z označenimi spremenljivkami in metaspremenljivkami,
- definiramo kontekstno neodvisne sodbe z množicami predpostavk,
- definiramo primerna strukturna pravila (slike 3.8, 3.7, 3.6),
- razložimo lastnosti, ki naredijo kontekstno neodvisno teorijo tipov *končno* in *standardno*,
- dokažemo naslednje metaizreke:
  - dopustnost substitucije (izrek 3.2.4, 3.2.7),
  - izpeljivost predpostavk (izrek 3.2.5),
  - dopustnost instanciacije metaspremenjivk (trditev 3.2.8),
  - dopustnost "ekonomičnih" pravil (trditev 3.2.9, 3.2.10)
  - principi inverzije (izrek 3.2.14),
  - enoličnost tipiziranja (izrek 3.2.15),

  - dopustnost krepitve (izrek 3.2.16),
- podamo prevedbo končnih kontekstno neodvisnih teorij tipov v končne teorije tipov s konteksti (izrek 3.3.5),
- podamo prevedbo standardnih teorij tipov s konteksti v standardne kontekstno neodvisne teorije tipov (izrek 3.3.10).

C.4 Poglavje 4: Učinkovni metajezik za teorije tipov

V poglavju 4 predstavimo metajezik Andromeda (AML), učinkovni programski jezik, ki omogoča prikladno manipulacijo sodb in pravil kontekstno neodvisne teorije tipov ter podpira pogoste tehnike za razvoj dokazov.

Definicija kontekstno neodvisnih teorij tipov je primerna, da nam lahko služi kot jedro dokazovalnega pomočnika. AML kombinira primitivne pojme teorije tipov za konstrukcijo sodb, meja in pravil s programerskimi konstruktorji za splošno rabo, kot so funkcije, algebrajske učinkovne operacije in poganjalci (runners). Navdih za operacijsko semantiko AML je dvosmerno tipiziranje. V §4.1.1 pojasnimo kako deklarativno definicijo teorije tipov izboljšamo v dvosmerno. Nato uvedemo *dvosmerno evalvacijo*, ki jo bomo posplošili na kontekstno neodvisne teorije tipov. Učinkovnost AML se kaže v uporabi *operacij* in *poganjalcev*, kar pojasnimo v razdelku 4.1.2. Definiramo formalno sintakso AML izračunov, vrednosti in ukazov na najvišjem nivoju (§4.2).

Sintaksa. Izrazi v metajeziku Andromeda se delijo na inertne *vrednosti* in *izračune* s (potencialnimi) učinki, ki se evalvirajo v *rezultate*.

	Konkretna sintaksa	Abstraktna sintaksa	Pomen	Način
Izračun $C \ni c$ ::=				
	`return` v	`return`(v)	vrednost	$\rightsquigarrow$
	`let` x `=` c_1 `in` c_2	`let`$(c_1,\ x.\,c_2)$	lokalna definicija	$\rightsquigarrow$, ✓
	`match` v `with` $(p \Rightarrow c)^*$	`match`$(v,\ (p.c)^*)$	primer ujemanja	$\rightsquigarrow$, ✓
	$op\ \ v$	`perform`$(op,\ v)$	izvedi operacijo	$\rightsquigarrow$, ✓
	`with` v `run` c	`run`$(v,\ c)$	poženi s poganjalcem	$\rightsquigarrow$, ✓

Slika C.4: Sintaksa splošnih AML izračunov (izvleček).

Izračuni na sliki 4.4 so znani konstrukti iz programskih jezikov za splošno rabo.

Izračuni na sliki 4.5 implementirajo pravila za kontekstno neodvisne teorije tipov. Da lahko konstruiramo samo izpeljive sodbe, zagotovimo tako, da nikoli neposredno ne manipuliramo zgolj z deli sintakse, kot so surovi termi, ampak vedno le s sodbami.

Vrednosti v AML (slika 4.6) lahko ponovno klasificiramo kot tiste, ki pritičejo programskim jezikom za splošno rabo, in tiste, ki so namenjene teoriji tipov. Imamo spremenljivke, funkcije, poganjalce in konstruktorje podatkovnih tipov v AML.

Konkretna sintaksa	Abstraktna sintaksa	Pomen	Način
Izračun $C \ni c$ ::=			
`c as v`	`ascribe(c, v)`	pripis meje	$\rightsquigarrow$
`tt-var v`	`tt-var(v)`	sveža tt-spremenljivka	$\rightsquigarrow$
`tt-abstr` v_1 v_2	`tt-abstr`(v_1, v_2)	abstrakcija	$\rightsquigarrow$
$v_1 \overset{\alpha}{==} v_2$	`tt-alpha-equal`(v_1, v_2)	alfa enakost	$\rightsquigarrow$
`tt-convert` v_1 v_2	`tt-convert`(v_1, v_2)	pretvorba	$\rightsquigarrow$
∂v	`tt-bdry-of`(v)	meja sodbe	$\rightsquigarrow$

Slika C.5: Sintaksa AML izračunov za teorije tipov (izvleček).

Konkretna sintaksa	Abstraktna sintaksa	Pomen
Vrednost $V \ni v$ ::=		
x	`var`(x)	spremenljivka
`runner` $(\mid \textit{op-case})^*$	`runner`$(\textit{op-case}^*)$	poganjalec
$\varnothing$	$\mathcal{J}$	sodba, glej sliko 3.1
Rezultat $R \ni r$ ::=		
$\varnothing$	`val`(v)	vrednost
$\varnothing$	`op`$(op, v_1, v_2, x.c_\kappa)$	operacija

Slika C.6: Sintaksa AML vrednosti in rezultatov (izvleček).

Konkretna sintaksa	Abstraktna sintaksa	Pomen
Sklad vrhnjega poganjalca $\mathcal{R}$ ::=		
$\mathcal{R}[c] =$ `with` v_1 `run` ... `with` v_n `run` c		
Teorija $\mathbb{T}$ ::=		
$\{\ldots, R \mapsto ($`rule` $M_1 : \mathcal{B}_1 \Longrightarrow \ldots \Longrightarrow$ `rule` $M_n : \mathcal{B}_n \Longrightarrow j), \ldots\}$		
Ukaz na najvišjem nivoju cmd ::=		
`rule` R v	`rule`(R, v)	definicija pravila
Vzorec p ::=		
$?x$	`p-var`(x)	spremenljivka
$S(p^*)$	`p-sym`(S, p^*)	uporaba simbola

Slika C.7: Sintaksa AML ukazov na najvišjem nivoju in vzorcev (izvleček).

Program v AML je sestavljen iz zaporedja *ukazov na najvišjem nivoju* (slika 4.7). Vzorci na sliki 4.7, ki jih uporabljamo skupaj z `match`, implementirajo metaizreke teorije tipov. Izrek o inverziji je implementiran preko `p-sym`(S, p^*).

Operacijska semantika. Bistvo operacijske semantike za AML zajema kombinacija *dvosmerne evalvacije* z operacijami in poganjalci. Osrednji vidik AML je uporaba učinkov in operacije pokažejo uspešno interakcijo z dvosmerno evalvacijo (§4.3.1). Uporabljamo prezentacijo velikih korakov v stilu “drobnozrnatega neučakanega izvajanja” (fine-grained call-by-value) (Levy in sod. 2003). Definiramo dve evalvacijski

funkciji, synth-comp : $Th \times C \rightharpoonup R$ in check-comp : $Th \times C \times B \rightharpoonup R$. Tu Th, C, B, in R po vrsti pomenijo CFTT teorije, izračune, CFTT meje in rezultate. Uporabljali bomo oznako $\mathbb{T} \mid c \rightsquigarrow r$ za "c sintetizira r" in $\mathbb{T} \mid c \ @ \ \mathcal{B} \checkmark r$ za "ko c preverimo z $\mathcal{B}$ se evalvira v r". V oznaki za evalvacijski funkciji bomo praviloma opuščali argument $\mathbb{T}$, ki označuje teorijo.

$$\textsc{Syn-op}\ \frac{}{\texttt{perform}(op, v) \rightsquigarrow \texttt{op}(op, v, \mathsf{Unbounded}, x.\texttt{return}\ (x))}$$

$$\textsc{Chk-op}\ \frac{}{\texttt{perform}(op, v) \ @ \ \mathcal{B} \checkmark \texttt{op}(op, v, \mathsf{Boundary}(\mathcal{B}), x.\texttt{return}\ (x))}$$

Slika C.8: Operacijska semantika operacij.

$$\textsc{Chk-let-val}\ \frac{c_1 \rightsquigarrow \texttt{val}\ (v) \qquad c_2[v/x] \ @ \ \mathcal{B} \checkmark r}{\texttt{let}(c_1,\ x.c_2) \ @ \ \mathcal{B} \checkmark r}$$

$$\textsc{Chk-let-op}\ \frac{c_1 \rightsquigarrow \texttt{op}(op,\ v_1,\ v_2,\ y.c_\kappa)}{\texttt{let}(c_1,\ x.c_2) \ @ \ \mathcal{B} \checkmark \texttt{op}(op,\ v_1,\ v_2,\ y.\texttt{let}(c_\kappa,\ x.\texttt{ascribe}(c_2,\ \mathcal{B})))}$$

Slika C.9: Operacijska semantika vezave "let" (izvleček).

$$\textsc{Syn-run-op-handle}\ \frac{\begin{array}{c} c \rightsquigarrow \texttt{op}(op, v_1, v_2, z.c_\kappa) \\ v = \texttt{runner}(\ldots,\ \texttt{case-op}(op,\ x.y.c_{op}),\ \ldots) \\ \left(\begin{array}{l} \texttt{let}\ z = \texttt{match}\ v_2\ \texttt{with} \\ \quad \mid\ \texttt{Unbounded} \to c_{op}[v_1/x, v_2/y] \\ \quad \mid\ \texttt{Boundary ?B} \to c_{op}[v_1/x, v_2/y]\ \texttt{as B} \\ \texttt{in with}\ v\ \texttt{run}\ c_\kappa \end{array}\right) \rightsquigarrow r \end{array}}{\texttt{with}\ v\ \texttt{run}\ c \rightsquigarrow r}$$

$$\textsc{Chk-run-op-handle}\ \frac{\begin{array}{c} c \ @ \ \mathcal{B} \checkmark \texttt{op}(op, v_1, v_2, z.c_\kappa) \\ v = \texttt{runner}(\ldots,\ \texttt{case-op}(op,\ x.y.c_{op}),\ \ldots) \\ \left(\begin{array}{l} \texttt{let}\ z = \texttt{match}\ v_2\ \texttt{with} \\ \quad \mid\ \texttt{Unbounded} \to c_{op}[v_1/x, v_2/y] \\ \quad \mid\ \texttt{Boundary ?B} \to c_{op}[v_1/x, v_2/y]\ \texttt{as B} \\ \texttt{in with}\ v\ \texttt{run}\ c_\kappa \end{array}\right) @ \ \mathcal{B} \checkmark r \end{array}}{\texttt{with}\ v\ \texttt{run}\ c \ @ \ \mathcal{B} \checkmark r}$$

Slika C.10: Operacijska semantika poganjalcev (izvleček).

Ko se c evalvira v operacijo op in ima poganjalec v vejo $x.y.c_{op}$, ki pripada operaciji op, se dviganje operacije op ustavi. Kontinuacija c_κ pričakuje, da bo rezultat poganjalca vezan na z. Torej evalviramo let-stavek, ki na z veže vrednost, ki jo pridela veja c_{op} poganjalca. Spomnimo se iz Chk-op, da operacija, ki se izvede v načinu

preverjanja, shrani svojo mejo za preverjanje kot `Boundary B`, za razliko od SYN-OP, ki bo pridelal `Unbounded` za v_2. Veja c_{op} poganjalca se evalvira v načinu sinteze, če velja v_2 = `Unbounded`. Če pa je v_2 = `Boundary B`, potem se je izvedla operacija med preverjanjem z `B` in tako je c_{op} ovita s pripisom meje `as B`.

Evalvacija izrazov v AML, ki se nanašajo na teorijo tipov, tesno sledi pravilom kontekstno neodvisnih teorij tipov.

$$\text{SYN-ASCR}\ \frac{c \ @\ \mathcal{B} \ \checkmark\ \mathcal{J}}{\texttt{ascribe}(c,\ \mathcal{B}) \ \rightsquigarrow\ \mathcal{J}} \qquad \text{CHK-SYN}\ \frac{\begin{array}{l}\texttt{let x} = c\ \texttt{in} \\ \texttt{let B} = \partial\texttt{x in} \\ \texttt{let b} = \texttt{B} \overset{\alpha}{==} \mathcal{B}\ \texttt{in} \\ \texttt{match b with} \\ \texttt{| true} \to \texttt{return x} \\ \texttt{| false} \to (\texttt{coerce x})\ \texttt{as}\ \mathcal{B}\end{array} \ \rightsquigarrow\ r}{c \ @\ \mathcal{B} \ \checkmark\ r}$$

Slika C.11: Operacijska semantika pripisa meje in menjave načina.

Pripis meje (SYN-ASCR) nam omogoča, da prisilno zamenjamo način iz sinteze v preverjanje. Če bi radi v načinu preverjanja evalvirali izračun c, ki je naravno v načinu sinteze, uporabimo pravilo CHK-SYN. Če za c ne moremo uporabiti informacije o meji, ki nam je na voljo, tak primer je c = `return` (v), potem izračun sintetiziramo in vežemo na `x`. Če je meja za preverjanje $\mathcal{B}$ alfa-enaka sintetizirani meji ∂ `x`, rezultat vrnemo. Če zaznamo neujemanje, izvedemo `coerce x` pod pripisom meje $\mathcal{B}$.

Prišli smo do pametnih konstruktorjev za teorijo tipov.

$$\text{SYN-TT-VAR}\ \frac{\mathsf{a}^A \ \text{fresh}}{\texttt{tt-var}(A\ \mathsf{type}) \ \rightsquigarrow\ \texttt{val}\ (\mathsf{a}^A : A)}$$

$$\text{SYN-TT-ABSTR}\ \frac{v_1 = \mathsf{a}^A : A \qquad v_2 = \mathcal{J} \text{ or } \mathcal{B} \qquad \mathsf{a}^A \notin \mathsf{fvt}(v_2)}{\texttt{tt-abstr}(v_1,\ v_2) \ \rightsquigarrow\ \texttt{val}\ (\{x{:}A\}\ v_2[x/\mathsf{a}^A])}$$

$$\text{SYN-TT-SUBST}\ \frac{v = \mathcal{J} \text{ or } \mathcal{B}}{\texttt{tt-subst}((\{x{:}A\}\ v),\ (t : A)) \ \rightsquigarrow\ \texttt{val}\ (v[t/x])}$$

Slika C.12: Operacijska semantika spremenljivk teorije tipov.

Dopustnost substitucije je implementirana preko SYN-TT-SUBST.

$$\text{SYN-TT-BDRY-OF}\ \frac{\mathcal{J} = \mathcal{B}\boxed{e}}{\texttt{tt-bdry-of}(\mathcal{J}) \ \rightsquigarrow\ \texttt{val}\ (\mathcal{B})}$$

Slika C.13: Operacijska semantika za projekcijo meje v teoriji tipov.

Dano sodbo lahko projiciramo na svojo mejo preko SYN-TT-BDRY-OF. Ta operacija realizira izrek 3.2.5 o izpeljivosti predpostavk v kontekstno neodvisni teoriji tipov.

Izpeljane oblike. Da bi pokazali kako ekspresiven je metajezik AML in priskrbeli uporabnikom-prijazen jezik, definiramo nekaj *izpeljanih oblik* (§4.4). Vsaka izmed izpeljanih oblik ima ekvivalentno obliko v AML. Slednja definira pomen prve in inducira operacijsko semantiko.

$c_R\ c_{arg}$
```
let R = c_R in
match R with rule (_ : ?bdry) _ →
  let a = c_arg as bdry in
  tt-inst R a
```
$\rightsquigarrow$

SYN-TT-APPLY-VAL-VAL

$$\frac{c_R \rightsquigarrow \texttt{val}\,(\texttt{rule}\,(M : \mathcal{B}) \Longrightarrow v) \qquad c_{arg}\ @\ \mathcal{B}\ \checkmark\ \texttt{val}\,(\mathcal{B}\boxed{e})}{c_R\ c_{arg} \rightsquigarrow \texttt{val}\,(v[e/M])}$$

Slika C.14: Sintaksa in inducirana operacijska semantika uporabe pravila.

Izpeljana oblika za uporabo pravila (slika 4.24) nam dovoljuje, da lahko zaporedno napišemo več iteriranih instanciacij metaspremenljivk. Ponavadi je c_R pravilo ali nadaljnja uporaba pravila. V tej situaciji bo vsaka naslednja uporaba pravila v c_R po vrsti evalvirala svoje argumente v načinu preverjanja z mejo, ki pripada premisi pravila instanciirani s prejšnjimi argumenti.

Predstavimo AML program, ki implementira lemo 3.2.17 in dovoljuje uporabniku, da transparentno dela s kontekstno neodvisnimi teorijami tipov ter ignorira pretvorbe (§4.4.2). Pripadajoča programska koda se nahaja v dodatku A.

Podamo kratko skico izrekov skladnosti in polnosti za AML za kontekstno neodvisne teorije tipov.

Domneva C.4.1 (Skladnost AML)**.** *Naj bo $\mathbb{T}$ standardna kontekstno neodvisna teorija tipov. Če velja $\mathbb{T} \mid c \rightsquigarrow \texttt{val}\,(\mathcal{J})$, potem je $\vdash \mathcal{J}$ izpeljiva v $\mathbb{T}$.*

Domneva C.4.2 (Polnost AML)**.** *Naj bo $\mathbb{T}$ standardna kontekstno neodvisna teorija tipov. Če ima $\mathcal{J}$ dobro-tipizirane oznake in velja $\vdash_{\mathbb{T}} \mathcal{J}$, potem obstaja AML program c, da velja $\mathbb{T} \mid c \rightsquigarrow \texttt{val}\,(\mathcal{J}')$ in $\lfloor \mathcal{J} \rfloor = \lfloor \mathcal{J}' \rfloor$.*

Podrobni dokazi obeh domnev so prepuščeni nadaljnjim raziskavam.

AML v Andromedi 2. Dokazovalni pomočnik Andromeda 2 (Bauer, Haselwarter in Petković Komel 2021) je implementacija ene od različic AML. Vmesnik za evalvacijo pametnih konstruktorjev za teorijo tipov zagotavlja jedro, ki je napisano v 3000 vrsticah OCaml programske kode in ustreza pravilom za kontekstno neodvisne teorije tipov.

Primer formalizacije z Andromedo 2 se nahaja v dodatku B. Več primerov, kot na primer definicijo računa konstrukcij, lahko najdemo v podmapi `theories/` izvorne kode za Andromedo.

Podamo tudi definicijo Harperjevega logičnega okvirja za enakost (Harper 2021) v Andromedi 2 (§B).

Prispevki. AML je praktičen visokonivojski učinkovni programski jezik za kontekstno neodvisne teorije tipov, ki podpira razvoj dokazov v teorijah tipov, ki jih določi uporabnik. Da povzamemo:

- vgradimo kontekstno neodvisne teorije tipov v programski jezik v stilu jezika ML,
- razširimo dvosmerno preverjanje tipov v dvosmerno evalvacijo,
- pokažemo, kako uporabiti poganjalce za razvoj dokazov z lokalnimi hipotezami,
- priskrbimo mehanizem za pravila, ki jih lahko definira uporabnik, ter mehanizem za izpeljana pravila,
- implementiramo metaizreke standardne kontekstno neodvisne teorije tipov v različici z učinki,
- implementiramo AML v dokazovalnem pomočniku Andromeda 2.

C.5 Poglavje 5: Zaključek

V poglavju 5 podamo pregled raziskovalnega področja in povezanih raziskav, ter zarišemo okvirne smernice za nadaljnje delo.

V razdelku 5.1.1 razpravljamo o povezavi med končnimi teorijami tipov in ostalimi splošnimi definicijami teorije tipov, ki so bile predlagane nedavno. Primerjamo pristop AML z obstoječimi učinkovnimi metajeziki, z dokazovalnimi pomočniki, ki jih uporabniki lahko razširijo, in z našim preteklim delom (§5.1.2). V razdelku 5.2 predlagamo naslednje korake za metateoretično študijo končnih teorij tipov in kontekstno neodvisnih teorij tipov. Skiciramo tudi nekaj zanimivih razširitev. Na koncu predlagamo teoretična in praktična vprašaja ter možne razširitve AML.

Appendix D

Bibliography

Aczel, Peter (1977). 'An Introduction to Inductive Definitions'. In: *Studies in Logic and the Foundations of Mathematics* 90. [↗] (cited on page 42).

Ahman, Danel and Andrej Bauer (2019). *Runners in Action*. arXiv: 1910.11629 [cs]. [↗] (cited on pages 26, 119, 126, 136, 152).

Ahman, Danel, Neil Ghani and Gordon D. Plotkin (2016). 'Dependent Types and Fibred Computational Effects'. In: *Foundations of Software Science and Computation Structures*. FoSSaCS'16. [↗] (cited on page 134).

Altenkirch, Thorsten, Paolo Capriotti and Nicolai Kraus (2016). 'Extending Homotopy Type Theory with Strict Equality'. In: *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). [↗] (cited on page 21).

Bauer, Andrej, Philipp G. Haselwarter and Anja Petković Komel (2021). *The Andromeda Proof Assistant*. [↗] (cited on pages 152, 188).

New Spaces in Mathematics (2021). *New Spaces in Mathematics: Formal and Conceptual Reflections*. Vol. 1. [↗] (cited on page 161).

Angiuli, Carlo, Kuen-Bang Hou (Favonia) and Robert Harper (2018). 'Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities'. In: *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. Vol. 119. Leibniz International Proceedings in Informatics (LIPIcs). [↗] (cited on page 161).

Annenkov, Danil, Paolo Capriotti, Nicolai Kraus and Christian Sattler (2019). *Two-Level Type Theory and Applications*. arXiv: 1705.03307 [cs]. [↗] (cited on page 156).

Asperti, Andrea, Wilmer Ricciotti, Claudio Sacerdoti Coen and Enrico Tassi (2009). 'Hints in Unification'. In: *Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science 5674. [↗] (cited on page 26).

— (2012). *A Bi-Directional Refinement Algorithm for the Calculus of (Co) Inductive Constructions*. arXiv: 1202.4905. [↗] (cited on pages 123, 160).

Assaf, Ali, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant and Ronan Saillard (n.d.). *Dedukti: A Logical Framework Based on the λΠ-Calculus modulo Theory* (cited on page 159).

Awodey, Steve (2014). 'Structuralism, Invariance, and Univalence'. In: *Philosophia Mathematica* 22.1. [↗] (cited on page 23).

Awodey, Steve and Andrej Bauer (2004). 'Propositions as [Types]'. In: *Journal of Logic and Computation* 14.4. [↗] (cited on page 25).

Awodey, Steve and Michael A. Warren (2007). 'Homotopy Theoretic Models of Identity Types'. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 146.1. arXiv: `0709.0248`. [↗] (cited on page 22).

Barras, Bruno, Jean-Pierre Jouannaud, Pierre-Yves Strub and Qian Wang (2011). '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*. [↗] (cited on page 24).

Bauer, Andrej, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar and Christopher A. Stone (2018). 'Design and Implementation of the Andromeda Proof Assistant'. In: in collab. with Michael Wagner. [↗] (cited on pages 25, 137, 159, 160).

Bauer, Andrej, Philipp G. Haselwarter and Peter LeFanu Lumsdaine (2020). *A general definition of dependent type theories*. arXiv: `2009.05539 [math.LO]` (cited on pages 22, 33, 34, 36, 41, 155, 157, 161).

Bauer, Andrej, Philipp G. Haselwarter and Anja Petković (2020). 'Equality Checking for General Type Theories in Andromeda 2'. In: *Mathematical Software – ICMS 2020*. Lecture Notes in Computer Science (cited on pages 159, 160).

Bauer, Andrej and Anja Petković Komel (2021). *An Extensible Equality Checking Algorithm for Dependent Type Theories*. arXiv: `2103.07397 [cs, math]`. [↗] (cited on pages 159, 160).

Bauer, Andrej and Matija Pretnar (2014). 'An Effect System for Algebraic Effects and Handlers'. In: *Logical Methods in Computer Science* 10.4. [↗] (cited on page 152).

— (2015). 'Programming with Algebraic Effects and Handlers'. In: *Journal of Logical and Algebraic Methods in Programming*. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) 84.1. [↗] (cited on pages 26, 129, 159).

Birkedal, Lars, Andreas Nuyts, G. A. Kavvos and Daniel Gratzer (2021). 'Multimodal Dependent Type Theory'. In: *Logical Methods in Computer Science* Volume 17, Issue 3. [↗] (cited on page 161).

Boulier, Simon, Pierre-Marie Pédrot and Nicolas Tabareau (2017). 'The next 700 Syntactical Models of Type Theory'. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. [↗] (cited on page 24).

Boulier, Simon Pierre (2018). 'Extending Type Theory with Syntactic Models'. PhD thesis. Ecole nationale supérieure Mines-Télécom Atlantique. [↗] (cited on page 24).

Bruggeman, Carl, Oscar Waddell and R. Kent Dybvig (1996). 'Representing Control in the Presence of One-Shot Continuations'. In: *ACM SIGPLAN Notices* 31.5. [↗] (cited on page 137).

Cartmell, J.W. (1978). 'Generalised Algebraic Theories and Contextual Categories'. PhD thesis. University of Oxford. [↗] (cited on pages 20, 22, 155).

Castellan, Simon, Pierre Clairambault and Peter Dybjer (2017). 'Undecidability of Equality in the Free Locally Cartesian Closed Category (Extended Version)'. In: *Logical Methods in Computer Science* Volume 13, Issue 4. [↗] (cited on page 21).

Cavallo, Evan, Anders Mörtberg and Andrew W. Swan (2020). 'Unifying Cubical Models of Univalent Type Theory'. In: *28th EACSL Annual Conference on Computer Science Logic (CSL 2020)*. Vol. 152. Leibniz International Proceedings in Informatics (LIPIcs). [↗] (cited on page 161).

The Coq development team (2021a). *The Coq Proof Assistant*. Version 8.13. [↗] (cited on pages 23, 157, 158).

— (2021b). *Setting Properties of a Function's Arguments*. Version 8.13.2. [↗] (cited on page 123).

— (2021c). *Core Language*. Version 8.13.2. [↗] (cited on page 158).

Cervesato, Iliano and Frank Pfenning (2002). 'A Linear Logical Framework'. In: *Information and Computation* 179.1. [↗] (cited on page 156).

Charguéraud, Arthur (2012). 'The Locally Nameless Representation'. In: *Journal of Automated Reasoning* 49 (cited on page 34).

Christiansen, David and Edwin Brady (2016). 'Elaborator Reflection: Extending Idris in Idris'. In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. [↗] (cited on page 159).

Church, Alonzo (1940). 'A Formulation of the Simple Theory of Types'. In: *The Journal of Symbolic Logic* 5.2. JSTOR: `2266170` (cited on page 20).

Cockx, Jesper and Andreas Abel (2016). 'Sprinkles of Extensionality for Your Vanilla Type Theory'. TYPES 2016 (Novi Sad, Serbia). [↗] (cited on page 24).

Cockx, Jesper, Nicolas Tabareau and Théo Winterhalter (2020). 'The Taming of the Rew: A Type Theory with Computational Assumptions'. In: *Proceedings of the ACM on Programming Languages*. POPL 2021. [↗] (cited on page 159).

Cohen, Cyril, Thierry Coquand, Simon Huber and Anders Mörtberg (2016). *Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom*. arXiv: `1611.02108 [cs, math]`. [↗] (cited on pages 156, 161).

Constable, Robert L., Stuart F. Allen, H. M. Bromley, Walter Rance Cleaveland, J. F. Cremer, Robert William Harper, Douglas J. Howe, Todd B. Knoblock, Nax Paul Mendler, Prakash Panangaden, James T. Sasaki and Scott F. Smith (1986). *Implementing Mathematics with the Nuprl Proof Development System*. [↗] (cited on pages 20, 23, 157).

Coquand, Thierry (1996). 'An Algorithm for Type-Checking Dependent Types'. In: *Science of Computer Programming* 26.1. [↗] (cited on page 119).

Coquand, Thierry and Gérard Huet (1988). 'The Calculus of Constructions'. In: *Inf. Comput.* 76.2/3 (cited on pages 19, 20, 177).

Cousineau, Denis and Gilles Dowek (2007). 'Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo'. In: *Typed Lambda Calculi and Applications*. Lecture Notes in Computer Science (cited on pages 156, 159).

Curien, Pierre-Louis (1993). 'Substitution up to Isomorphism'. In: *Fundam. Inf.* 19.1-2 (cited on page 22).

Danvy, Olivier (1992). 'Back to Direct Style'. In: *ESOP '92*. Lecture Notes in Computer Science (cited on page 26).

De Bruijn, Nicolaas G. (1972). 'Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem'. In: *Indagationes Mathematicae* 75.5. [↗] (cited on page 34).

Delahaye, David (2000). 'A Tactic Language for the System Coq'. In: *Logic for Programming and Automated Reasoning*. Vol. 1955. Lecture Notes in Artificial Intelligence. [↗] (cited on page 158).

De Moura, Leonardo, Soonho Kong, Jeremy Avigad, Floris van Doorn and Jakob von Raumer (2015). 'The Lean Theorem Prover (System Description)'. In: *Automated Deduction - CADE-25*. Lecture Notes in Computer Science (cited on pages 23, 157).

Dunfield, Jana (2014). 'Elaborating Intersection and Union Types'. In: *Journal of Functional Programming* 24.2-3. [↗] (cited on page 156).

Dunfield, Jana and Neel Krishnaswami (2021). 'Bidirectional Typing'. In: *ACM Computing Surveys* 54.5. [↗] (cited on pages 119, 122).

Ebner, Gabriel, Sebastian Ullrich, Jared Roesch, Jeremy Avigad and Leonardo de Moura (2017). 'A Metaprogramming Framework for Formal Verification'. In: *Proceedings of the ACM on Programming Languages* 1 (ICFP). [↗] (cited on page 159).

Fiore, Marcelo and Ola Mahmoud (2014). *Functorial Semantics of Second-Order Algebraic Theories*. arXiv: `1401.4697 [cs, math]`. [↗] (cited on page 155).

FSF (2021a). *AUCTeX*. Version 13.0.11. Free Software Foundation, Inc. (cited on page 10).

— (2021b). *GNU Emacs*. Version 28.0.50. Free Software Foundation, Inc. (cited on page 10).

Geuvers, Herman, Robbert Krebbers, James McKinna and Freek Wiedijk (2010). 'Pure Type Systems without Explicit Contexts'. In: *Electronic Proceedings in Theoretical Computer Science* 34 (cited on pages 75, 155, 181).

Girard, Jean-Yves (1972). 'Interprétation Fonctionelle et Élimination Des Coupures de l'arithmétique d'ordre Supérieur'. PhD thesis. Université Paris VII (cited on page 20).

Gonthier, Georges, 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 (2013). 'A Machine-Checked Proof of the Odd Order Theorem'. In: *Interactive Theorem Proving*. Lecture Notes in Computer Science (cited on page 23).

Gonthier, Georges, Assia Mahboubi and Enrico Tassi (2015). *A Small Scale Reflection Extension for the Coq System*. report. Inria Saclay Ile de France. [↗] (cited on page 23).

Gordon, Mike (2000). 'From LCF to HOL: A Short History.' In: *Proof, Language, and Interaction*. [↗] (cited on page 25).

Gordon, Mike, Robin Milner, Lockwood Morris, Malcolm Newey and Christopher Wadsworth (1978). 'A Metalanguage for Interactive Proof in LCF'. In: *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. [↗] (cited on pages 20, 157).

Gordon, Mike, Robin Milner and Christopher Wadsworth (1979). *Edinburgh LCF: A Mechanised Logic of Computation*. Lecture Notes in Computer Science 78. [↗] (cited on pages 25, 158).

Gratzer, Daniel (2021). *Normalization for Multimodal Type Theory*. arXiv: `2106.01414 [cs]`. [↗] (cited on page 161).

Gratzer, Daniel, Jonathan Sterling and Lars Birkedal (2019). 'Implementing a Modal Dependent Type Theory'. In: *Proceedings of the ACM on Programming Languages* 3 (ICFP). [↗] (cited on page 162).

Harper, Robert (1985). 'Aspects of the Implementation of Type Theory'. PhD thesis. Cornell University. [↗] (cited on page 121).

— (2021). *An Equational Logical Framework for Type Theories*. arXiv: `2106.01484 [cs, math]`. [↗] (cited on pages 30, 156, 157, 161, 165, 188).

Harper, Robert, Bruce F. Duba and David Macqueen (1993). 'Typing First-Class Continuations in ML†'. In: *Journal of Functional Programming* 3.4. [↗] (cited on page 158).

Harper, Robert, Furio Honsell and Gordon Plotkin (1993). 'A Framework for Defining Logics'. In: *Journal of the ACM* 40.1. [↗] (cited on pages 25, 155, 156).

Harper, Robert and Robert Pollack (1991). 'Type Checking with Universes'. In: *Theoretical computer science* 89.1. [↗] (cited on pages 124, 157).

Herbelin, Hugo (2015). 'A Dependently-Typed Construction of Semi-Simplicial Types'. In: *Mathematical Structures in Computer Science* 25.05. [↗] (cited on page 21).

Hofmann, Martin (1994). 'On the Interpretation of Type Theory in Locally Cartesian Closed Categories'. In: *CSL* (cited on page 22).

— (1997). *Extensional Constructs in Intensional Type Theory*. CPHC/BCS Distinguished Dissertations. [↗] (cited on pages 21, 24).

Hofmann, Martin and Thomas Streicher (1994). 'The Groupoid Model Refutes Uniqueness of Identity Proofs'. In: *Proceedings Ninth Annual IEEE Symposium on Logic in Computer Science*. LiCS94 (cited on page 20).

Huet, Gérard (1988). *Extending the Calculus of Constructions with Type:Type* (cited on page 26).

The Isabelle development team (2016). *Isabelle*. [↗] (cited on page 157).

Isaev, Valery (2016). *Algebraic Presentations of Dependent Type Theories*. arXiv: `1602.08504 [cs, math]`. [↗] (cited on page 155).

Johnstone, Peter T. (2003). *Sketches of an Elephant: A Topos Theory Compendium* (cited on page 18).

Kaiser, Jan-Oliver, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas and Derek Dreyer (2018). 'Mtac2: Typed Tactics for Backward Reasoning in Coq'. In: *Proceedings of the ACM on Programming Languages* 2 (ICFP). [↗] (cited on page 158).

Kammar, Ohad (2014). 'Algebraic Theory of Type-and-Effect Systems'. PhD thesis. [↗] (cited on page 161).

Kapulkin, Chris and Peter LeFanu Lumsdaine (2012). *The Simplicial Model of Univalent Foundations (after Voevodsky)*. arXiv: `1211.2851 [math]`. [↗] (cited on page 21).

Karachalias, Georgios, Filip Koprivec, Matija Pretnar and Tom Schrijvers (2021). 'Efficient Compilation of Algebraic Effect Handlers'. In: *Proceedings of the ACM on Programming Languages* 5 (OOPSLA). [↗] (cited on page 137).

Kirchner, Florent and César Muñoz (2010). 'The Proof Monad'. In: *The Journal of Logic and Algebraic Programming* 79.3–5. [↗] (cited on page 26).

Kiselyov, Oleg and K. C. Sivaramakrishnan (2018). 'Eff Directly in OCaml'. In: *Electronic Proceedings in Theoretical Computer Science* 285. arXiv: `1812.11664`. [↗] (cited on page 162).

Lennon-Bertrand, Meven (2021). 'Complete Bidirectional Typing for the Calculus of Inductive Constructions'. In: *12th International Conference on Interactive Theorem Proving (ITP 2021)*. Vol. 193. Leibniz International Proceedings in Informatics (LIPIcs). [↗] (cited on page 122).

Leroy, Xavier, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy and Jérôme Vouillon (2021). *The OCaml System, Release 4.12.0*. [↗] (cited on page 158).

Levy, Paul Blain, John Power and Hayo Thielecke (2003). 'Modelling Environments in Call-by-Value Programming Languages'. In: *Information and Computation* 185.2. [↗] (cited on pages 133, 185).

Lukšič, Žiga (2020). 'Applications of Algebraic Effect Theories'. PhD thesis. University of Ljubljana. [↗] (cited on pages 152, 161).

Luo, Zhaohui (1990). 'An Extended Calculus of Constructions'. PhD thesis. University of Edinburgh (cited on page 157).

Madsen, Lars and Peter R. Wilson (2021). *Memoir*. Version v3.7o. [↗] (cited on page 10).

Mahboubi, Assia and Enrico Tassi (2013). 'Canonical Structures for the Working Coq User'. In: *Interactive Theorem Proving*. Lecture Notes in Computer Science (cited on pages 26, 129).

Martin-Löf, Per (1979). 'Constructive Mathematics and Computer Programming'. In: *Logic, Methodology and Philosophy of Science VI, Proceedings of the Sixth International Congress, Hannover 1979*. Vol. 104. Studies in Logic and the Foundations of Mathematics. [↗] (cited on page 20).

— (1982). 'Constructive Mathematics and Computer Programming'. In: *Studies in Logic and the Foundations of Mathematics*. Vol. Volume 104. [↗] (cited on pages 20, 126, 157).

— (1998). ‘An Intuitionistic Theory of Types’. In: *Twenty-Five Years of Constructive Type Theory*. Oxford Logic Guides 36 (cited on pages 19, 20, 177).

McBride, Conor (2018). *Basics of Bidirectionalism*. pigworker in a space. [↗] (cited on page 122).

McKinna, James and Robert Pollack (1993). ‘Pure Type Systems Formalized’. In: *International Conference on Typed Lambda Calculi and Applications (TLCA)*. Vol. 664. Lecture Notes in Computer Science (cited on page 34).

Milner, Robin (1972). *Logic for Computable Functions : Description of a Machine Implementation*. Defense Technical Information Center. [↗] (cited on page 20).

— (1978). ‘A Theory of Type Polymorphism in Programming’. In: *Journal of Computer and System Sciences* 17.3. [↗] (cited on pages 20, 134).

Milner, Robin, Mads Tofte and Robert Harper (1990). *The Definition of Standard ML* (cited on page 145).

Norell, Ulf (2007). ‘Towards a Practical Programming Language Based on Dependent Type Theory’. PhD thesis. Chalmers University of Technology (cited on pages 23, 123).

— (2009). ‘Dependently Typed Programming in Agda’. In: *Advanced Functional Programming: 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. Lecture Notes in Computer Science. [↗] (cited on page 157).

Pédrot, Pierre-Marie (2019). ‘Ltac2: Tactical Warfare’. In: CoqPL’19 (cited on pages 26, 158).

Pédrot, Pierre-Marie and Nicolas Tabareau (2017). ‘An Effectful Way to Eliminate Addiction to Dependence’. In: [↗] (cited on page 24).

— (2019). ‘The Fire Triangle: How to Mix Substitution, Dependent Elimination, and Effects’. In: *Proceedings of the ACM on Programming Languages* 4 (POPL). [↗] (cited on page 134).

Petković Komel, Anja (2021). ‘Towards an Elaboration Theorem’. Invited Talk. HoTT/UF 2021 (cited on page 161).

Pfenning, Frank (2001). ‘Logical Frameworks’. In: *Handbook of Automated Reasoning (in 2 volumes)* (cited on page 156).

— (2004). ‘Lecture Notes on Bidirectional Type Checking’. [↗] (cited on page 122).

Pfenning, Frank and Carsten Schürmann (1999). ‘System Description: Twelf — A Meta-Logical Framework for Deductive Systems’. In: *Automated Deduction — CADE-16*. Lecture Notes in Computer Science (cited on pages 156, 159).

Pientka, Brigitte (2015). *Mechanizing Types and Programming Languages: A Companion*. [↗] (cited on page 156).

Pientka, Brigitte and Jana Dunfield (2010). ‘Beluga: A Framework for Programming and Reasoning with Deductive Systems (System Description)’. In: *Automated Reasoning*. [↗] (cited on pages 156, 159).

Pierce, Benjamin C. (2002). *Types and Programming Languages* (cited on page 121).

— (2005). *Advanced Topics in Types and Programming Languages* (cited on page 121).

Pierce, Benjamin C. and David N. Turner (1998). 'Local Type Inference'. In: *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Diego, California* (cited on page 122).

Pinckney, Donald, Arjun Guha and Yuriy Brun (2020). 'Wasm/k: Delimited Continuations for WebAssembly'. In: *Proceedings of the 16th ACM SIGPLAN International Symposium on Dynamic Languages*. DLS 2020. [↗] (cited on page 162).

Plotkin, Gordon D. and John Power (2003). 'Algebraic Operations and Generic Effects'. In: *Applied Categorical Structures* 11.1. [↗] (cited on page 129).

Plotkin, Gordon D. and Matija Pretnar (2013). 'Handling Algebraic Effects'. In: *Logical Methods in Computer Science* Volume 9, Issue 4. [↗] (cited on page 161).

Pollack, Robert (1992). 'Implicit Syntax'. In: *Informal Proceedings of First Workshop on Logical Frameworks* (cited on page 124).

Pretnar, Matija (2010). 'Logic and Handling of Algebraic Effects'. PhD thesis. University of Edinburgh. [↗] (cited on pages 151, 152).

— (2015). 'An Introduction to Algebraic Effects and Handlers. Invited Tutorial Paper'. In: *Electronic Notes in Theoretical Computer Science* 319. [↗] (cited on page 126).

Rémy, Didier (2015). *Mathpartir*. Version 1.3.1 (cited on page 10).

Reppy, John H. (1991). 'CML: A Higher Concurrent Language'. In: *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. PLDI '91. [↗] (cited on page 158).

Reynolds, John C. (1974). 'Towards a Theory of Type Structure'. In: *Colloque Sur La Programmation, Paris, France*. Vol. 19. Lecture Notes in Computer Science (cited on page 20).

Russell, Bertrand (1903). *The Principles of Mathematics*. [↗] (cited on page 19).

— (1908). 'Mathematical Logic as Based on the Theory of Types'. In: *American Journal of Mathematics* 30.3. JSTOR: 2369948 (cited on page 19).

Saïbi, Amokrane (1997). 'Typing Algorithm in Type Theory with Inheritance'. In: *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '97. [↗] (cited on page 160).

Scholze, Peter (2019). *Lectures on Condensed Mathematics* (cited on page 161).

Schreiber, Urs and Michael Shulman (2014). 'Quantum Gauge Field Theory in Cohesive Homotopy Type Theory'. In: *Electronic Proceedings in Theoretical Computer Science* 158. arXiv: 1408.0054. [↗] (cited on page 161).

Schuster, Philipp, Jonathan Immanuel Brachthäuser and Klaus Ostermann (2020). 'Compiling Effect Handlers in Capability-Passing Style'. In: *Proceedings of the ACM on Programming Languages* 4 (ICFP). [↗] (cited on page 137).

Seely, Robert A. G. (1984). 'Locally Cartesian Closed Categories and Type Theory'. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 95.01. [↗] (cited on pages 20, 22).

Shulman, Mike (2014). *Homotopy Type Theory Should Eat Itself (but so Far, It's Too Big to Swallow)*. Homotopy Type Theory. [↗] (cited on page 21).

Sivaramakrishnan, KC, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer and Anil Madhavapeddy (2021). 'Retrofitting Effect Handlers onto OCaml'. In: *Proceedings*

of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021. [↗] (cited on pages 137, 158, 162).
Sozeau, Matthieu and Nicolas Oury (2008). 'First-Class Type Classes'. In: *Theorem Proving in Higher Order Logics*. Lecture Notes in Computer Science (cited on page 129).
Spiwack, Arnaud (2010). 'An Abstract Type for Constructing Tactics in Coq'. In: Proof Search in Type Theory. [↗] (cited on pages 26, 158).
Streicher, Thomas (1991). *Semantics of Type Theory*. [↗] (cited on page 157).
— (2011). 'A Model of Type Theory in Simplicial Sets'. [↗] (cited on page 21).
Swamy, Nikhil, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue and Santiago Zanella-Béguelin (2016). 'Dependent Types and Multi-Monadic Effects in F*'. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. [↗] (cited on page 134).
Tarski, Alfred (1955). 'A lattice-theoretical fixpoint theorem and its applications'. In: *Pacific Journal of Mathematics* 5.2 (cited on page 42).
Uemura, Taichi (2019). *A General Framework for the Semantics of Type Theory*. arXiv: `1904.04097 [cs, math]`. [↗] (cited on pages 156, 157, 161).
Univalent Foundations Program, The (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. [↗] (cited on page 21).
Van der Walt, Paul and Wouter Swierstra (2013). 'Engineering Proof by Reflection in Agda'. In: *Implementation and Application of Functional Languages*. Lecture Notes in Computer Science (cited on page 159).
Voevodsky, Vladimir (2006). 'A Very Short Note on Homotopy λ-Calculus' (cited on page 22).
— (2013). 'HTS - A Simple Type System with Two Identity Types'. [↗] (cited on pages 21, 156).
— (2014). *The Equivalence Axiom and Univalent Models of Type Theory. (Talk at CMU on February 4, 2010)*. arXiv: `1402.5556 [math]`. [↗] (cited on page 21).
Voevodsky, Vladimir, Benedikt Ahrens, Daniel Grayson et al. (n.d.). *UniMath — a Computer-Checked Library of Univalent Mathematics*. available at https://github.com/UniMath/UniMath. [↗] (cited on page 24).
Watkins, Kevin, Iliano Cervesato, Frank Pfenning and David Walker (2003). *A Concurrent Logical Framework I: Judgments and Properties*. Carnegie Mellon University. [↗] (cited on page 156).
Ziliani, Beta, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski and Viktor Vafeiadis (2013). 'Mtac: A Monad for Typed Tactic Programming in Coq'. In: *ACM SIGPLAN Notices* 48.9. [↗] (cited on page 26).
Ziliani, Beta and Matthieu Sozeau (2015). 'A Unification Algorithm for Coq Featuring Universe Polymorphism and Overloading'. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015. [↗] (cited on page 26).