

# Objective Metatheory of Cubical Type Theories

(thesis proposal)

Jonathan Sterling

August 15, 2020

School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA 15213

## **Thesis Committee:**

Robert Harper, Chair

Lars Birkedal

Jeremy Avigad

Karl Craty

Favonia

*Submitted in partial fulfillment of the requirements  
for the degree of Doctor of Philosophy.*

Copyright © 2020 Jonathan Sterling

I gratefully acknowledge the support of the Air Force Office of Scientific Research through MURI grant FA9550-15-1-0053. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the AFOSR.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Computation in dependent type theory . . . . .	1
1.2	Syntactic, phenomenal, and semantic aspects of equality . . . . .	3
1.3	Subjective metatheory: the mathematics of formalisms . . . . .	6
1.4	Objective metatheory: a syntax-invariant perspective . . . . .	6
1.5	Towards principled implementation of proof assistants . . . . .	20
1.6	Thesis Statement . . . . .	20
1.7	Acknowledgments . . . . .	21
<b>2</b>	<b>Background and Prior Work</b>	<b>23</b>
2.1	RedPRL: a program logic for cubical type theory . . . . .	23
2.2	The redtt proof assistant . . . . .	28
2.3	XTT: cubical equality and gluing . . . . .	39
<b>3</b>	<b>Proposed work</b>	<b>45</b>
3.1	Algebraic model theory . . . . .	46
3.2	Contexts and injective renamings . . . . .	46
3.3	Characterizing normal forms . . . . .	48
3.4	Normalization of cartesian cubical type theory . . . . .	49
3.5	redtt reloaded: abstract elaboration . . . . .	50
3.6	Timeline and fallback positions . . . . .	50
<b>A</b>	<b>redtt: supplementary materials</b>	<b>51</b>
A.1	The RTT core language . . . . .	51
A.2	Normalization by evaluation for RTT . . . . .	59
A.3	Elaboration relative to a boundary . . . . .	65
	<b>Bibliography</b>	<b>77</b>

The implementation and semantics of dependent type theories can be studied in a syntax-independent way: the *objective metatheory* of dependent type theories exploits the universal property of their initial models to endow them with computational content, mathematical meaning, and practical implementation (normalization, type checking, elaboration). The semantic methods of the objective metatheory enable the design and implementation of *correct-by-construction* elaboration algorithms, providing a principled interface between real proof assistants and ideal mathematics.

# Chapter 1

## Introduction

The dialectical contrast between presentations of abstract concepts and the abstract concepts themselves, as also the contrast between word problems and groups, polynomial calculations and rings, etc. can be expressed as an explicit construction of a new adjoint functor out of any given adjoint functor. Since in practice many abstract concepts (and algebras) arise by means other than presentations, it is more accurate to apply the term “theory”, not to the presentations as had become traditional in formalist logic, but rather to the more invariant abstract concepts themselves which serve a pivotal role, both in their connection with the syntax of presentations, as well as with the semantics of representations.

F. William Lawvere [Law04]

### 1.1 Computation in dependent type theory

Type theory is a discipline spanning multiple areas of application; in its purest essence, it can be boiled down to the commitment to *substitution-invariant* or *internal* construction, but its utility and implementability are inextricably linked to the type-theoretic facility of computation. Dependent type theory adds to this an emphasis on the individual fibers  $x : A \vdash B(x)$  *type* of a family of types, breaking with the common mathematical practice of considering a family first as a morphism  $B \rightarrow A$ .

Whereas for simple type theory, computation was primarily of *metatheoretic* importance (*e.g.* the execution of already-written programs), computation becomes essential at an intrinsic level for dependent type theory due to the fact one must consider the

identity of each fiber of a family of types. For example, consider the following family of types which can be defined in some versions of Martin-Löf’s type theory:

$$x : \mathbf{bool} \vdash B(x) \stackrel{\text{def}}{=} \mathbf{if } x \mathbf{ then } \mathbb{1} \mathbf{ else } \mathbb{0} \text{ type}$$

If  $\star : \mathbb{1}$  is the unique inhabitant of the unit type, then one expects that the judgment  $\vdash \star : B(\mathbf{true})$  shall hold; indeed, this holds by virtue of the fact that  $B(\mathbf{true})$  is equal to  $\mathbf{if } \mathbf{true} \mathbf{ then } \mathbb{1} \mathbf{ else } \mathbb{0}$  which is equal to  $\mathbb{1}$ . The closure of type inhabitation under equality is called the *conversion rule*, usually written as follows:

$$\begin{array}{c} \text{CONVERSION} \\ \hline \Gamma \vdash A = B \text{ type} \quad \Gamma \vdash M : A \\ \hline \Gamma \vdash M : B \end{array}$$

The ease of use associated with most current implementations of type theory (such as Agda [Nor07], Coq [Coq16], and Lean [Mou+15]) is directly correlated to the ability of a computer to efficiently check the equation  $\Gamma \vdash A = B \text{ type}$ , suggesting that the correct interpretation of the formal type-theoretic equality judgment is that of *calculational equality*, or computations that proceed along High School–style identities (as opposed to full mathematical equality, which can induce obligations of arbitrary quantifier complexity).

In our example of  $\vdash \star : B(\mathbf{true})$ , the conversion rule can be automatically invoked by the computer, which then calculates  $B(\mathbf{true})$  to  $\mathbb{1}$ . Not all computations take the form of reductions, however! Given a family of types  $x : \mathbb{1} \vdash C(x)$  and elements  $\vdash M : \mathbb{1}$  and  $\vdash N : C(\star)$ , we also usually have  $\vdash N : C(M)$ . This is because of the *universal property* of the unit type, which makes  $\vdash M = \star : \mathbb{1}$ , and thence by congruence  $\vdash C(M) = C(\star)$ ; this is of course not a reduction, but it is still a computation.

### 1.1.1 Open computation vs. closed computation


A computational interpretation of closed terms (such as the one provided by Martin-Löf’s meaning explanation of type theory) is essential for establishing the constructivity of a language; it is, however, too weak to fulfill on its own the needs of usable proof assistants, which require a notion of computation that goes quite beyond the extension of the closed term transitions to open terms justified by the naturality of reduction.

Computation with open terms is fundamentally different from computation with closed terms. Open computation must account properly for extensionality principles that are automatic at the level of closed terms, such as  $\eta$ -expansions and commuting conversions; as an example, it is not necessary for *closed* operational semantics to account for any reductions of the following form:

$$(\lambda x : A \times (\mathbb{1} \rightarrow \mathbb{1}).x) \longrightarrow (\lambda x : A \times (\mathbb{1} \rightarrow \mathbb{1}).\langle x, \lambda \_.\star \rangle) \quad (1.1)$$

## 1.2. SYNTACTIC, PHENOMENAL, AND SEMANTIC ASPECTS OF EQUALITY

On the contrary, an operational semantics for open terms *must* derive Reduction 1.1 in one form or another, because the open computation relation is intended to be complete for (typed) judgmental equality, whatever that turns out to be. Open computation construed as the mere extension of closed computation to terms with variables cannot account for Reduction 1.1, and is really only applicable for characterizing the judgmental equality of type theories that lack cartesian products and exponentials, replacing them with weaker versions.

**Example 1.1.1.** The Calculus of Constructions (CoC) is an example of a weak type theory that is not even cartesian closed, and is consequently compatible with a simpler notion of open computation which does not broach the extensionality principles of Reduction 1.1, arising by the extension of ordinary closed computation to a congruence on open terms. However, the main trend in dependent type theory has been toward increasing levels of extensionality and the use of type connectives that have universal properties where possible, as in Nuprl, Epigram, and Agda. 

Nuprl is an early example of a proof assistant that implements proper (universal) versions of all type connectives, including dependent function and sum and even coproduct; in Nuprl it was not possible to capture judgmental equality purely through rules of open computation, but we intend to nurture the idea (first realized in the context of Nuprl) that type connectives should be treated extensionally and universally where possible.

### 1.1.2 Open computation at higher dimension

The requirements of open computation diverge even further from closed computation in the context of *cubical* type theories: in addition to the fact that we must treat Reduction 1.1, there is a further class of (typed) equations that must be rendered computationally, arising from the *boundaries* of the geometric figures treated in cubical type theory (e.g. neutral compositions and coercions, eliminations of path types, etc.).

These “boundary equations” are already present, in a simpler form, in any type theory that has singleton types [SH00; SH06; ACP09], and can be seen to be a form of  $\eta$  law — in the sense that they reflect the inversion of certain introduction rules. In this way,  $\eta$  laws take on a role of extreme importance in cubical type theories, in which their automation is of the essence (see Section 1.2). These matters will be treated in more detail in Sections 2.1 and 2.2.

## 1.2 Syntactic, phenomenal, and semantic aspects of equality

Type theories differ in their treatment of equality, and any specific type theory may express multiple kinds of equality; what *all* kinds of equality share is a way to coerce

	$\alpha$	$\beta$	$\eta^\ominus$	$\eta^\oplus$	function ext.
Nuprl, RedPRL	+T, +A	+T, -A	+T, -A	+T, -A	+T, -A
Coq < 8.5	+T, +A	+T, +A	none	-T, -A	none
Agda, Lean, Coq > 8.5	+T, +A	+T, +A	+T, +A	-T, -A	none
Twelf	+T, +A	+T, +A	+T, +A	(N/A)	(N/A)
Cubical Agda, redtt	+T, +A	+T, +A	+T, +A	-T, -A	-T, -A

Table 1.1: A summary of the syntactic and phenomenal aspects of several classes of equation in different type theoretic formalisms.  $T$  means *tacit* and  $A$  means *automatic*;  $\alpha$  means renaming of variables,  $\beta$  means contractions for elimination forms applied to introduction forms,  $\eta^\ominus$  means uniqueness of introduction forms, and  $\eta^\oplus$  means uniqueness of elimination forms. Cells colored green indicate “extreme sweet spots” in the design space; cells colored red indicate “extreme sour spots” which are known to cause difficulties.

elements from one type to another “equal” type. These coercion mechanisms may differ along a number of distinct (but often conflated) axes:

- 1) *Syntactic* — Coercion along an equation may or may not leave behind a trace in a term. We say that an equation is **tacit** if it leaves no trace in a term.
- 2) *Phenomenal* — Coercion along an equation may or may not require intervention from the user of a proof assistant. We say that an equation is **automatic** if coercing along it requires no intervention. Basic results in computability of course place final limitations on the class of automatable equations.
- 3) *Semantic* — A relation of equality may or may not contain various extensionality and exactness principles: for instance, pairs of functions with equal input-output behavior may or may not be identified, and types whose elements are in bijection may or may not be identified, etc.

Table 1.1 summarizes the syntactic and phenomenal aspects of several important classes of equation in extensional-style and intensional-style formalisms for dependent type theory. The considerations listed above are distinct, but not independent of one another:

- 1) If traces of coercions are left behind in terms (syntactic), then one may ask when two such traces are the same (semantic). Therefore, the apparent disadvantages of non-tacit coercions can always be mitigated by adding further coherence equations, an insight that we have used to great effect in the design of XTT (Section 2.3).
- 2) If some coercions are made tacit (syntactic), many other equations may cease to be automatable (phenomenal). Here are three examples of equations which, if made tacit, will prevent automation of all  $\beta$  and  $\eta$  rules:



## 1.2. SYNTACTIC, PHENOMENAL, AND SEMANTIC ASPECTS OF EQUALITY

- a) Any equation that follows only from an assumption (equality reflection and function extensionality).
- b) Uniqueness of elimination forms for an empty or infinite inductive type.

In either case, it becomes possible to form fixed points whose  $\beta$ -unfoldings may not terminate, scuttling the automation of computational equations.

As such, the syntactic and phenomenal aspects of equality must be investigated simultaneously. Based on the above and the analysis in Table 1.1, we argue that the main problems of the Intensional Type Theory (ITT) formalisms are semantic (things that should be equal in one sense or the other are not equal), whereas the main problems with equality in Extensional Type Theory (ETT) and Nuprl-style formalisms are not semantic, but rather syntactical–phenomenal. In Nuprl, the insistence that coercion along equations following from assumptions be *tacit* (including consequently the uniqueness principles for elimination forms) ensures that there can be no effective strategy to automate  $\beta$  rules, nor the negative fragment of  $\eta$  rules.

On the one hand, type theories like ITT where equality is “semantically wrong” cannot be used directly for formalizing mathematics. One must either extend ITT with axioms (leading to a cluster of semantic and phenomenal defects), or one must *embed* inside ITT a type theory with a corrected notion of equality (as in the setoid construction). While the setoid construction is semantically defensible, the unbelievable difficulty and bureaucracy of *using* setoids makes this a non-starter.

On the other hand, it becomes quickly intractable to *use* a type theory in which basic calculational equalities (like  $\beta$  laws) require user-intervention; difficulties of this kind were recognized from the earliest days of the implementation of type theory, and one of the main motivations of the creators early proof assistants (PL/CV2, PL/CV3,  $\lambda$ -PRL, Nuprl) was to investigate whether the use of *tactical heuristics* from LCF could serve as an effective mitigation [CZ84], a central theme in the next three decades of PRL family research. Years of experience using and implementing both Nuprl-style proof assistants and ITT-style proof assistants have led the author to the following theses:

- 1) The semantic aspects of equality in type theory must be dealt with correctly from the start: embedding an exact completion like setoids into a proof assistant is impractical and reflects design errors in ITT. Cubical type theories, like those implemented in RedPRL, redtt, and Cubical Agda, provide a way forward.
- 2) Tactical heuristics do *not* sufficiently mitigate the failure of automatic  $\beta/\eta$ -conversion in Nuprl-style proof assistants. The AUTO tactics of Nuprl and its descendents are, while ingenious, too brittle and unpredictable.
- 3) Therefore, we *insist* that basic calculational equations must be automatic, and consequently we do not consider tacit coercion along assumed equations (equality reflection). We do *not* insist that all automatic equations must be tacit, a

common commitment which reflects an over-emphasis of syntactic aspects of equality and an under-emphasis of its semantic and phenomenal aspects.

- 4) The role of scientists investigating type theoretic formalisms is to increase over time the class of automatic equations (as in the work of Allais, McBride, and Boutillier [AMB13]), in order to improve the lives of those who learn, teach, and invent mathematics using proof assistants.

### 1.3 Subjective metatheory: the mathematics of formalisms

We promote the term *subjective metatheory* for the traditional study of the mathematical properties of formalisms, as opposed to the study of the type theories they present. The subjective metatheory of a formalism for type theory often involves defining reduction relations on raw terms and establishing the admissibility of various rules or reasoning principles (e.g. cut or substitution, weakening, strengthening, subject reduction, etc.), with an eye toward establishing the following results relative to the presentation:

- 1) *Canonicity*, the existence of canonical representatives of judgmental equivalence classes for closed terms of base type [LH12; Hub18; AHH17].
- 2) *Normalization*, the existence of canonical representatives of judgmental equivalence classes for open terms of arbitrary type [Abe09; Abe13; AAD07; ACP09; AVW17; GSB19b].
- 3) *Sound and complete interpretation*, or the construction of an initial model of type theory by constraining and then quotienting the raw terms of the formalism [Str91].
- 4) *Elaboration*, the translation of a convenient surface language into type theory [Mil+97; HS00; CH09; Bra13; McB99; Bra+11].

Soundness and completeness for a formalism will usually be obtained as a consequence of the normalization theorem, rather than the other way around. The difficulty is to establish the coherence of the interpretation under the very common circumstance that the raw terms of type theory carry less information than the derivations, as explained by Streicher [Str91]. This aspect of the subjective metatheory is simultaneously the most laborious and the least informative; it can be side-stepped entirely in the *objective metatheory* which we detail below.

### 1.4 Objective metatheory: a syntax-invariant perspective

We coin the term *objective metatheory* to refer to the study of *presentation-invariant* structures over type theories, i.e. structures on type theory that are inherited by

all formalisms presenting it. Many aspects of type theory that have historically been studied in a subjective way may also be studied objectively, including canonicity, normalization, conservativity, decidability of equivalence, coherence, and even elaboration.

The objective metatheory distinguishes itself from the subjective not only in its invariance, but also through the emphasis of structure over property. In fact, it is only by passing from property (proof-irrelevant structure) to proper structure (proof-relevant structure) that it becomes possible to entertain invariant accounts of the metatheory of type theory, as we will see in Section 1.4.3.

We are inspired by Lawvere’s use of the prefix “objective” to refer to the study of the laws of reality (including as a special case the deductive processes of “matter-that-thinks”, i.e. humans), in contrast to the pure study of the laws of thought [LS09; Sch00; Law94]; Lawvere’s distinction, applied in the context of categorical algebra and continuum dynamics, draws from Hegel’s opposition of the subjective and the objective [Heg69].

In the context of type theory, the study of deduction in formalisms is subjective in the sense that a formal deduction is a thinking subject’s reflection on a real process — for instance, in a formalism for Euclidean geometry, the deduction of a formal line from a pair of formal points is a subjective reflection of the objective process of drawing this line, which exists regardless of what rules of construction are distinguished as “basic” or “primitive”, and which are distinguished as derived.

The subjective metatheory in logic and type theory corresponds to “choosing a coordinate system” or “choosing a basis” in other areas of mathematics; historically, this has been a necessary step in carrying out calculations of a determinate nature, but recent years have furnished evidence that many calculations (including normalization, the computation of canonical representatives of equivalence classes) can be carried out without choosing any “coordinate system” beforehand [Fio02; Coq19a]. This is what we mean by the *objective metatheory*.

Subjective reflections of real processes may, as a whole, exhibit anomalies and points of tension that mystify and complicate the study of the objects they present. For instance, the bitter controversy between implicit and explicit substitutions in type theory is purely subjective: at the objective level, substitution is characterized by a universal property which can be presented in either an implicit or explicit way without affecting any substantive metatheorem.

### 1.4.1 Lawvere theories and objective algebra

Lawvere laid the groundwork for the *objective metatheory* in his doctoral thesis [Law63] by distinguishing theories from their presentations for the first time, and going on to develop a new kind of categorical model theory called the *functorial semantics*. Aiming to be intelligible to the existing community studying universal algebra, Lawvere restricted his attention to untyped algebraic theories, but his ideas have been extended

in the subsequent years to account for sophisticated binding structures as well as dependent sorts [Car78; PV07; FM10; FH10; Uem19].


**1.4.1.1 Equational presentations and algebras** Originally, an algebraic theory was a pair  $(\mathcal{O}, \mathcal{E})$  in which  $\mathcal{O}$  is a set of operation symbols equipped with arities, and  $\mathcal{E}$  is a set of equational sequents  $\langle n \vdash s \equiv t \rangle$  in which  $s, t$  are *terms* of arity  $n$ ; the terms of arity  $n$  are generated inductively by variables  $x_i$  with  $i < n$  and operations  $o(s_0, \dots, s_{m-1})$  where  $o \in \mathcal{O}_m$  and each  $s_i$  is a term of arity  $n$ .

Following Lawvere [Law63], we refer to the pair  $(\mathcal{O}, \mathcal{E})$  as an *equational presentation* of a theory rather than as a theory. The theory that  $(\mathcal{O}, \mathcal{E})$  presents will be a structure that forgets the difference between generating operations  $o$  and derived terms  $s$ . Each presentation in this sense generates a collection of *algebras*, which are structures equipped with operations corresponding to  $\mathcal{O}$ , obeying the equations in  $\mathcal{E}$ . Between algebras is a class of homomorphisms, giving the collection of algebras the structure of a category.

**Example 1.4.1** (Equational presentation of monoids). The algebraic theory of monoids can be presented by the following operations and equations:


$$\mathcal{O}_n = \begin{cases} \{\varepsilon\} & n = 0 \\ \{\odot\} & n = 2 \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathcal{E} = \left\{ \begin{array}{l} \langle 1 \vdash \odot(\varepsilon, x_0) \equiv x_0 \rangle, \\ \langle 1 \vdash \odot(x_0, \varepsilon) \equiv x_0 \rangle, \\ \langle 3 \vdash \odot(x_0, \odot(x_1, x_2)) \equiv \odot(\odot(x_0, x_1), x_2) \rangle \end{array} \right\}$$

The category of *algebras* for this equational presentation is exactly the category of monoids in the ordinary sense! 

One of the fundamental theorems of modern algebra is that there is a *weak equivalence* (an “isomorphism up to isomorphism”) between the category of equational presentations and the category of theories: a theory can be presented in many different ways, but these presentations are all isomorphic. The minor differences between presentations that are flattened out at the level of theories often constitute a significant source of the difficulty in developing the *subjective* metatheory — located for instance in the very fragile presupposition-admissibility lemmas, etc. Consequently, experienced practitioners of the subjective metatheory have accumulated numerous heuristic design principles for obtaining “good” presentations that evince simpler proofs of normalization, decidability of equivalence, coherence, etc.

**Example 1.4.2** (Cut elimination). Although any two sequent calculus presentations of intuitionistic first-order logic are necessarily isomorphic in a precise sense, it is much easier to decide equality of proofs in a presentation for which the general *cut* and *identity* rules are admissible but not derivable; most structural proof theorists and

type theorists have accordingly developed an attuned awareness of minor differences in presentations that are likely to facilitate or disrupt the admissibility of cut and identity. 

The strength of the *objective metatheory* lies in providing tools that are robust under these differences; these tools include (for example) generalized algebraic theories and logical frameworks [Car78; HHP93; Uem19], as well as Artin gluing [MS93; Cro93; Tay99; Joh02; Fio02; KHS19; SS18]. A weapon more recently added to the arsenal of the objective metatheorist is the homology theory of term rewriting systems [MM16; Ike19], a computational invariant that reflects lower bounds on the *number* of operations and equations required to present a given theory. Thus, the invariance of the objective metatheory does *not* prevent its use for studying non-invariant objects.

**1.4.1.2** *Equational presentations vs. formalisms* According to our terminology, an equational presentation is a specific kind of formalism; other kinds of formalism are possible, and indeed, one of the main aspects of this proposal is to promote formalisms that present a theory by elaboration rather than by equational constraints. The most famous exemplar of this latter kind of formalism is the surface language of Standard ML.

## 1.4.2 Algebraic type theory and logical frameworks

Since the early 1990s, many type theorists have argued that type theories should not be defined first as “hand-crafted” formalisms, but should instead be given as *mathematical objects* in some simpler framework — for instance, Martin-Löf’s Logical Framework (MLLF) [NPS90] or the Edinburgh Logical Framework (ELF) [HHP93], or the (binding-free) discipline of Generalized Algebraic Theories (GATs) [Car86].

Type theorists have preferred logical frameworks for a number of reasons, including the fact that some logical frameworks (mainly MLLF and ELF) enable object-level binders to be represented by a meta-level function space. This facility removes the difficulty and bureaucracy associated with specifying binding structure and substitutions, but it is by no means the only reason to adopt logical frameworks. At the most basic level, logical frameworks free their adopters from the endemic difficulties of the subjective metatheory that pertain to the admissibility of presuppositions and closure under conversion.

Unfortunately, neither the model theory of MLLF nor even that of ELF has ever been satisfactorily worked out: this would entail definitions of categories of models, initial algebras, a functorial semantics in the sense of Lawvere [Law63], as well as an upgrade of the standard Lawvere–Gabriel–Ulmer duality between theories and models. Until this year, these shortcomings have rendered a truly mathematical account of the metatheory of dependent type theories somewhat out of reach.

Luckily, significant progress has been made by researchers in the past year: in particular, Uemura defines a logical framework generalizing MLLF that may be used to

specify almost any (non-modal) type theory, automatically equipped with a functorial semantics and a (bi-)initial model [Uem19].<sup>1</sup> The universal property of the bi-initial  $\mathbb{T}$ -model is the critical ingredient to execute the objective metatheory and obtain canonicity, normalization, etc.; early progress toward the objective metatheory of Martin-Löf Type Theory (MLTT) in terms of Uemura’s framework is made by Sterling and Angiuli [SA20], who have given a general construction of computability models of MLTT.

### 1.4.3 Canonical forms and computability: structure vs property

In the subjective metatheory, the notion of “canonical form” is expressed as a *property* of *raw* terms. For instance, in operational semantics one defines the judgment  $\vdash M$  *value*, and it is clear that this notion of canonical form or value is not invariant under judgmental equality — for if it were, one would expect  $\vdash (\lambda x.x)(5)$  *value*.

The objective counterpart to this notion, which we will continue to refer to as “canonical form”, is to be given as a *structure* over the denotations of *typed* terms in any model  $\mathcal{C}$ ; in the usual case, when  $\mathcal{C}$  is the initial model of the type theory, the notion of canonical form can be seen to be a structure over equivalence classes of typed terms. One ceases to speak of “ $M$  is canonical” (which has no sense if  $M$  is an equivalence class of terms) but rather “ $m$  is a canonical form of  $M$ ”.

Adopting a notion of canonical form that is compatible with abstract terms (we mean, typed terms quotiented up to judgmental equality) confers significant advantages. Many of the highly technical aspects of canonicity, normalization, and parametricity proofs can be boiled down to pushing around proofs of properties that are completely non-trivial for raw terms, but totally automatic for abstract terms. For instance, the burdensome *coherent expansion* conditions endemic to the subjective metatheory of cubical type theory [AHH17; Hub18; Ang19] may be totally dispensed with in the objective metatheory, as illustrated in joint work with Angiuli and Gratzer [SAG19].

**History 1.4.3.** The notion of canonical forms as structures rather than properties goes back to Peter Freyd’s development of *sconing* (gluing along the global sections functor) in 1978, though it can be argued that the structural perspective was latent in the work of Martin-Löf [Mar75a]: already in 1975, Martin-Löf summarizes a computability model as an example, in which a dependent sum is used to express the concept of a term together with a witness of computability.

The “book proof” of canonicity for typed  $\lambda$ -calculus has long been based on structures of canonical forms over equivalence classes of typed terms (see for instance Crole [Cro93]), and this perspective was extended by Fiore [Fio02] to full normalization, providing the objective counterpart to normalization by evaluation.

<sup>1</sup>Bi-initiality is a higher dimensional universal property for the term model which is more suited to categorical manipulations.

While a number of scientists (including Awodey, Spitters, and others) have long promoted the idea of developing a version of the gluing technique that applies to dependent type theories, the technical realization of this idea was greatly delayed by a pervading consensus within the community that partial equivalence relations over raw terms were required; to the author’s knowledge, the first application of gluing to dependent type theory appeared in the paper of Shulman [Shu15], but it was not until the note of Coquand [Coq18] was made public that the technology became generally understood within the community.

Already in 2016, however, Altenkirch and Kaposi [AK16] presented an objective version of normalization by evaluation (without using the language of gluing) for dependent type theory. Since 2018, a number of authors including myself have participated in developing the theory and applications of gluing for dependent type theories with universe hierarchies [Ste18; Coq19a; KHS19; CHS19; SAG19; SA20; SAG20].

#### 1.4.4 Tait’s method, then and now

How do you prove that every term may be equipped with a canonical form? This question is answered by the *method of computability* introduced by Tait [Tai67] and refined by Girard and Martin-Löf; although the technical details have changed greatly over time (even during the writing of this document), the main idea remains unchanged.

The essence of Tait’s computability method is to construct an interpretation of the type theory in which each type is rendered as some kind of predicate on the elements of that type, and each operation must preserve this predicate. Depending on which metatheorem one wishes to prove (e.g. canonicity for closed terms, canonicity for open terms, parametricity, etc.), a number of parameters of the computability model may be tweaked.

*1.4.4.1 The classical method of computability* Classically, computability predicates were construed as families of *propositions* over raw terms (or typed raw terms, not taken up to judgmental equality), indexed in renamings; there is a great deal of technical difficulty in verifying that these objects are well-defined, including:

- 1) For most metatheorems (but not all), one needs all computable elements to be well-typed.
- 2) Computability must be closed under judgmental equality (whatever that ends up being).
- 3) The computability predicate must be closed under (renamings, substitutions, etc.) in a functorial way.

The situation becomes strictly more difficult in the context of dependent type theory with universes: it is usually not possible to prove normalization (or even closed

canonicity) using ordinary computability predicates over raw terms, and so one has typically passed to a more complex argument involving *two* semantic constructions of approximately equal difficulty:<sup>2</sup>

- 1) A computability interpretation, in which one considers “pairs of equal computable elements” rather than computable elements. This interpretation of type theory into partial equivalence relations is enough to establish the *completeness* part of normalization. This stage involves an ingenious but quite subtle fixed point construction about which a number of difficult closure and saturation conditions must be established — one of the main obstacles to a manageable proof of normalization, as can be seen from the technical report of Gratzer, Sterling, and Birkedal [GSB19c].
- 2) A binary logical relation between the computability interpretation and the raw syntax of the formalism, to establish the *soundness* part of normalization. Here one requires an additional sequence of somewhat technical saturation conditions.

Recent examples of proving normalization for dependent type theory with universes using *subjective* computability include Wieczorek and Biernacki [WB18] and Gratzer, Sterling, and Birkedal [GSB19b]. There appears to be only a limited extent by which this style of construction can be simplified or modularized, but we note that Abel, Öhman, and Vezzosi [AÖV17] present some improvements.

**1.4.4.2** *Objective computability, or computability families* All of the difficulties described above may be dealt with simultaneously in the context of the objective metatheory, using a *computability families* technique inspired by Artin gluing; computability families might alternatively be referred to as *proof-relevant computability predicates*. Computability families exhibit a few main differences from earlier approaches:

- 1) Computability is a structure rather than a property: the use of structure rather than property allows us to finally overcome the reliance on raw terms and reduction relations; overcoming reduction will be of great importance in the objective metatheory of cubical type theory, in which the standard *untyped head expansion* property of semantic equality is replaced by a more sophisticated one that involves non-trivial equality and typing conditions.
- 2) Computability is defined relative to an arbitrary model  $\mathcal{C}$ ; in particular,  $\mathcal{C}$  may be the initial model and in this case one may speak of an *equivalence class of typed terms* being computable. One does not consider raw terms at any point in the process.

A consequence of working with abstract terms (equivalence classes of typed terms) is that the computability families construction does in fact give rise to a *model* of

<sup>2</sup>It is worth noting that in the special case of type theories without universes, such as logical frameworks, much simpler and more elegant proofs are available, e.g. Harper and Pfenning [HP05].



the theory; then, the difficulty in establishing the soundness of normalization for dependent type theory can be discharged using basic algebra: namely, the universal property of the *initial* model  $\mathcal{C}$ .

### 1.4.5 Open canonical forms in the objective metatheory

This section introduces a mathematical version, due to Fiore [Fio02], of normalization for typed  $\lambda$ -calculus.

Canonical forms of open terms are generally divided into “neutrals” and “normals”; neutrals are generated by variables and elimination forms (such as application and projection), whereas normals are usually constructor-forms and include neutrals of atomic type. Taking a cue from Gentzen [Gen35a; Gen35b] via Pfenning [Pfe16], we prefer to think refer to these as *left-normal* and *right-normal* forms respectively.

The first step toward objective normalization is to treat these normal forms as a language totally distinct from the underlying type theory: in contrast to accounts based on rewriting or operational semantics, normal forms are *not* mere subset of the “raw terms” of the type theory.<sup>3</sup> It is most appropriate to define the normal forms lying over *equivalence classes* of terms; for instance, we may write  $\boxed{\Gamma \vdash \sigma \text{ right } \rightsquigarrow [s]}$  to refer to the collection of right-normal forms of the *equivalence class*  $[s]$  of the term  $\Gamma \vdash s : \sigma$ , and likewise the same for  $\boxed{\Gamma \vdash \sigma \text{ left } \rightsquigarrow [s]}$ .

For instance, we may define the following (inductive) rules of construction for left and right normal forms respectively:

$$\begin{array}{c}
 \text{var} \\
 \hline
 \dots, x_i : \sigma_i, \dots \vdash \sigma_i \text{ left } \rightsquigarrow [x_i] \\
 \\
 \begin{array}{cc}
 \text{fst} & \text{snd} \\
 \frac{\Gamma \vdash \sigma \times \tau \text{ left } \rightsquigarrow [p]}{\Gamma \vdash \sigma \text{ left } \rightsquigarrow [\pi_1(p)]} & \frac{\Gamma \vdash \sigma \times \tau \text{ left } \rightsquigarrow [p]}{\Gamma \vdash \tau \text{ left } \rightsquigarrow [\pi_2(p)]} \\
 \\
 \text{app} & \text{ret} \\
 \frac{\Gamma \vdash \sigma \Rightarrow \tau \text{ left } \rightsquigarrow [p] \quad \Gamma \vdash \sigma \text{ right } \rightsquigarrow [s]}{\Gamma \vdash \tau \text{ left } \rightsquigarrow [p(s)]} & \frac{\Gamma \vdash \iota \text{ left } \rightsquigarrow [s]}{\Gamma \vdash \iota \text{ right } \rightsquigarrow [s]} \\
 \\
 \text{pair} & \text{abs} \\
 \frac{\Gamma \vdash \sigma \text{ right } \rightsquigarrow [s] \quad \Gamma \vdash \tau \text{ right } \rightsquigarrow [t]}{\Gamma \vdash \sigma \times \tau \text{ right } \rightsquigarrow [\langle s, t \rangle]} & \frac{\Gamma, x : \sigma \vdash \tau \text{ right } \rightsquigarrow [t]}{\Gamma \vdash \sigma \Rightarrow \tau \text{ right } \rightsquigarrow [\lambda x.t]}
 \end{array}
 \end{array}$$


<sup>3</sup>Indeed, we do not even assume any particular representation of raw terms: almost any representation will work, so we do not dwell on it further.

A right-normal form for the equivalence class  $[s]$  of  $s : \sigma$  in this sense is a *derivation* of the judgment  $\Gamma \vdash \sigma \text{ right } \rightsquigarrow [s]$ . Because we have defined normal forms over equivalence classes, any derivation  $m :: \Gamma \vdash \sigma \text{ right } \rightsquigarrow [s]$  is simultaneously a derivation of  $\Gamma \vdash \sigma \text{ right } \rightsquigarrow [(\lambda x.x)(s)]$ , etc. In this sense, right/left normal forms may be seen as abstract representatives of equivalence classes of typed terms.

**1.4.5.1 Computability families for normalization** In what sense are the families of normal forms specified above well-defined? Under what structural rules are they closed? We have been intentionally vague so far, and now is the time to make things precise. Ultimately, we will define a suitable notion of computability family which encompasses the normal forms: that is, the right/left normal forms of type  $\sigma$  will be organized into a computability family over  $\sigma$ .

We begin by observing that normal forms are not *a priori* closed under general substitutions: at the intuitive level, this is because  $x(5)$  may be a normal form, whereas the substitution instance  $[\lambda z.z/x]x(5) = (\lambda z.z)5$  is not a normal form! This observation may be translated into the formal language of left/right normal forms by noting that such a substitution action would apparently require a rule to transform a right-normal form into a left-normal form — essentially, Gentzen’s cut rule, which we have intentionally omitted.

Of course, one may consider a *canonizing* notion of substitution, but the well-definedness of the canonizing substitutions is the normalization theorem we are trying to prove. This is why, regardless of the specifics of technique, we always begin from a notion of normal form whose well-definedness is immediate, but which is closed under only a limited gamut of substitutions; then, the remainder of our work will be to establish the existence of canonizing substitutions. For most type theories, a suitable restricted class of substitutions (conventionally called “renamings”) may be generated by the structural rules of type theory: weakening, exchange, and contraction.<sup>4</sup>

**Remark 1.4.4.** Choosing a suitable class of renamings is a matter of taste, and is not at all essential, once basic requirements have been met. Classically, many authors have arranged the category of renamings to be a pre-order (in which there is at most one “renaming” between any two contexts, usually given by a generalized weakening); aesthetically we prefer a more direct and structural approach in which the category of renamings has the structure of finite products, but our techniques are not sensitive to these decisions. 

In the end, we must specify an admittedly complex notion: we need to define a model of our type theory in which computability predicates are closed only under renamings (to enable the formulation of normal forms), but in which every definable *general* substitution is computable. In the subjective metatheory, this situation has been referred to as *Kripke logical predicates*; in early exemplars of the objective

<sup>4</sup>Sometimes it may be necessary to omit contraction (for instance, when proving normalization for a type theory with exact coproducts or enumeration types [Alt+01]).

metatheory, these were referred to somewhat mysteriously as “Kripke logical predicates of varying arity” [JT93]. Following the path set out by Jung and Tiuryn [JT93] and Fiore [Fio02], we will see that category theory provides a simplifying perspective that may be used to tame the “method” of Kripke logical predicates into a mathematical theory.

**1.4.5.2 Formal contexts and renamings** Let  $\mathcal{C}$  be the syntactic category of our type theory (in this case, the simply typed lambda calculus with a base type  $\iota$ ); then, we may define a category  $\mathcal{R}$  of *formal contexts* and *formal renamings*, which is equipped with an interpretation functor  $\mathcal{R} \xrightarrow{\epsilon} \mathcal{C}$ . The functor  $\mathcal{R} \xrightarrow{\epsilon} \mathcal{C}$  interprets a formal context  $\Gamma = x : A, y : B, \dots$  as the product  $\epsilon(\Gamma) = A \times B \times \dots$ , and a formal renaming  $\rho$  as the corresponding substitution  $\epsilon(\rho)$  in  $\mathcal{C}$ .

**1.4.5.3 A semantic universe for objects that respect renaming** Our computability predicates will ultimately be some kinds of objects that are equipped with a renaming action. This notion may be specified succinctly in terms of *functor categories*, in particular presheaves on  $\mathcal{R}$ . A presheaf  $F$  on  $\mathcal{R}$ , written  $F : \mathbf{Pr}(\mathcal{R})$ , is a functor  $\mathcal{R}^{\text{op}} \xrightarrow{F} \mathbf{Set}$ ; unfolding things, we have for each formal context  $\Gamma$  a set  $F(\Gamma)$  and for each formal substitution  $\Delta \xrightarrow{\rho} \Gamma : \mathcal{R}$ , we have a *contravariant* renaming action  $F(\Gamma) \xrightarrow{\rho^*} F(\Delta)$ .

The simplest example of an object that respects renaming is the collection of terms for each type  $\sigma$ . This defines a presheaf  $\mathbf{Tm}[\sigma] : \mathbf{Pr}(\mathcal{R})$  as follows:

$$\begin{aligned} \mathbf{Tm}[\sigma](\Gamma) &= \mathcal{C}[\epsilon(\Gamma), \sigma] \\ \rho^*(s \in \mathbf{Tm}[\sigma](\Gamma)) &= s \circ \epsilon(\rho) \end{aligned}$$

In fact, these presheaves may be organized into a functor  $\mathcal{C} \xrightarrow{\mathbf{Tm}[-]} \mathbf{Pr}(\mathcal{R})$ ; categorically, this is the *nerve* generated by the functor  $\mathcal{R} \xrightarrow{\epsilon} \mathcal{C}$ , a fundamental construction that lies at the heart of the mathematical reconstruction of Kripke logical predicates. We may now define the category  $\mathcal{G}$  of computability families:

- 1) An object of  $\mathcal{G}$  is a morphism  $E_\sigma \xrightarrow{p_\sigma} \mathbf{Tm}[\sigma]$  in  $\mathbf{Pr}(\mathcal{R})$ ; that is, a *family of presheaves on renamings, indexed in terms of type  $\sigma$* . The fiber of  $p_\sigma$  over a term  $t$  is the set of witnesses that  $t$  is computable.
- 2) A morphism from  $E_\sigma \xrightarrow{p_\sigma} \mathbf{Tm}[\sigma]$  to  $E_\tau \xrightarrow{p_\tau} \mathbf{Tm}[\tau]$  is a commuting square  $p_t$  of the following kind for some  $E_\sigma \xrightarrow{E_t} E_\tau : \mathbf{Pr}(\mathcal{R})$  and  $\sigma \xrightarrow{t} \tau : \mathcal{C}$ :

$$\begin{array}{ccc} E_\sigma & \xrightarrow{E_t} & E_\tau \\ p_\sigma \downarrow & & \downarrow p_\tau \\ \mathbf{Tm}[\sigma] & \xrightarrow{\quad} & \mathbf{Tm}[\tau] \\ & \mathbf{Tm}[t] & \end{array} \quad (1.2)$$

Unraveling this diagram into intuitive language, it is *exactly* the standard Kripke computability clause for substitutions: a witness that  $t$  is computable is equivalently given by the following data (subject to some additional naturality constraints):

For each formal context  $\Gamma : \mathcal{R}$  and term  $\epsilon(\Gamma) \xrightarrow{s} \sigma : \mathcal{C}$  together with a witness  $\alpha$  that  $s$  is computable, a witness  $\beta(s)$  that  $\epsilon(\Gamma) \xrightarrow{t \circ s} \tau$  is computable.

The category  $\mathcal{G}$  so-defined is in fact an instance of the *gluing construction*, standard in category theory since the early 1970s [AGV72; Wra74];  $\mathcal{G}$  is the gluing of  $\mathcal{C}$  along the nerve functor  $\mathcal{C} \xrightarrow{\mathbf{Tm}[-]} \mathbf{Pr}(\mathcal{R})$ , which may be written using Lawvere’s comma construction [Law63] succinctly as  $\mathbf{id}_{\mathbf{Pr}(\mathcal{R})} \downarrow \mathbf{Tm}[-]$ . There is an evident projection functor  $\mathcal{G} \xrightarrow{\mathbf{gl}} \mathcal{C}$  (in fact, a fibration) that projects from each computability family  $E_\sigma \xrightarrow{p_\sigma} \mathbf{Tm}[\sigma]$  the underlying type  $\sigma : \mathcal{C}$ , and from each morphism of computability families  $p_\sigma \rightarrow p_\tau$ , the underlying map  $\sigma \rightarrow \tau$ .

In the language of the gluing fibration, we would re-construct the situation of Diagram 1.2 as in Diagram 1.3 below (pronounced “ $p_t$  lies over  $t$ ”):

$$\begin{array}{ccc}
 p_\sigma & \xrightarrow{p_t} & p_\tau & & \mathcal{G} \\
 \downarrow & & \downarrow & & \downarrow \\
 \sigma & \xrightarrow{t} & \tau & & \mathcal{C}
 \end{array} \tag{1.3}$$

We will write  $\mathbf{gl}[\sigma]$  for the fiber of  $\mathcal{G} \xrightarrow{\mathbf{gl}} \mathcal{C}$  at the type  $\sigma$ , in other words the collection of computability predicates  $E_\sigma \xrightarrow{p_\sigma} \mathbf{Tm}[\sigma]$  whose base is  $\mathbf{Tm}[\sigma]$ . Those who have encountered category theory before will recognize that each fiber  $\mathbf{gl}[\sigma]$  is equivalent to the slice  $\mathbf{Pr}(\mathcal{R})/\mathbf{Tm}[\sigma]$ .

**1.4.5.4 Closure of computability families under connectives** Standard arguments in categorical algebra establishes that each fiber  $\mathbf{gl}[\sigma]$  is cartesian closed (i.e. a model of the typed  $\lambda$ -calculus) and that the reindexings between  $\mathbf{gl}[\sigma]$  and  $\mathbf{gl}[\tau]$  induced by  $\sigma \xrightarrow{t} \tau$  preserve the  $\lambda$ -calculus; therefore  $\mathcal{G}$  is cartesian closed and the functor  $\mathcal{G} \xrightarrow{\mathbf{gl}} \mathcal{C}$  preserves the  $\lambda$ -calculus. These results may be *immediately* used to show a closed canonicity result for the  $\lambda$ -calculus, reconstructing the classical canonicity argument in an arguably more conceptual way. To establish open canonicity (normalization), we will need to work a bit harder.

The simple argument above establishes the existence of suitable computability families for each connective automatically, irrespective of the metatheorem being proved. Traditionally, the actual definitions of the computability predicates of (for example) function types has been viewed by some as the “meat” of the construction (and therefore laboriously reproduced in their specificity with every metatheorem). In

contrast, the categorical viewpoint clarifies that these are uniquely determined by a universal property—therefore, the creative aspect of computability must lie elsewhere.

**1.4.5.5 Normal forms as computability families** We are now ready to reconstruct our naïve notion of objective normal forms, expressed in the judgments  $\Gamma \vdash \sigma \text{ left } \rightsquigarrow [s]$  and  $\Gamma \vdash \sigma \text{ right } \rightsquigarrow [s]$ , as computability families. In particular, we will define for each type  $\sigma$  two computability families  $\mathbf{L}[\sigma], \mathbf{R}[\sigma] : \mathbf{gl}[\sigma]$  of left- and right-normal forms respectively. Additionally, we will define computability families  $\mathbf{V}[\sigma] : \mathbf{gl}[\sigma]$  of *variables*: an element of  $E_{\mathbf{V}[\sigma]}(\Gamma)$  over a term  $\epsilon(\Gamma) \xrightarrow{s} \sigma$  is a witness that  $s$  is equal to a projection  $\pi_i$  from  $\Gamma$  for some  $i$ , i.e. a variable.

We may reconstruct the inductive definition of normal forms in this language by asking for certain commutative squares to exist in  $\mathbf{Pr}(\mathcal{R})$ ; for instance, the normal forms of the first projection and the pairing constructor may be specified in the following somewhat laborious way:

$$\begin{array}{ccc}
 E_{\mathbf{L}[\sigma \times \tau]} & \xrightarrow{\text{fst}} & E_{\mathbf{L}[\sigma]} \\
 \downarrow \mathbf{L}[\sigma \times \tau] & & \downarrow \mathbf{L}[\sigma] \\
 \mathbf{Tm}[\sigma \times \tau] & \xrightarrow{\quad} & \mathbf{Tm}[\sigma] \\
 & \mathbf{Tm}[\pi_1] & 
 \end{array}
 \qquad
 \begin{array}{ccc}
 E_{\mathbf{R}[\sigma]} \times E_{\mathbf{R}[\tau]} & \xrightarrow{\text{pair}} & E_{\mathbf{R}[\sigma \times \tau]} \\
 \downarrow \mathbf{R}[\sigma] \times \mathbf{R}[\tau] & & \downarrow \mathbf{R}[\sigma \times \tau] \\
 \mathbf{Tm}[\sigma] \times \mathbf{Tm}[\tau] & \xrightarrow{\sim} & \mathbf{Tm}[\sigma \times \tau]
 \end{array}$$

Some explanation is in order: in the specifying diagram for `pair`, the downstairs map is a canonical isomorphism which exists because the functor  $\mathcal{C} \xrightarrow{\mathbf{Tm}[-]} \mathbf{Pr}(\mathcal{R})$  commutes with products; this is an immediate consequence of the fact that  $\mathbf{Tm}[-]$  is the composition of two continuous functors.

We can, however, do quite a bit better than the diagrams above (which will not scale to defining normal forms for more complicated languages), by working in the higher-level language of the fibration  $\mathcal{G} \xrightarrow{\mathbf{gl}} \mathcal{C}$  rather than analytically in the language of  $\mathbf{Pr}(\mathcal{R})$ . The diagrams above, rendered in the language of  $\mathbf{gl}$ , are simplified as various purely bureaucratic details may be swept under the rug without any loss of precision:

$$\begin{array}{ccc}
 \mathbf{L}[\sigma \times \tau] & \xrightarrow{\text{fst}} & \mathbf{L}[\sigma] \\
 \downarrow & & \downarrow \\
 \sigma \times \tau & \xrightarrow{\pi_1} & \sigma
 \end{array}
 \qquad
 \begin{array}{ccc}
 \mathbf{R}[\sigma] \times \mathbf{R}[\tau] & \xrightarrow{\text{pair}} & \mathbf{R}[\sigma \times \tau] \\
 \downarrow & & \downarrow \\
 \sigma \times \tau & \xrightarrow{\text{id}_{\sigma \times \tau}} & \sigma \times \tau
 \end{array}$$

The diagrams above are meant to be read as follows: the upstairs map lives in  $\mathcal{G}$  and the downstairs map is the projection of this map under  $\mathbf{gl}$ , i.e. the underlying term in  $\mathcal{C}$ . We freely use the fact that  $\mathbf{gl}$  commutes with products (and in fact, every

connective) in order to situate  $\sigma \times \tau$  lying underneath  $\mathbf{R}[\sigma] \times \mathbf{R}[\tau]$ . Technically, the latter product lies over a type that is only (uniquely) *isomorphic* to  $\sigma \times \tau : \mathcal{C}$ ; this universal property of the cartesian product justifies a notation where these canonical isomorphisms are left implicit.

Because `pair` lies over an identity map in our presentation, we may express it even more succinctly by simply requiring a map  $\mathbf{R}[\sigma] \times \mathbf{R}[\tau] \xrightarrow{\text{pair}} \mathbf{R}[\sigma \times \tau]$  in the fiber  $\text{gl}[\sigma \times \tau]$  without any further conditions; by combining the specifications of `fst`, `snd` into a single normal form that implements a simultaneous split, we may in fact specify all normal forms “vertically” in this sense (namely, as maps in various fibers of the gluing fibration):

$$\begin{array}{ccc} \text{gl}[\sigma \times \tau] \ni \mathbf{L}[\sigma \times \tau] \xrightarrow{\text{split}} \mathbf{L}[\sigma] \times \mathbf{L}[\tau] & \text{gl}[\sigma \times \tau] \ni \mathbf{R}[\sigma] \times \mathbf{R}[\tau] \xrightarrow{\text{pair}} \mathbf{R}[\sigma \times \tau] & \\ \text{gl}[\sigma \Rightarrow \tau] \ni \mathbf{L}[\sigma \Rightarrow \tau] \xrightarrow{\text{app}} \mathbf{R}[\sigma] \Rightarrow \mathbf{L}[\tau] & \text{gl}[\sigma \Rightarrow \tau] \ni \mathbf{V}[\sigma] \Rightarrow \mathbf{R}[\tau] \xrightarrow{\text{abs}} \mathbf{R}[\sigma \Rightarrow \tau] & \\ & \text{gl}[\sigma] \ni \mathbf{V}[\sigma] \xrightarrow{\text{var}} \mathbf{L}[\sigma] & \end{array}$$

Now that we have distilled the notion of normal forms to their purest essence, we may begin to grasp what is *really* going on in the seemingly ad hoc notion of normal form. The rules for each connective of type theory (introduction, elimination, computation, and uniqueness) may be seen as establishing an isomorphism between the collection of elements of some type, and some other more complex collection. For instance, the four rules for the product connective establish an isomorphism between the collection of elements of the type  $\sigma \times \tau$  and the product of the collections of elements of  $\sigma$  and  $\tau$  respectively.

On the other hand, the problem of normalization is to transform the constituents of a language that has a non-trivial equational theory into a language that has a discrete equational theory. One way to achieve this goal is to replace each type with *two* or more “avatars”, carefully arranging the operations of the theory to ensure that any term to which an equation might apply becomes untypeable.

In our new presentation of normal forms, we have taken the two sides of the defining isomorphism for each connective and changed their types so that they may never be composed, thus doing away with the compatibility laws  $(\beta, \eta)$ : because there is no way to compose `pair` with `split`, there is no need for equations governing their composition.

**1.4.5.6 Normalization by evaluation** A normal form of a term  $\sigma \xrightarrow{t} \tau : \mathcal{C}$  should then be a map  $\mathbf{L}[\sigma] \xrightarrow{t} \mathbf{R}[\tau] : \mathcal{G}$  lying over  $t$  in the sense of the following diagram:

$$\begin{array}{ccc} \mathbf{L}[\sigma] & \xrightarrow{t} & \mathbf{R}[\tau] \\ \downarrow & & \downarrow \\ \sigma & \xrightarrow{t} & \tau \end{array} \quad (1.4)$$

To prove the *completeness* of this equation-free representation, which will require at least the existence of a canonizing substitution operation, is then the point of the normalization theorem.

1.4.5.7 *Computability and the yoga of reify–reflect* While the “automatic” computability model of the  $\lambda$ -calculus in  $\mathcal{G}$  suffices to prove closed canonicity, to prove open canonicity (normalization) we must equip the image of the interpretation with a *saturation structure*, familiar from the subjective metatheory of the  $\lambda$ -calculus.

To see why this is the case, consider the situation of a term  $\sigma \xrightarrow{t} \tau : \mathcal{C}$ ; using the interpretation into the computability model, we automatically have computability families  $\llbracket \sigma \rrbracket : \text{gl}[\sigma]$  and  $\llbracket \tau \rrbracket : \text{gl}[\tau]$ , and a map  $\llbracket \sigma \rrbracket \xrightarrow{\llbracket t \rrbracket} \llbracket \tau \rrbracket : \mathcal{G}$  in the following configuration:

$$\begin{array}{ccc}
 \llbracket \sigma \rrbracket & \xrightarrow{\llbracket t \rrbracket} & \llbracket \tau \rrbracket \\
 \downarrow & & \downarrow \\
 \sigma & \xrightarrow{t} & \tau
 \end{array} \tag{1.5}$$

A normal form over  $t$  would, in contrast, be a map  $\mathbf{L}[\sigma] \xrightarrow{t} \mathbf{R}[\tau]$ ; if we had maps  $\mathbf{L}[\sigma] \rightarrow \llbracket \sigma \rrbracket$  and  $\llbracket \tau \rrbracket \rightarrow \mathbf{R}[\tau]$  both lying over the identity function, we would be done:

$$\begin{array}{ccccccc}
 & & & t & & & \\
 & & & \curvearrowright & & & \\
 \mathbf{L}[\sigma] & \xrightarrow{\quad} & \llbracket \sigma \rrbracket & \xrightarrow{\llbracket t \rrbracket} & \llbracket \tau \rrbracket & \xrightarrow{\quad} & \mathbf{R}[\tau] \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 \sigma & \xrightarrow{\text{id}_\sigma} & \sigma & \xrightarrow{t} & \tau & \xrightarrow{\text{id}_\tau} & \tau \\
 & & & \curvearrowleft & & & \\
 & & & t & & & 
 \end{array} \tag{1.6}$$

Of course, just as we may wish to normalize a map  $\sigma \rightarrow \tau$ , we may likewise need to normalize a map  $\tau \rightarrow \sigma$ ; therefore, both  $\llbracket \sigma \rrbracket, \llbracket \tau \rrbracket$  should be made to lie in between the computability families of left- and right-normal forms. We therefore require the following “saturation structure” in the fiber  $\text{gl}[v]$  for every type  $v : \mathcal{C}$ :

$$\text{gl}[v] \ni \mathbf{L}[v] \xrightarrow{\text{reflect}_v} \llbracket v \rrbracket \xrightarrow{\text{reify}_v} \mathbf{R}[v] \tag{1.7}$$

Indeed, these maps correspond exactly to the classical “reify–reflect yoga” of normalization by evaluation. In each case, the definition of these maps is totally mechanical; but it is nonetheless in these definitions that the real essence of normalization algorithms lies, namely the mystery of  $\eta$ -expansion.

## 1.5 Towards principled implementation of proof assistants

Proof assistants comprise many components: concrete syntax, abstract syntax, equivalence checking, type checking, unification, and elaboration (to name a few). While the importance of minimizing the “trusted basis” of a proof assistant has long been emphasized as a way to mitigate purely technical errors in implementation (e.g. a unification module that emits ill-typed unifiers), comparatively little effort has been spent on ensuring that these “trusted core languages” are actually mathematically meaningful.

This question of mathematical meaning is not merely philosophical or hypothetical: interpretation of most type-theoretic formalisms into mathematical models relies on normalization results that have *not* been proved for the core language actually in use.<sup>5</sup> Current tools (rewriting, Kripke logical relations, etc.) render the array of necessary metatheorems nearly intractable for the complex formalisms that underlie many proof assistants used today (including Coq, Lean, Agda, Idris, and `redtt`).

The algebraic approach to type theory and its objective metatheory promise to at least partly resolve these difficulties, by significantly simplifying the verification of standard metatheorems using modern mathematical tools, and by giving rise to a new style of un-annotated formalism which derives its equational theory directly from the annotated internal language via elaboration, in a manner inspired by the elaborative semantics of Standard ML [HS00; LCH07; CH09].

## 1.6 Thesis Statement

The implementation and semantics of dependent type theories can be studied in a syntax-independent way: the *objective metatheory* of dependent type theories exploits the universal property of their initial models to endow them with computational content, mathematical meaning, and practical implementation (normalization, type checking, elaboration). The semantic methods of the objective metatheory enable the design and implementation of *correct-by-construction* elaboration algorithms, providing a principled interface between real proof assistants and ideal mathematics.

To substantiate my thesis statement, I intend to prove normalization and decidability of judgmental equality for Cartesian Cubical Type Theory ( $\text{CuTT}_\times$ ), using

<sup>5</sup>For instance, although an idealized version of Coq’s core language is known to be strongly normalizing, there is no corresponding proof for the core language actually in use; this is not a matter of nit-picking, because the “real Coq” uses typed conversion rules that are not currently compatible with the techniques used to prove normalization for the Predicative Calculus of Inductive Constructions (pCIC).



this result to define an elaborative semantics for a `reddt`-like proof assistant. My plan to accomplish this goal can be found in Chapter 3.

## 1.7 Acknowledgments

In addition to the members of my thesis committee, I especially thank Mathieu Anel, Steve Awodey, Carlo Angiuli, Evan Cavallo, Thierry Coquand, Daniel Gratzer, Favonia, Marcelo Fiore, Jonas Frey, Dan Licata, Anders Mörtberg, Mike Shulman, and Bas Spitters for their helpful advice.



# Chapter 2

## Background and Prior Work

Discover the truth through practice, and again through practice verify and develop the truth. Start from perceptual knowledge and actively develop it into rational knowledge; then start from rational knowledge and actively guide revolutionary practice to change both the subjective and the objective world. Practice, knowledge, again practice, and again knowledge. This form repeats itself in endless cycles, and with each cycle the content of practice and knowledge rises to a higher level. Such is the whole of the dialectical-materialist theory of knowledge, and such is the dialectical-materialist theory of the unity of knowing and doing.

Chairman Mao Zedong [Mao37]

### 2.1 RedPRL: a program logic for cubical type theory

RedPRL [Ang+18b] was our first interactive proof assistant for  $\text{CuTT}_\times$ ; rather than a tool for deriving facts about all models of  $\text{CuTT}_\times$ , RedPRL is a tool optimized for proving theorems about the objects of a specific *computational* semantics for  $\text{CuTT}_\times$ , Computational Higher Type Theory (CHTT) [AHH17; Ang19]. We do not have the space to detail CHTT here, beyond to say that it is a semantics for  $\text{CuTT}_\times$  in which each type is a code for a *cubical behavior*, a partial equivalence relation on *cubical programs* that is closed under coherent reductions in a cubical version of operational semantics.

RedPRL is a *proof refinement logic* inspired by the design of Nuprl, MetaPRL, and JonPRL [Con+86; Hic01; SGR15]. RedPRL was primarily designed and implemented by the author, Favonia, and Carlo Angiuli; in addition, we are grateful to a number of

other people for their contributions, including Eugene Akentyev, Tim Baumann, Evan Cavallo, David Christiansen, Daniel Gratzer, Robert Harper, and James Wilcox.

### 2.1.1 Exact equality via equality reflection

In nearly all modern versions of dependent type theory, including most versions of MLTT, there is a *judgmental* function extensionality principle which says roughly that the following judgments are interderivable:

$$\text{FUNCTION EXTENSIONALITY}$$

$$\frac{\Gamma, x : A \vdash F(x) = G(x) : B}{\Gamma \vdash F = G : A \rightarrow B}$$

On the other hand, this may hold or not hold in an *internal* sense relative to the equality *type*. Internal function extensionality is the principle that the following canonical map is an isomorphism in the type theory:

$$\text{ap} : \text{Eq}_{A \rightarrow B}(F, G) \rightarrow ((x : A) \rightarrow \text{Eq}_B(F(x), G(x)))$$

A particularly simple way to induce internal function extensionality is to provide a rule to identify internal equality with judgmental equality:

$$\text{EQUALITY REFLECTION}$$

$$\frac{\Gamma \vdash \_ : \text{Eq}_A(M, N)}{\Gamma \vdash M = N : A}$$

Proof assistants in the Nuprl tradition employ equality reflection for this purpose (among others). Because equality reflection renders the equational theory of types undecidable, proof refinement logics are organized around constructing well-typedness derivations for raw terms rather than around constructing elements of types abstractly.

### 2.1.2 Decidability, presuppositions, and auxiliary subgoals

Any formal presentation of type theory (including proof refinement logics) expresses some invariants about the terms appearing in judgments. The presuppositions of a form of judgment distinguish the meaningful instances of the raw syntax of judgmental expressions from the “garbage” which is not assigned a meaning.

In a semantic situation, these invariants can be very strong and non-elementary; in a proof assistant, however, the presuppositions must always be decidable, regardless of whether the *evidence* of the judgment is decidable. This is because, no matter how the presuppositions are defined, it is critical that it be possible to determine mechanically whether the statement the user has proposed to prove is actually assigned meaning by the semantics of the proof assistant. In some cases, more statements are assigned

meaning and in others, fewer are assigned meaning: a designer of proof assistants is free to squeeze the balloon of presuppositions in whatever direction is appropriate based on concrete conditions and objectives.


It is customary in both intensional and extensional variants of MLTT that in the judgment  $\Gamma \vdash M : A$ , the context expression  $\Gamma$  is presupposed to be a valid context, and the type expression  $A$  is presupposed to be a valid type in context  $\Gamma$ ; because these judgments are decidable in intensional type theories, these presuppositions can be maintained without modification in proof calculi. In contrast, proof calculi for extensional type theories (including Computational Type Theory (CTT) [Con+86]) must negotiate these presuppositions otherwise. There are two possible approaches:

- 1) One could replace the expressions  $\Gamma, A$  in  $\Gamma \vdash M : A$  with *proofs* of the existence of contexts and types in context; then, the presuppositions are implementable even in the context of equality reflection, because it is always possible by definition to check whether a proof expression is correct. This is roughly the approach used in the Andromeda proof assistant [Bau+16], which implements equality reflection.
- 2) Alternatively, one can weaken the presuppositions to something that *can* be decided; as explained in Section 1.2, this presupposition cannot usually require deciding more than  $\alpha$ -equivalence, and is therefore usually not stronger than requiring that the raw expressions  $\Gamma, A$  are well-scoped. This approach to weakening presuppositions is employed in all exponents of the Nuprl methodology, from Nuprl and MetaPRL to JonPRL, and RedPRL.

**2.1.2.1 Auxiliary subgoals** The weakening of presuppositions employed in Nuprl expresses a number of downsides involving the need to repetitively prove numerous very complicated well-formedness subgoals. These well-formedness subgoals appear roughly because to exhibit a proof of  $\Gamma \gg A$ , one must not only find an “element” of  $A$ , but one must also show that  $A$  is a type in context  $\Gamma$ . For instance, in the introduction rule for the dependent sum, it is actually necessary to prove that the family  $B$  is functional in  $A$ :

$$\begin{array}{l} \text{SUM/RIGHT} \\ \Gamma \gg (x : A) \times B \ni \text{pair}(a, b) \\ \left| \begin{array}{l} \Gamma \gg A \ni a \\ \Gamma \gg [a/x]B \ni b \\ \Gamma, x : A \gg \text{istype}(B) \end{array} \right. \end{array}$$


The situation is in fact similar to the repetitive auxiliary premises appearing in many declarative formalisms for intensional MLTT in order to obtain the *admissibility of presuppositions* prior to establishing the injectivity of type constructors (see Gratzer, Sterling, and Birkedal [GSB19b] for discussion); the situation in Nuprl is different only in that these auxiliary subgoals may be very difficult to prove and are not possible to automate in general (whereas they can always be automated in intensional type theory, and in fact do not appear in the algorithmic presentations).

**Remark 2.1.1** (Caching well-formedness subgoals). Historical efforts to develop reliable caching mechanisms for preventing duplicative well-formedness subgoals have met with limited success, in my view, because the equivalence relation on the labels of nodes in the cache must necessarily be subcomputational (as explained in Section 1.2). My experience using the Nuprl proof assistant suggests that these caching measures are without a doubt better than nothing, and yet brittle enough that it is more of a surprise than an expectation when a simple well-formedness subgoal can in fact be discharged automatically. 

### 2.1.3 Cubical computation and untyped programs

Although it was possible to partly mitigate the disadvantages of weakened presuppositions in Nuprl by taking advantage of untyped computation, we have found that this approach is less effective in the context of cubical type theory.

*2.1.3.1 Failure of ordinary head expansion* The computational semantics of MLTT and CTT are closed under a general head expansion rule, allowing untyped reductions to be performed while proving the well-formedness of a type or element.

**Principle 2.1.2** (Untyped head expansion). *First, if  $A \mapsto^* A'$  and  $A'$  is a type in context  $\Gamma$ , then  $A$  is a type in context  $\Gamma$  and moreover  $A = A'$ . If additionally  $M \mapsto^* M'$  and  $M'$  is an element of type  $A'$  in context  $\Gamma$ , then  $M$  is an element of type  $A$  in context  $\Gamma$  and moreover  $M = M'$ .* 

It is easy to see that this version is too naïve to be employed in a cubical setting, pointing to the need for the generalized head expansion principle defined by Angiuli [Ang19]. The simplest example involves the eliminator for the circle  $S^1$ , a higher inductive type generated by one point constructor  $\text{pt} : S^1$  and one path constructor  $\text{loop}[i] : S^1 [\partial i \rightarrow \text{pt}]$ . If Principle 2.1.2 holds, we can prove that  $\text{tt} = \text{ff} : \text{bool}$ .

*Proof.* Consider the following untyped term:

$$M \stackrel{\text{def}}{=} \begin{array}{l} \mathbf{elim\ loop}[i] \mathbf{into\ } \_.\mathbf{bool\ with} \\ \left| \begin{array}{l} \text{pt} \rightarrow \text{tt} \\ \text{loop}[j] \rightarrow \text{ff} \end{array} \right. \end{array}$$

According to the untyped operational semantics of CHTT, we see that  $M \mapsto^* \text{ff}$ , and therefore by Principle 2.1.2 we have  $M : \text{bool}$  and moreover  $M = \text{ff} : \text{bool}$ . By dimension substitution, we furthermore have  $\langle 0/i \rangle M = \text{ff} : \text{bool}$ . Next, we observe that  $\langle 0/i \rangle M \mapsto^* \text{tt}$ ; this is because  $\langle 0/i \rangle \text{loop}[i] \mapsto^* \text{pt}$ . By Principle 2.1.2 again, we therefore have  $\langle 0/i \rangle M = \text{tt} : \text{bool}$ , and by transitivity we have  $\text{tt} = \text{ff} : \text{bool}$ .  $\square$

In the computational semantics of zero-dimensional type theory, programs containing ill-typed components can themselves be well-typed by virtue of the fact that

these ill-typed components are “dead code”. This fact, embodied in the head expansion principle, enables many well-formedness subgoals to be simplified heuristically by simply “computing”; although this manual use of computational heuristics to discharge subgoals is quite a bit weaker than the typed open computation which can be automatically employed in intensional type theories, it has proved to be nonetheless extremely useful in controlling auxiliary subgoals in Nuprl-style systems.

The reason head expansion is possible in ordinary type theory is that the rules of computation commute with all substitutions. For instance, if  $M \mapsto^* M'$ , then  $[N/x]M \mapsto^* [N/x]M'$ . In contrast, cubical computation does not commute with dimension substitutions, and the cubical *behaviors* are instead required to classify only programs that do in fact evaluate coherently (up to the behavior) with dimension substitution [Ang19].

*2.1.3.2 A cubical generalization of head expansion* There is a generalized, higher-dimensional version of the head expansion principle which is validated by the computational semantics of cubical type theory, exposed and employed by both Angiuli, Hou (Favonia), and Harper [AHH17] and Huber [Hub18]. The advantage of ordinary, untyped head expansion in the context of Nuprl is that it can be used to reduce subgoals without incurring any further typing or functionality obligations; unfortunately, the generalized head expansion of cubical type theory cannot be used to solve the same problem, as it involves non-trivial typing and equational side conditions.

## 2.1.4 The shining path to **redtt**

In the course of the **RedPRL** project, we did not satisfactorily generalize certain untyped aspects of the Nuprl methodology into the higher-dimensional setting, though it is of course possible that some radically different approach could make progress on this problem. Although untyped programs may be used as raw materials for a computational semantics to establish the constructivity of cubical type theory, our results suggest that untyped programs must remain “behind the curtain”:<sup>1</sup> the strictures of coherent higher-dimensional computation (embodied in the generalization of ordinary head expansion to coherent expansion [AHH17]) prevent untyped aspects of cubical type theory from playing the leading role they had previously enjoyed in the context of zero-dimensional type theory and its meaning explanation.

Failing to obtain sufficient leverage from untyped computation, we set out to investigate what forms of computation could be employed in cubical proof assistants to maximize usability. Beginning in January of 2018, I began to revisit the idea of *typed open computation* in the form of a cartesian cubical version of normalization by evaluation and bidirectional elaboration which I had begun to conjecture, and in March I initiated the next stage of the **RedPRL** project, **redtt**. The *types first* ethos which pervades **redtt** turned out to be the key to resolving most of the difficulties

<sup>1</sup>In contrast to zero-dimensional type theory, quite apart from any consideration of whether it is desirable to be able to “see” the untyped terms which encode mathematical structures.

we experienced in making **RedPRL** practical for use in formalizing mathematics and higher-dimensional programs.

## 2.2 The **redtt** proof assistant

**redtt** is an interactive proof assistant for  $\text{CuTT}_\times$ , an infinite-dimensional type theory with support for function extensionality, univalence, and higher inductive types [Ang+19; AHH17; CH19]. **redtt** was primarily designed and implemented by the author and Favonia; additionally, Evan Cavallo contributed many parts of the **redtt** mathematical library; and Carlo Angiuli implemented the **redtt** editing mode for the Vim text editor.

### 2.2.1 Introduction to cubical type theory and **redtt**

Cubical type theories extend MLTT by an interval  $\mathbb{I}$ , together with at least two constants  $0, 1 : \mathbb{I}$ .

*2.2.1.1 Building blocks of cubical type theory* The interval is “abstract” in the sense that there is no form of *case*-statement associated to it; in this way, a map  $i : \mathbb{I} \vdash a(i) : A$  determines not only two elements  $a(0)$  and  $a(1)$ , but also an interior  $a(i)$  for every variable  $i : \mathbb{I}$ . This interior is called a *line*, and is often visualized as follows:

$$i : \mathbb{I} \vdash a(0) \xrightarrow{a(i)} a(1) : A \quad (2.1)$$

**Note 2.2.1** (The interval is not a type). In cubical type theory, the interval is not a type; this is because the *types* will be required to satisfy certain closure conditions which are inconsistent for the interval. Instead, it is a separate syntactic category with its own element classification judgment  $r : \mathbb{I}$  and context extension  $\Gamma, i : \mathbb{I}$ . ☛

In addition to lines of elements, we may also consider lines of types  $i : \mathbb{I} \vdash A(i)$  *type*:

$$i : \mathbb{I} \vdash A(0) \xrightarrow{A(i)} A(1) \text{ type} \quad (2.2)$$

In the spirit of dependent type theory, these two notions naturally give rise to the notion of a line of elements  $i : \mathbb{I} \vdash a(i) : A(i)$  over the line of types  $i : \mathbb{I} \vdash A(i)$  *type*, which we may denote schematically as follows:

$$i : \mathbb{I} \vdash \begin{array}{ccc} a(0) & \xrightarrow{a(i)} & a(1) \\ \downarrow & & \downarrow \\ A(0) & \xrightarrow{A(i)} & A(1) \end{array} \quad (2.3)$$



The notions of a line in a type (Diagram 2.1), a line of types (Diagram 2.2), and a line of elements over a line of types (Diagram 2.3) may be internalized into cubical type theory as types, which we express in *redtt*'s notation below:

```

1 def line-in-type (A : type) : type = dim -> A
2 def line-of-types : type^1 = dim -> type
3 def line-over-line (A : line-of-types) : type = (i : dim) -> A i

```

In the above,  $\text{type}^1$  denotes a universe of types strictly larger than  $\text{type}$ . When an object varies in more than one dimension, it will often be referred to as a *cube*.


**2.2.1.2 Path types and extension (pre)types** In cubical type theory, it is important to consider the lines that have *specific* endpoints; for instance, the concept of a line  $i : \mathbb{1} \vdash a(i)$  such that  $a(0) = x$  and  $a(1) = y$  can be internalized into cubical type theory as a type of *paths* between  $x$  and  $y$ , written  $\text{Path}_A(x, y)$ .

Paths between objects in cubical type theory give the appropriate notion of “mathematical sameness” that is approximated by *equality* in traditional foundations; indeed, for many kinds of objects (such as traditional data-types and functions between data-types), paths can be seen to coincide with traditional equality. The strength of the higher-dimensional perspective reveals itself when objects with more structure come into play, such as the collection of types themselves: for instance, since at least the 1960s it has been clear that the correct notion of sameness between sets is *not* equality, but bijection.

In fact, following Riehl and Shulman [RS17], it is reasonable to internalize more general constraints on the boundaries of higher-dimensional shapes than merely setting the endpoints; the resulting notion is called an *extension type*. An  $n$ -dimensional extension type  $(\vec{i} : \mathbb{1}^n) \rightarrow A(\vec{i})$  [ $\phi(\vec{i}) \triangleright a(\vec{i})$ ] consists of an  $n$ -cube of types  $i_0 : \mathbb{1}, \dots, i_n : \mathbb{1} \vdash A(\vec{i})$  type together with a suitable *subshape*  $\phi(\vec{i})$  of the  $n$ -cube, such that  $a(\vec{i})$  is an element of  $A(\vec{i})$  defined *only* on the subshape  $\phi(\vec{i})$ . The elements of this extension type, then, are the  $n$ -cubes in  $A(\vec{i})$  that agree with  $a(\vec{i})$  on the subshape determined by  $\phi(\vec{i})$ .

The description above requires some unpacking: first of all, what is a subshape and what does it mean to be defined only on a subshape? While the real answer is considerably more technical, it is entirely correct to think of a subshape of the  $n$ -cube as a *predicate* or *constraint* defined on  $n$  variables ranging over the interval  $\mathbb{1}$ . A partial element defined on a subshape  $\phi(\vec{i})$  is then an element which is defined by first assuming that the constraint  $\phi(\vec{i})$  is satisfied.

**Remark 2.2.2.** Not all possible constraints  $\phi(\vec{i})$  lead to an extension *type*: as we remarked in Note 2.2.1, *types* in cubical type theory must be closed under certain operations (called “composition” and “coercion”). One necessary condition is that  $\phi(\vec{i})$  mention no additional variables except  $\vec{i}$ , but other conditions may be required depending on the specifics of the type theory.

However, it is very common that one may wish to consider the extension of a “bad” constraint  $\phi(\vec{i})$ ; for this reason (among others), `redtt` implements a *two-level type theory* [ACK17; AHH18] which supports both types and “pretypes”. Every possible extension expression gives rise to a pretype. 

Some examples will serve to demystify this concept.

**Example 2.2.3** (Paths via extension types). The *boundary*  $\partial(i)$  of the 1-cube  $i : \mathbb{1}$  is the disjunction of  $i = 0$  and  $i = 1$ ; a partial element defined on  $\partial(i)$  is just a pair of elements. The type of paths between two elements of a type may therefore be defined in terms of the extension type constructor:

$$\text{Path}_A(x, y) \triangleq (i : \mathbb{1}) \rightarrow A \left[ \partial(i) \triangleright \begin{cases} x & \text{if } i = 0 \\ y & \text{if } i = 1 \end{cases} \right]$$

In `redtt`’s notation, this concept may be written as follows:

```
1 def path (A : type) (x y : A) : type =
2   [i] A [i=0 -> x | i=1 -> y]
```

In fact, it is possible to be a bit more general and describe the type of paths between elements lying over a path between types:

```
1 def path (A : dim -> type) (x : A 0) (y : A 1) : type =
2   [i] A i [i=0 -> x | i=1 -> y]
```



**Example 2.2.4** (Singletons via extension types). The simplest subshape  $\phi(\vec{i})$  is the one that selects the entire cube, or equivalently, the constantly true constraint. In the formal language of cubical type theory, we may write this constraint as  $0 = 0$ ; then, an element of  $A(\vec{i})$  defined only on  $0 = 0$  is an ordinary element of  $A(\vec{i})$ .

What then are the elements of the extension type  $(\vec{i} : \mathbb{1}^n) \rightarrow A(\vec{i}) [0 = 0 \triangleright a(\vec{i})]$ ? There is exactly one element, because any potential element must agree with  $a(\vec{i})$  everywhere. Therefore, the singleton types of Stone and Harper [SH06] and Abel, Coquand, and Pagano [ACP09] may be reconstructed directly by constraining the 0-cube:

$$\mathcal{S}_A(a) \triangleq (\cdot : \mathbb{1}^0) \rightarrow A [0 = 0 \triangleright a]$$



While our concrete examples of extension types are relatively simple, the ability to encode more sophisticated constraints  $\phi(\vec{i})$  plays a critical role in `redtt`’s reconstruction of proof goals, enabling a significantly smoother interactive proof experience than is currently possible in other implementations of cubical type theory, such as Cubical Agda [VMA19].

**Example 2.2.5** (Function extensionality). Function extensionality, the Tarpeian rock of intensional type theory, obtains a particularly simple account when phrased in terms of the interval and the cubical path types:

```
1 def funext (A : type) (B : A -> type) (f g : (x : A) -> B x) (h : (x : A) -> path (B x) f g)
2   : path ((x : A) -> B x) f g
3   = \i x -> h x i
```



**2.2.1.3 Partial element pretypes** The situation of being defined only on a subshape of a cube may also be internalized into cubical type theory, this time as a *pretype*. *redtt* does not currently have a user-level notation for partial element pretypes, but they play an important role in the reconstruction of proof goals (in conjunction with extension pretypes). In pseudo-*redtt*-code, we may define a pretype of elements defined only on one endpoint of an interval:

```
1 def example (r : dim) (A : type) : type =
2   (r=0) -> A
```

**2.2.1.4 Composition and coercion** Modern accounts of topology and geometry all proceed from the same basic insight: spaces of interest (such as topological spaces, manifolds, schemes, etc.) may be reconstructed from some restricted class of observations; these observations usually take the form of probing the space by figures of a certain shape, often finitary. In geometry, this perspective is referred to as the “functor of points”, whereas in topology one speaks of “singular complexes”. For instance, one may reconstruct a topological space up to a suitably weak notion of equivalence from the collection of all the simplices (triangles of finite dimension) that can be drawn in it; cubical type theory is based on the idea of studying spaces in terms of the collection of *cubes* of finite dimension that may be drawn in them.

In the case of cubes, one begins with a category  $\square$  of cubes of finite dimension; we have used the *cartesian* cubes [AHH17; Ang+19], the free finite product category generated by an interval [Awo15].<sup>2</sup> A *cubical set*, formally defined as a functor  $\square^{\text{op}} \xrightarrow{F} \mathbf{Set}$ , determines at each dimension  $[n]$  a collection of elements called *n-cubes*, subject to suitable restriction maps that allow the projection of an edge from a square, or the extension of a point into a line, etc.

Each cube  $[n] : \square$  may be realized in an obvious way as a space  $\square_n : \mathbf{Top}$  called the “standard topological *n-cube*”, determining a functor  $\square \xrightarrow{\square_\bullet} \mathbf{Top}$ . Therefore, an arbitrary space  $X : \mathbf{Top}$  gives rise to a cubical set  $\mathbf{NX}_\bullet$  called the *singular cubical complex* of  $X$ , defined by probing  $X$  at each dimension  $[n]$  by the standard topological *n-cube*:

$$\mathbf{NX}_\bullet = \mathbf{Top}[\square_\bullet, X]$$

<sup>2</sup>This is the same as to say, the category of contexts and substitutions of the theory with two constants  $0, 1 : \mathbb{1}$ .

Not all cubical sets arise in this way by probing a space, however: some operations which make sense in a space do not always make sense in an arbitrary cubical set. For instance, if we can continuously draw a line  $x \text{---} y$  between two points in a space and we can also draw a line  $y \text{---} z$ , we should be able to draw a line  $x \text{---} z$ . Our logical intuitions also suggest that such an operation would be essential: if “ $x$  is the same as  $y$ ” and “ $y$  is the same as  $z$ ”, surely we must have “ $x$  is the same as  $z$ ”. Moreover, in a space you can turn a line  $x \text{---} y$  around to get a line  $y \text{---} x$ ; an example of a (cartesian) cubical set lacking this property is the interval  $\mathbb{I}_\bullet = \square[\bullet, [1]]$ .

**History 2.2.6.** To identify those cubical sets that are “well-behaved” in the above sense (both logically and geometrically), algebraic topologists have imposed various conditions (often called *Kan conditions*) which translate the decisive aspects of geometrical spaces into the combinatorial language of cubical sets. In the context of *cubical type theory*, an upgrade of these conditions as algebraic structures has been required, pioneered by Bezem, Coquand, and Huber [BCH14], Cohen, Coquand, Huber, and Mörtberg [Coh+17], and Angiuli, Brunerie, Coquand, Hou (Favonia), Harper, and Licata [Ang+19] and further elucidated by Awodey [Awo18a].

In more recent work, Awodey, Cavallo, Coquand, Riehl, and Sattler have discovered a more refined version of these structures which exhibits a Quillen equivalence between a version of cubical type theory and topological spaces, a long sought-after result which essentially concludes the search for a constructive and type-theoretic account of the homotopy types of standard spaces [Awo19; Coq19b; Rie19].  $\heartsuit$

In addition to composing and flipping paths in types, it is also necessary to be able to transport between fibers of dependent types that are connected by paths; for instance, if we have a path  $A \xrightarrow{\beta} B$  between types, and then we have an element  $a : A$ , we should be able to transport  $a$  to  $B$ :

$$\begin{array}{ccc}
 a & \text{-----} & \beta_* a \\
 \downarrow & & \downarrow \\
 A & \xrightarrow{\beta} & B
 \end{array} \tag{2.4}$$

In `redtt`'s version of cubical type theory (cartesian cubical type theory [Ang+19]), primitive operations are provided that may be used to compose, flip, and transport along paths. For instance, we may compose two paths as follows:

```

1 def compose
2   (A : type)
3   (x0 x1 x2 : A)
4   (p : path A x0 x1)
5   (q : path A x1 x2)
6   : path A x0 x2 =
7   \i ->
8   comp 0 1 (p i) [
9   | i=0 -> \_ -> p 0
10  | i=1 -> \j -> q j
11  ]
12 def flip
13   (A : type)
14   (x0 x1 : A)
15   (p : path A x0 x1)
16   : path A x1 x0 =
17   \i ->
18   comp 0 1 (p 0) [
19   | i=0 -> \j -> p j
20   | i=1 -> \_ -> p 0
21  ]

```

In fact, we may simplify the types of these operations considerably by taking advantage of extension types:

```

1 def compose
2   (A : type)
3   (p : dim -> A)
4   (q : [i] A [i=0 -> p 1])
5   : path A (p 0) (q 1) =
6   \i ->
7   comp 0 1 (p i) [
8   | i=0 -> \_ -> p 0
9   | i=1 -> q
10  ]
11 def flip
12   (A : type)
13   (p : dim -> A)
14   : path A (p 1) (p 0) =
15   \i ->
16   comp 0 1 (p 0) [
17   | i=0 -> p
18   | i=1 -> \_ -> p 0
19  ]

```

Geometrically speaking, the `comp` operation of *redtt* obtains the dotted line in the following diagrams:

$$\begin{array}{ccc}
 \begin{array}{c} [i=0] \\ [j=1] \end{array} \begin{array}{c} \text{-----} \\ [j=1] \end{array} \begin{array}{c} [i=1] \\ [j=1] \end{array} & \begin{array}{c} p(0) \text{-----} q(1) \\ | \\ p(0) \end{array} & \begin{array}{c} p(1) \text{-----} p(0) \\ | \\ p(j) \end{array} \\
 [i=0] \begin{array}{c} | \\ | \\ [j=0] \end{array} & \begin{array}{c} | \\ | \\ p(i) \end{array} & \begin{array}{c} | \\ | \\ p(0) \end{array} \\
 \begin{array}{c} [i=0] \\ [j=0] \end{array} \begin{array}{c} \text{-----} \\ [j=0] \end{array} \begin{array}{c} [i=1] \\ [j=0] \end{array} & p(0) \xrightarrow{p(i)} p(1) & p(0) \xrightarrow{p(0)} p(0) \quad (2.5)
 \end{array}$$

*redtt* has a built-in *coercion* operation to transport elements from one side of a line of types to another; in fact, coercion in *redtt* may also be used to transport from a side of a line to its *interior* (and vice versa), enabling us to define the following operation:

```

1 def transport (A : dim -> type) (a0 : A 0) : (a1 : A 1) * path A a0 a1 =
2   (coe 0 1 a0 in A, \i -> coe 0 i a0 in A)

```

**Example 2.2.7.** Putting these constructions together, it becomes possible to define a (weak) version of the classic **J** eliminator, familiar from Martin-Löf’s identity types:

```
1 def J (A : type) (p : dim -> A) (C : [i] A [i=0 -> p 0] -> type) (d : C (\_ -> p 0)) : C p =
2   coe 0 1 d in \i -> C (\j -> comp 0 j (p 0) [i=0 -> \k -> p 0 | i=1 -> p])
```

**2.2.1.5 The univalence principle** `redtt` additionally supports the *univalence* principle of Voevodsky [Voe10], which states roughly that paths between types correspond to equivalences (type-theoretic isomorphisms) between them. Equipping this principle with computational content has been a long struggle, with credit belonging to a number of authors [Coh+17; AHH17; Hub18; Ang+19].

`redtt` implements the univalence principle by adding a primitive higher-dimensional type former, the *V-type* introduced by Angiuli, Hou (Favonia), and Harper [AHH17], which extends a line of types along an equivalence. We begin by defining a few preliminaries:

```
1 def fiber (A B : type) (f : A -> B) (b : B) : type =
2   (a : _) * path _ (f a) b

3 def is-contr (C : type) : type =
4   (c : _) * ((c' : _) -> path C c' c)

5 def is-equiv (A B : type) (f : A -> B) : type =
6   (b : B) -> is-contr (fiber _ _ f b)

7 def equiv (A B : type) : type =
8   (f : A -> B) * is-equiv _ _ f
```

With the above in hand, it is possible to define a path between types from an equivalence, using the primitive *V-type* connective:

```
9 def ua (A B : type) (e : equiv A B) : path^1 type A B =
10  \i -> V i A B e
```

The univalence principle states that the map `ua` is itself an equivalence; `redtt`’s mathematical library contains two different proofs of this fact [Red18c; Red18d].

**2.2.1.6 Higher inductive types and quotients** `redtt` also supports a restricted class of *higher inductive types* based on the schema and operational semantics of Cavallo and Harper [CH19]; higher inductive types are just like inductive types, except that in addition to specifying points in the type, one may also specify *paths* — resolving the long-standing problem of computational quotients in dependent type theory. In `redtt`, the quotient at an arbitrary family type-valued relation is defined as follows:

```

1 data (A : type) (R : A -> A -> type) |- quotient where
2 | point (a : A)
3 | glue (a b : A) (p : R a b) (i : dim) [
4 | i=0 -> point a
5 | i=1 -> point b
6 ]

```

For each  $A, R$ , we then have a type quotient  $A/R$ ; the point constructor takes an element of  $A$  to its “equivalence class”, and the glue constructor can be used to construct a path between any two points related by  $R$ . Of course, this datatype is only a genuine quotient if  $R$  is an equivalence relation.

**2.2.1.7** *What’s the deal with quotients in type theory?* Some dependent type theories have featured “quotient” types, most notably Nuprl’s CTT. We may now illustrate the sense in which the cubical type theoretic quotients significantly improve on previous attempts.

In the broadest sense, a quotient  $A/R$  is an object equipped with a *universal* map  $A \xrightarrow{e} A/R$  in the following sense: if we have  $A \xrightarrow{f} B$  such that  $f(x) = f(y)$  whenever  $R(x, y)$ , then  $A \xrightarrow{f} B$  factors uniquely through  $A \xrightarrow{e} A/R$ . This universal property is rendered type theoretically as an *elimination rule*, and is satisfied by all type theoretic accounts of quotients including those of CTT.

In mathematics, the universal property is usually only half the ordeal: “quotients” in the sense above are perfectly well-defined, but not particularly useful unless they additionally possess the *effectivity* condition, an instance of **descent**. Effectiveness states that the relation  $R$  of a quotient  $A/R$  may be recovered by pullback:

$$\begin{array}{ccc}
 R & \longrightarrow & A \\
 \downarrow & \lrcorner & \downarrow e \\
 A & \xrightarrow{e} & A/R
 \end{array} \tag{2.6}$$

In more type theoretic language, effectiveness is the condition that equal points in the quotient  $A/R$  lie in the relation  $R$ :

```
(x y : A) (p : path (quotient A R) (point x) (point y)) -> R x y
```

The effectivity of quotients of equivalence relations holds in all topoi, but may fail in other settings, notably assemblies and computational type theories including CTT. In assemblies and CTT, it is possible to obtain effectivity by replacing equivalence relations with something one might call “strong equivalence relations”, but this introduces further problems; the “strong” equivalence relations are not the ones arising naturally in mathematical practice. These difficulties are cited in the original Nuprl book [Con+86], and attempts to work around them are described by Nogin [Nog02].

An alternative resolution, adopted in the Lean proof assistant [Mou+15], is to *postulate* “good” quotient types with the appropriate effectiveness axiom. In some sense, this approach has been a great success, enabling users of Lean to formalize ordinary mathematics without fuss. While postulating effective quotients is sound for the intended models (usually topoi, including the topos of sets), the practical and philosophical disadvantages of non-computational postulates are well-known.

The struggle for suitable universes in which to *do* mathematics coincides in essence with the pursuit of *descent*, a kind of compatibility relationship between certain classes of colimits and limits resembling the distributivity of products over sums in a ring [AJ19]. In ordinary mathematics, the vast majority of possible descent properties necessarily fail (with coproducts and quotients being the notable exceptions), whereas higher topoi and their homotopy type theories (including cubical type theory) provide a setting in which all colimits have descent [Rij19; Uni13]. One of the deepest aspects of higher topos theory is that general descent coincides with the existence of enough univalent universes [Ane19].

To develop the foundations of constructive mathematics means to form and test a hypothesis about the nature of “mathematical construction”. A particularly attractive hypothesis, inspired by Martin-Löf’s meaning explanations of type theory [Mar79; Mar84], is that constructive meaning can be founded on partial equivalence relations over untyped operational semantics [All87; Con+86; Har92]. Since at least the late 1980s, however, it has been common knowledge in a different community that this notion of constructivity (at least in its existing form) was inconsistent with good quotients [CFS88].

Unfortunately, prior to the advent of cubical type theory, the only available constructive accounts of good quotients involved working inside of exact completions (like setoids or “ana-functions”), which is disappointingly distant from the beautiful and intuitive view of constructive meaning based on the execution of programs advanced by Martin-Löf [Mar79]. For this reason, we view the combination of computational meaning [AHH17] with effective quotients to be a central advance of cubical type theory.

### 2.2.2 Higher-dimensional interactive proof in `redtt`

There is a great distance between a *type checker* (e.g. [Coq+09; Coh+18; GSB19a]) and a *proof assistant* like `redtt`, Cubical Agda, Coq, Lean, Idris, or Nuprl; proof assistants perform many tasks, such as the implicit resolution of arguments and coercions, as well as the interpretation of proof tactics. The most important aspect of a proof assistant is the facility of *interactive* proof, which is in essence the ability of a user to submit an incomplete proof to the proof assistant and receive useful feedback about how to proceed. This feedback generally includes the type in which one needs to construct an element, as well as the types of all variables in context.

Suppose a user of `redtt` is trying to prove function extensionality. She would begin by declaring the goal, and leaving a *hole* ? in place of a proof:



```

1 def funext
2   (A : type) (B : A -> type) (f g : (x : A) -> B x)
3   (h : (x : A) -> path (B x) (f x) (g x))
4   : path ((x : A) -> B x) f g
5   = ?

```

When the user runs *redtt* on this file, she will receive the following feedback from the proof assistant:<sup>3</sup>

```

?Hole:
A : type
B : A -> type
f,g : (x : A) -> B x
h : (x : A) -> [i] B x [i=0 -> f x | i=1 -> g x]
|- [i] (x : A) -> B x [i=0 -> f | i=1 -> g]

```

The next step is to introduce the extension type, using a  $\lambda$ -abstraction. Our user changes her input to the following:

```

1 def funext
2   (A : type) (B : A -> type) (f g : (x : A) -> B x)
3   (h : (x : A) -> path (B x) (f x) (g x))
4   : path ((x : A) -> B x) f g
5   = \i -> ?

```

It is at this moment that *redtt*'s novel elaboration algorithm begins to shine. Prior to the  $\lambda$ -abstraction, the displayed was an extension type with a boundary constraint  $[i=0 \rightarrow f \mid i=1 \rightarrow g]$ ; the usual typing rule for a  $\lambda$ -abstraction in an extension type renders the checking of this constraint on the body as a side-condition:

$$\frac{\Gamma, i : \mathbb{I} \vdash M : A(i) \quad (\Gamma, i : \mathbb{I}, \phi \vdash M = N : A(i))}{\Gamma \vdash \lambda i.M : (i : \mathbb{I}) \rightarrow A(i) [\phi \triangleright N]}$$

This *synchronous* checking of constraints is of course incompatible with interactive proof: the meaning of a hole `?` is to take the current proof goal, reify it as a top-level axiom, and return that axiom applied to all local variables. In the case of our user, the expression `\i -> ?` would create a hole when the current proof goal is simply  $(x : A) \rightarrow B x$ , leading to a top-level postulate of the following type:

```

my-hole
: (A : ...) (B : ...) (f g : ...) (h : ...) (i : dim)
-> ((x : A) -> B x)

```

<sup>3</sup>For the purpose of these examples, we render the output of *redtt* using the notation of its input language; in reality, *redtt*'s output is currently displayed in the abstract notation of its core language.

Since my-hole  $A\ B\ f\ g\ h\ i$  does *not* satisfy the constraint  $[i=0 \rightarrow f \mid i=1 \rightarrow g]$ , the synchronous elaboration strategy would lead to an error immediately, bringing to a premature end the interactive construction process. In fact, this is how some other cubical proof assistants (such as Cubical Agda) work: the boundary constraint is into a *unification* problem and forgotten from the goal. The disadvantages of the unification-based approach are clear:

- 1) By forgetting the constraint in the goal and externalizing it immediately as a unification problem, the actual goal ceases to be local, and the main advantages of interactive proof are lost. As soon as one descends below a dimension binder, the proof state becomes *contingent* (rendered yellow in Agda) until it is completed by hand, with very little help from the proof “assistant”.<sup>4</sup>
- 2) In practice, boundary constraints *never* lead to solutions of unification problems; this is because a solution to the unification problem cannot be lifted out from under a scope that constrains  $i=0$ , etc.

In `redtt` we have pioneered a completely different approach in which boundary constraints are kept local to the goal as long as possible, independently proposed by McBride in unpublished work. When our user submits the expression  $\backslash i \rightarrow ?$  to `redtt`, the proof assistant instead responds with the following output:

```
?Hole:
A : type
B : A -> type
f, g : (x : A) -> B x
h : (x : A) -> [i] B x [i=0 -> f x | i=1 -> g x]
i : dim
|- (x : A) -> B x
```

with the following faces:

```
i=0 -> f
i=1 -> g
```

In the above, the boundary constraint has been transferred from the type of the goal to *judgment* of the goal; in fact `redtt` always elaborates relative to an extensible context of cubical constraints, which are adjusted as each refinement rule is called. For instance, when our user replaces  $\backslash i \rightarrow ?$  with  $\backslash i\ x \rightarrow ?$ , the following response is issued:

<sup>4</sup>In many cases, it is possible to reconstruct appropriate local states by inspecting the global unification state; recent versions of Cubical Agda feature do exactly this, simulating in many cases the ease of use associated with `redtt`’s boundary-sensitive elaboration algorithm.

```

?Hole:
A : type
B : A -> type
f, g : (x : A) -> B x
h : (x : A) -> [i] B x [i=0 -> f x | i=1 -> g x]
i : dim
x : A
|- B x

```

with the following faces:

```

i=0 -> f x
i=1 -> g x

```

## 2.3 XTT: cubical equality and gluing

XTT is a variant of  $\text{CuTT}_\times$  meant for strict Bishop sets with function extensionality; the interval and the corresponding path structure are used to characterize the *exact equality* of elements of sets rather than identifications in  $\infty$ -groupoids. Elements of the equality types in XTT are definitionally proof irrelevant, leading to a significant reduction in the number of coherence-related subgoals relative to alternative presentations of set-level type theory.

Motivated by the considerations in Section 1.2 and detailed in Section 2.3, XTT treads a middle ground between ETT and the more bureaucratic term calculi for set-level type theory represented by extensions of MLTT with axioms. Like ETT and unlike axiomatic formalisms, XTT has the *canonicity* property: every closed term of boolean type is a constant; unlike ETT and like some axiomatic formalisms, XTT supports a well-behaved notion of open computation which we conjecture can be used to decide judgmental equality in a practical way.

In Sterling, Angiuli, and Gratzer [SAG19], we took the first steps in developing the *objective metatheory* of cubical type theories in the sense of Section 1.4, by presenting the syntax and model theory of XTT simultaneously as a generalized algebraic theory [Car86], and proving a canonicity theorem using a dependent type-theoretic version of Artin gluing, an idea independently proposed by Awodey.

A critical idea in the objective metatheory of XTT (recently popularized by Coquand [Coq19a], and due to Freyd [Fre78]) is to treat canonical forms *not* as a property of untyped raw terms, but as a *structure* lying over equivalence classes of typed terms. This measure significantly simplifies the canonicity argument relative to those of Huber [Hub18] and Angiuli, Hou (Favonia), and Harper [AHH17], and enables us to dispense permanently with partial equivalence relations on raw terms, instead adopting a proof-relevant version of computability predicates which we refer to as *computability families*.

### 2.3.1 Equality types & boundary separation

2.3.1.1 *Equality types from an interval* Equality types in XTT are the standard path types from cubical type theory:

<p style="text-align: center; margin: 0;">FORMATION</p> $\frac{\begin{array}{l} \Gamma, i : \mathbb{I} \vdash A \text{ type} \\ \Gamma \vdash N_0 : [0/i]A \\ \Gamma \vdash N_1 : [1/i]A \end{array}}{\Gamma \vdash \text{Eq}_{i.A}(N_0, N_1) \text{ type}}$	<p style="text-align: center; margin: 0;">INTRODUCTION</p> $\frac{\begin{array}{l} \Gamma, i : \mathbb{I} \vdash M : A \\ \Gamma \vdash [0/i]M = N_0 : [0/i]A \\ \Gamma \vdash [1/i]M = N_1 : [1/i]A \end{array}}{\Gamma \vdash \lambda(i.M) : \text{Eq}_{i.A}(N_0, N_1)}$
<p style="text-align: center; margin: 0;">ELIMINATION</p> $\frac{\Gamma \vdash M : \text{Eq}_{i.A}(N_0, N_1) \quad \Gamma \vdash r : \mathbb{I}}{\begin{array}{l} \Gamma \vdash M(r) : [r/i]A \\ \Gamma \vdash M(0) = N_0 : [0/i]A \\ \Gamma \vdash M(1) = N_1 : [1/i]A \end{array}}$	<p style="text-align: center; margin: 0;">COMPUTATION</p> $\frac{\Gamma, i : \mathbb{I} \vdash M : A \quad \Gamma \vdash r : \mathbb{I}}{\Gamma \vdash (\lambda(i.M))(r) = [r/i]M : [r/i]A}$
<p style="text-align: center; margin: 0;">UNICITY</p> $\frac{\Gamma \vdash M : \text{Eq}_{i.A}(N_0, N_1)}{\Gamma \vdash M = \lambda(i.M(i)) : \text{Eq}_{i.A}(N_0, N_1)}$	

2.3.1.2 *Strict coherence from boundary separation* Because XTT intends to capture the exact equality in sets rather than arbitrary identifications in  $\infty$ -groupoids, one would expect that any two elements  $\text{Eq}_{i.A}(N_0, N_1)$  should be equal in some sense. This is forced in XTT by means of *boundary separation*, a closure condition on the collections of types and elements inspired by Coquand's universes of Bishop sets [Coq17].

<p style="text-align: center; margin: 0;">BSEP/TY</p> $\frac{\begin{array}{l} \Gamma \vdash r : \mathbb{I} \\ \Gamma, r = 0 \vdash A = B \text{ type} \\ \Gamma, r = 1 \vdash A = B \text{ type} \end{array}}{\Gamma \vdash A = B \text{ type}}$	<p style="text-align: center; margin: 0;">BSEP/ELT</p> $\frac{\begin{array}{l} \Gamma \vdash r : \mathbb{I} \\ \Gamma, r = 0 \vdash M = N : A \\ \Gamma, r = 1 \vdash M = N : A \end{array}}{\Gamma \vdash M = N : A}$
--	--

It is easy to see that the boundary separation rules above imply that the equality types are all strictly subsingleton (i.e. they are definitionally proof irrelevant), as in Figure 2.1.

### 2.3.2 First steps in objective metatheory

The prior work presented in Section 2.2 reflects a state of understanding pre-dating my pursuit of the *objective metatheory*, the main theme of the proposed work. XTT represents a first step in developing presentation-independent (objective) methods for studying higher-dimensional type theories.

$$\begin{array}{c}
\frac{\Gamma \vdash P : \text{Eq}_{i.A}(N_0, N_1)}{\Gamma \vdash P(0) = N_0 : [0/i]A} \quad \frac{\Gamma \vdash P : \text{Eq}_{i.A}(N_0, N_1)}{\Gamma \vdash Q(0) = N_0 : [0/i]A} \\
\hline
\frac{\Gamma \vdash P(0) = Q(0) : [0/i]A}{\Gamma, i : \mathbb{1}, i = 0 \vdash P(i) = Q(i) : A} \quad \dots \quad \text{BSEP/ELT} \\
\hline
\frac{\Gamma, i : \mathbb{1} \vdash P(i) = Q(i) : A}{\Gamma \vdash \lambda(i.P(i)) = \lambda(i.Q(i)) : \text{Eq}_{i.A}(N_0, N_1)} \\
\hline
\Gamma \vdash P = Q : \text{Eq}_{i.A}(N_0, N_1)
\end{array}$$

Figure 2.1: A derivation of the judgmental uniqueness of identity proofs from boundary separation.

The (idealized) theory of XTT is stated algebraically by extending the generalized algebraic theory of *natural models* [Awo18b], a categorical reformulation of the notion of categories with families [Dyb96]. Following Awodey [Awo14], we have used an *informal* style of specification in which we directly specify theories using categorical language, rather than manually specifying dozens of operations and generators for a generalized algebraic equational presentation.<sup>5</sup>

XTT is specified by attaching structure to a natural model  $\tilde{\mathcal{U}} \xrightarrow{\varpi} \mathcal{U}$  in  $\mathbf{Pr}(\mathcal{C})$ , where  $\mathcal{C}$  is a category with a terminal object.  $\mathcal{C}$  is then the category of contexts of XTT, and  $\mathcal{U}$  is the collection of types in context, and  $\tilde{\mathcal{U}}$  is the collection of typed elements in context. In order to guarantee that this style of definition in fact gives rise to a type theory with a category of models and an *initial* model (the abstract syntax of XTT), it is necessary to check that every structure imposed can itself be expressed in a generalized algebraic way, or (equivalently) using the language of finite limits.

The use of presheaves and natural transformations ensures that every object under consideration is automatically closed under substitution; and we will see that the internal language of  $\mathbf{Pr}(\mathcal{C})$  will justify a *local-style* presentation of the rules of type theory, rationalizing the style pioneered in Martin-Löf [Mar84].

**2.3.2.1 Context comprehension** As explained by Awodey [Awo18b],  $\varpi$  is required to be a *representable* natural transformation in the sense that any pullback of  $\varpi$  over a representable object is itself representable; unfolding things, this defines a *context comprehension*  $\Gamma.A$  for each type  $\varepsilon \Gamma \xrightarrow{A} \mathcal{U}$ .

**2.3.2.2 Cubical contexts** The contexts of XTT are *cubical* in the sense that they contain both dimension variables and term variables; this is accomplished by fixing a representable object  $\mathbb{1} : \mathbf{Pr}(\mathcal{C})$  together with two constants  $0, 1 : \mathbb{1}$ . We also support extending

<sup>5</sup>In fact, Sterling, Angiuli, and Gratzer [SAG19] did not go far enough in this direction, leading to a bureaucratized presentation; but I feel free to present here the more streamlined version.

the context by the assumption of an *equation* of dimensions,  $\Gamma.r = s$ ; the weakening substitution  $\Gamma.r = s \rightarrow \Gamma$  can be understood more simply in the case of the equation  $i = r$ , where it is simply the substitution  $\langle r/i \rangle$ . We additionally require that the context  $\Gamma.0 = 1 : \mathcal{C}$  be *initial*, in the there is a exactly one substitution  $\Gamma.0 = 1 \rightarrow \Delta$  for each  $\Delta$ .

Finally, we require a *continuity* condition on the natural model  $\tilde{\mathcal{U}} \xrightarrow{\varpi} \mathcal{U}$ , ensuring that the presheaves  $\tilde{\mathcal{U}}, \mathcal{U}$  preserve the terminal object of  $\mathcal{C}^{\text{op}}$ : specifically, we must have  $\tilde{\mathcal{U}}(\Gamma.0 = 1) \cong \mathcal{U}(\Gamma.0 = 1) \cong \{*\}$ . This continuity condition exhibits the “large elimination” of the inconsistent context, yielding a unique type and unique element of that type in inconsistent contexts  $\Gamma.0 = 1$ .<sup>6</sup>

### 2.3.3 Canonicity by gluing & Kripke computability families

We adapted Coquand’s dependently typed version of Kripke computability families [Coq18] to the cubical setting, and proved an objective canonicity theorem for XTT by gluing along a nerve functor from the syntactic category of XTT into a category of cubical sets  $\mathbf{Pr}(\square)$ . The idea of proving canonicity for cubical type theories by gluing along a nerve from a type theory into cubical sets was independently proposed by Awodey as early as 2015, and has more recently been investigated by Awodey and Fiore in the context of a version of type theory with an interval.

**2.3.3.1 The cubical nerve and the gluing construction** The interval object  $\mathbb{1} : \mathcal{C}$  induces by standard yoga a finite product preserving functor  $\square \xrightarrow{i} \mathcal{C}$ , taking each cubical context  $\Psi : \square$  to the XTT context  $i(\Psi) : \mathcal{C}$  containing only the dimension variables  $\Psi$  and no term variables, exposing  $\square$  as a full subcategory of the category of contexts. Just as in Section 1.4.5.3, this subcategory evinces a nerve  $\mathcal{C} \xrightarrow{\mathbf{N}} \mathbf{Pr}(\square)$  by precomposing the corresponding change of base with the Yoneda embedding:

$$\begin{array}{ccc}
 \mathcal{C} & \xrightarrow{\mathfrak{Y}_{\mathcal{C}}} & \mathbf{Pr}(\mathcal{C}) \\
 \searrow \mathcal{N} & & \downarrow \\
 & & \mathbf{Pr}(\square)
 \end{array} \tag{2.7}$$

A suitable category of Kripke computability families is then obtained from the comma construction  $\mathcal{G} \stackrel{\text{def}}{=} \mathbf{Pr}(\square) \downarrow \mathbf{N}$ , which we observe can be equipped with a fibration  $\mathcal{G} \xrightarrow{\pi} \mathcal{C}$ . We will in fact use this category of computability families as the category of contexts for a new *displayed model* of XTT in the sense of Kaposi, Huber, and Sattler [KHS19], summarized as a homomorphism of XTT-algebras  $\mathcal{G} \xrightarrow{\pi} \mathcal{C}$ .

<sup>6</sup>In fact, this continuity condition was incorrectly omitted from Sterling, Angiuli, and Gratzler [SAG19].

We must therefore verify that  $\mathcal{G}$  can be equipped with the structure of an XTT-algebra; this can be carried out systematically by extending a general result of Sterling and Angiuli [SA20], explained by Sterling, Angiuli, and Gratzer [SAG19].

**2.3.3.2 The base type and the canonicity theorem** In order to state the canonicity theorem, we must have a base type whose elements are *observable*; a type of booleans is the simplest and most natural choice. We will choose an appropriate computability family  $\widetilde{\text{bool}} \rightarrow \mathbf{N}(\text{bool})$  in  $\mathcal{G}$  whose fibers contain the observation of which boolean constant they lie over. We simply define  $\widetilde{\text{bool}} \stackrel{\text{def}}{=} \mathbf{2}$ :

$$\begin{array}{ccc}
 \mathbf{1} & \searrow \mathbf{N}(\text{tt}) & \\
 \text{inl} \downarrow & & \\
 \widetilde{\text{bool}} \stackrel{\text{def}}{=} \mathbf{2} & \xrightarrow{\quad \quad \quad} & \mathbf{N}(\text{bool}) \\
 \text{inr} \uparrow & & \\
 \mathbf{1} & \nearrow \mathbf{N}(\text{ff}) & 
 \end{array}$$

It is trivial to check that this structure models the boolean type.

**Theorem 2.3.1 (Canonicity).** *If  $\cdot \vdash M : \text{bool}$ , then either  $M = \text{tt}$  or  $M = \text{ff}$ .*

*Proof.* We have  $\mathbf{1} \xrightarrow{M} \text{bool}$  in  $\mathcal{C}$ ; by the universal property of the initial algebra  $\mathcal{C}$ , we have  $\mathbf{1} \xrightarrow{!_{\mathcal{G}}(M)} \widetilde{\text{bool}}$ ; considering the fibers of  $\widetilde{\text{bool}}$ , we have  $(\pi \circ !_{\mathcal{G}})(M) \in \{\text{tt}, \text{ff}\}$ . But by initiality we have the following triangle in  $\mathbf{Alg}[\mathbf{XTT}]$ :

$$\begin{array}{ccc}
 \mathcal{C} & \xrightarrow{!_{\mathcal{G}}} & \mathcal{G} \\
 \searrow \text{id}_{\mathcal{C}} & & \downarrow \pi \\
 & & \mathcal{C}
 \end{array}$$

□





# Chapter 3

## Proposed work

I propose to develop the objective metatheory of Cartesian Cubical Type Theory ( $\text{CuTT}_\times$ ) in several steps, with an aim to justifying the correctness of proof assistants like `redtt` which are based on cubical normalization by evaluation.

- 1) (Section 3.1) Develop the *algebraic model theory* of  $\text{CuTT}_\times$ : a 2-category of models of  $\text{CuTT}_\times$  which has a bi-initial object (the term model). The universal property of the bi-initial object will be used to prove normalization later on.
- 2) (Section 3.2) Define an appropriate category  $\mathcal{R}$  of formal contexts and formal injective renamings displayed over  $\mathcal{C}$ . It is in the presheaves on  $\mathcal{R}$  that the notion of normal forms will be well-defined, and by gluing along the corresponding nerve  $\mathcal{C} \xrightarrow{N} \mathbf{Pr}(\mathcal{R})$  one obtains a suitable category of computability families  $\mathcal{G} = \mathbf{Pr}(\mathcal{R}) \downarrow N$ .

The definition of  $\mathcal{R}$  is non-trivial in the context of cubical type theories, where one must account for the uniqueness of types and terms in the inconsistent context  $\Gamma.0 = 1$ .

- 3) (Section 3.3) Characterize the left-normal (trad. neutral) and right-normal (trad. normal) forms of types and elements respectively as computability families in  $\mathcal{G}$ .
- 4) (Section 3.4) Construct the *normalization model* of  $\text{CuTT}_\times$  by gluing along a nerve functor from the syntax of the type theory into  $\mathbf{Pr}(\mathcal{R})$ , extending the techniques described in the preceding chapters in a non-trivial way. **A corollary to the construction of this model will be the normalization and decidability of judgmental equality for  $\text{CuTT}_\times$ .**
- 5) (Section 3.5) Finally, I will specify a convenient surface language in the style of `redtt` and give an elaborative semantics in  $\text{CuTT}_\times$ ; by the preceding constructions, this specification in fact evinces an algorithm.


### 3.1 Algebraic model theory

Consolidating the gains initiated by our algebraic model theory for XTT (Section 2.3.2), I will define the model theory of  $\text{CuTT}_\times$  in a “local” style by using the logical framework of a category of  $\mathbf{Pr}(\mathbb{C})$ . This technique bears some superficial resemblance to that of Orton and Pitts [OP16] and Licata, Orton, Pitts, and Spitters [Lic+18], but it is fundamentally different.

The authors of Orton and Pitts [OP16] and Licata, Orton, Pitts, and Spitters [Lic+18] specify *sufficient* conditions on a topos for it to be a model of cubical type theory in a rather unspecified sense, but we note that this internal specification of models does not in any way give rise to a suitable model theory of  $\text{CuTT}_\times$ ; in particular, there is no version of the “syntactic category of cubical type theory” which could serve as an instance of the Orton-Pitts constraints, simply because the syntactic category of any sensible version of cubical type theory will never be a topos. In other words, although one has a *soundness theorem* for the Orton-Pitts semantics, there one does not necessarily have a corresponding *completeness theorem* (though it is possible that completeness for topos models may be obtained as a corollary of my work).

The fact that the syntax of cubical type theory is not a topos is not the only obstacle: the collection of *all* types is represented by a context in the Orton-Pitts style, whereas this is never the case in the syntax of (predicative) type theory. My more refined methodology, which I have begun to explain in Section 2.3.2 and developed in much more detail in Sterling, Angiuli, and Gratzer [SAG20] involves specifying necessary *and* sufficient conditions on a natural model to serve as an algebra for  $\text{CuTT}_\times$  with an infinite hierarchy of universes.

I will use the framework of Uemura [Uem19] to obtain a 2-category of models of  $\text{CuTT}_\times$  which has a bi-initial object, as detailed in joint work with Angiuli and Gratzer [SAG20]; among these models will be the “standard” ones characterized by Licata, Orton, Pitts, and Spitters [Lic+18], but in contrast, the *syntax* of  $\text{CuTT}_\times$  will also be an instance. This non-trivial generalization of the prevailing account of the syntax and semantics of cubical type theory is a necessary first step toward developing the objective metatheory of  $\text{CuTT}_\times$ , including abstract normalization and the decidability of judgmental equivalence.

**Open Question 3.1.1.** *It is likely that any instance of my model theory embeds into a topos using Yoneda; can this embedding be used to prove the completeness of  $\text{CuTT}_\times$  for topos models of the kind considered by Licata, Orton, Pitts, and Spitters [Lic+18]?* 

### 3.2 Contexts and injective renamings

To prove normalization for a type theory, one begins by considering the class of substitutions under which the normal forms will be *a priori* closed; for instance, in Section 1.4.5.1 it was sufficient to consider the class of all substitutions corresponding

to renamings of variables (or, equivalently, the substitutions generated by the  $\lambda$ -calculus where all term constructors are omitted except variables).

The *a priori* closure of normal forms under renamings is then extended to an *a posteriori* closure of normal forms under general substitution as a consequence of a normalization theorem; this closure under all substitutions can be called the “hereditary” or “canonizing” substitution operation [Wat+04].

### 3.2.1 The problem with contraction

In more sophisticated type theories, the appropriate notion of normal form is often not even closed under all renamings. The main culprit is usually the renaming taking two variables  $[x, y]$  to the same variable  $z$ ; this renaming, called a *contraction* or a *diagonal* depending on perspective, does not have an action on the normal forms of (for instance) simple type theory with strict coproducts, and likewise, it fails for the normal forms of  $\text{CuTT}_\times$ .

In  $\text{CuTT}_\times$ , the difficulty is that the diagonal substitution of a single dimension  $k$  for two dimensions  $[i, j]$  does not have an obvious action on the normal forms of any cubical phenomena expressing a diagonal constraint in their boundary, such as Kan operations and elimination forms of extension types. One might imagine that because dimensions form such a restricted sublanguage of cubical type theory, it may be possible to define a naïve hereditary *dimension* substitution operation prior to proving full normalization.

It is easy to see, however, that hereditary substitution for dimensions actually in turn requires full normalization for terms (which we will only obtain as a corollary of the main result). Consider the context

$$\Delta \equiv [p : \text{Ext}(i, j. \mathbb{N} \rightarrow \mathbb{N} \mid i = j \rightarrow \text{fib}), i : \mathbb{N}, j : \mathbb{N}]$$

where  $\text{fib}$  is a closed term representing the Fibonacci sequence. In this context, the application  $p(i, j)(23)$  is a normal form and yet the diagonal instance  $p(k, k)(23)$  must actually be equal to 17711. Therefore, implementing a hereditary substitution operation for dimensions is no easier than proving full normalization.

### 3.2.2 Injective renamings and eliminating dimension constraints

To resolve the problem in Section 3.2.1, we propose to develop a category of contexts and *injective* renamings of variables (generalizing to cubical type theory the construction of Altenkirch, Dybjer, Hofmann, and Scott [Alt+01]). This generalization is highly non-trivial, however, because variables are not the only things appearing in the contexts of  $\text{CuTT}_\times$ : additionally, we have context extension by arbitrary dimension equations  $\Gamma, r = s$ ; these equalizers can be thought of as inducing a restricted form of equality reflection which is “OK” because the theory of dimensions is decidable in every context.

Which of the context morphisms involving  $r = s$  constitute injective renamings? This is quite unclear at first: consider the “harmless” weakening substitution  $\Gamma.r = r \rightarrow \Gamma$  on the one hand, and the “harmful” weakening  $\Gamma.0 = 1 \rightarrow \Gamma$  on the other. We propose to resolve the question definitively by developing a separate representation of contexts to make these equalizers *admissible*.

First, we note that the equalizers  $\Gamma.i = r$  and  $\Gamma.r = r$  can be eliminated, using  $(r/i)^*\Gamma$  and  $\Gamma$  respectively; the false equalizer  $\Gamma.0 = 1$  cannot be eliminated so easily, so we include a special context  $\boxtimes$  representing any context under which  $0 = 1$  is true. It is then easy enough to define an appropriate notion of injective renaming for these “normalized contexts”.

We then obtain the mathematically appropriate notion of injective renamings between *ordinary* contexts by transporting the simpler notion along a weak equivalence between the category of normalized contexts and the category of ordinary contexts. This weak equivalence constitutes, in fact, a *two-dimensional normalization theorem* for contexts with dimension equalizers, in the sense that it exhibits for each context  $\Gamma$  a *different* normalized context  $\mathbf{nf}(\Gamma)$  together with an invertible substitution  $\Gamma \cong \mathbf{nf}(\Gamma)$ .<sup>1</sup> The constructions described in this section have already been initiated by the author in joint work with Carlo Angiuli.

### 3.3 Characterizing normal forms

The next step will be to characterize the left- and right-neutral forms of  $\text{CuTT}_\times$ , adapting the the presentation of Section 1.4.5 to the dependently typed and cubical setting. The main difficulties will arise in the treatment of *binding* in normal forms, considering that in  $\text{CuTT}_\times$  there are bindings of not only term variables, but dimension variables as well as “silent” proofs of dimension equality  $r = s$ .


To make matters more subtle, the restriction to *injective renamings* in the base category (discussed in Section 3.2) makes it impossible to use the presheaf exponential  $\mathbf{Var}[A] \rightarrow \mathbf{Nf}[B]$  to represent binding internally; phrased in the language of logical frameworks [HL07], the naïve higher-order representation would contain exotic “normal forms” and therefore fail to be *adequate*.

The reason the logical framework’s exponential is suitable when considering *all* renamings is that the context extension is a cartesian product on the left (preserved by the Yoneda embedding), and therefore an exponential when transposed to the right. In the case of injective renamings, the context extension is only a *monoidal* product, evincing a substructural (affine) form of variable binding for normal forms *not* embodied in the exponential.

In unpublished joint work with Carlo Angiuli, I have already begun to conjecture appropriate representations of affine binders for term variables, dimension variables,

<sup>1</sup>We note that the use of the word “normalization” refers here not to  $\beta/\eta$ -normalization as in the rest of this chapter, but rather to the elimination of dimension equalizers.


and dimension equations, which can be used to characterize not only the normal forms of  $\text{CuTT}_\times$ , but also the specification of elaboration (Section 3.5).

**Open Question 3.3.1.** *Must the notions of normal form be treated monolithically for a theory, or can the closure of neutral/normal forms under (e.g.) cartesian products or function types or path types be expressed by a universal property? There is a tantalizing similarity between the characterization of left- and right-normal forms for the  $\lambda$ -calculus and the adjunction models of sequent calculus and Call-By-Push-Value [Lev03; CFM16].* 

## 3.4 Normalization of cartesian cubical type theory

The next stage is to combine the results of Sections 3.2 and 3.3 to construct the *Kripke computability families* model of  $\text{CuTT}_\times$ . As in Section 1.4.5.1, we will begin by constructing a nerve from cubical type theory into presheaves on renamings  $\mathcal{C} \xrightarrow{\mathbf{N}} \mathbf{Pr}(\mathcal{R})$ . This nerve in fact lifts to a pseudomorphism of natural models, so it is a good candidate for gluing [KHS19; SA20].

In the context of normalization, one needs not only a Kripke computability family over each context, but also a further *structure* over types which embodies the reflection of neutral forms into computable elements and the reification of computable elements into normal forms; this “reify-reflect yoga” is the objective counterpart of the *saturation conditions* on admissible logical relations in the subjective metatheory of type theory. These operations will in fact abstractly implement the main aspects of the normalization by evaluation algorithm used by `redtt` and other proof assistants for cubical type theories.

**Open Question 3.4.1.** *Because of the need for this additional structure, the normalization of even ordinary dependent type theory is not a corollary of the general results of Kaposi, Huber, and Sattler [KHS19] and Sterling and Angiuli [SA20]; can we obtain a more general result that encompasses normalization in addition to other applications of gluing? This would be something to hope for if Open Question 3.3.1 can be answered positively.* 

The closure of the gluing model of  $\text{CuTT}_\times$  under the Kan operations and univalent universes will reflect the *algorithmic* aspects of the most difficult parts of the implementing univalent type theories, transforming the declarative equations of the theory into effective procedures for calculation. In fact, these parts of the  $\text{CuTT}_\times$ -algebra will resemble certain parts of the `redtt` codebase, and can be “extracted” as algorithms for use in the implementation of future proof assistants.

### 3.5 **redtt** reloaded: abstract elaboration

I propose to give an elaborative semantics to a convenient surface language inspired by **redtt**, which may be operationalized using the results of the preceding sections (primarily normalization). In the design of proof assistants, many aspects are increasingly well-specified (core language, type checking, normalization, etc.) but others remain the domain of hacking: these latter aspects include elaboration and even unification, which are less often treated mathematically.

My goal is to give a *mathematical* definition of the elaboration of **redtt**, in a style which combines the type theoretic rigor of the work of Harper and Stone [HS00] and Lee, Crary, and Harper [LCH07] with the advances of the objective metatheory described in Section 1.4. Working with the RedPRL Development Team (Angiuli, Cavallo, Favonia, Harper, and Licata), I expect to synthesize these theoretical developments into a new version of **redtt** which corrects a number of design mistakes in the current version.

The correct and *efficient* implementation of  $\text{CuTT}_\times$  is in many respects an open problem; for example, all instances of substitution in ordinary MLTT which are needed in type checking and elaboration may be phrased as the evaluation of closures in an environment machine, but we have not yet succeeded in extending this treatment (due to Coquand [Coq96]) to cubical type theory, in which non-local substitutions of dimension variables abound. Therefore, negotiating the execution of dimension substitutions will be an important component of my investigation into the theory and practice of cubical proof assistants.

### 3.6 Timeline and fallback positions

I believe that the proposed work can be completed in the space of a year; a more granular prediction is difficult to make, considering the interdependency of many aspects of the proposed work. There are, however, some clear intermediate results which would constitute a satisfactory contribution on which to base my dissertation, if some parts of the proposed work turns out to be intractable:

- 1) [**Now + 6 months**] A conceptual proof of  $\beta/\eta$ -normalization for MLTT extended by a cartesian interval, the logic of the face lattice, and extension types: this is  $\text{CuTT}_\times$  minus univalent universes and higher inductive types.
- 2) [**Now + 8 months**] A mathematical specification of the elaboration of a convenient **redtt**-style surface language to the core type theory described in 1).

# Appendix A

## reddt: supplementary materials

In this appendix, we present a formalism named RTT which may be thought of as the elaboration target of reddt.

### A.1 The RTT core language

The grammar of the RTT formalism is presented in Figure A.1.

#### A.1.1 Judgments and presuppositions

A judgmental presentation of a type theory begins by defining *pre-judgments* ranging over raw terms, and then distinguishing the proper judgments from among them by specifying appropriate preconditions, called *presuppositions*. There are numerous choices as to how one may interpret the declaration of a formal rule of inference.

Inspired by logical frameworks, we fix a convention in which every rule must contain sufficient premises to guarantee that each pre-judgment mentioned is a judgment. This results in numerous duplicated premises, which we mitigate at the visual level

(contexts)	$\Gamma$	$\cdot \mid \Gamma, i : \mathbb{I} \mid \Gamma, \xi \mid \Gamma, x : A$
(dimensions)	$r, s$	$i \mid \varepsilon$
(endpoints)	$\varepsilon$	$0 \mid 1$
(equations)	$\xi$	$r = s$
(levels)	$\alpha, \beta$	$\bar{n} \mid \omega \mid \star$
(kinds)	$\kappa$	$\text{pre} \mid \text{fib}$
(types)	$A, B$	$\dots$
(terms)	$M, N$	$x \mid \dots$
(case trees)	$S, T$	$\xi \rightarrow N$

Figure A.1: The basic grammar of the RTT core language.

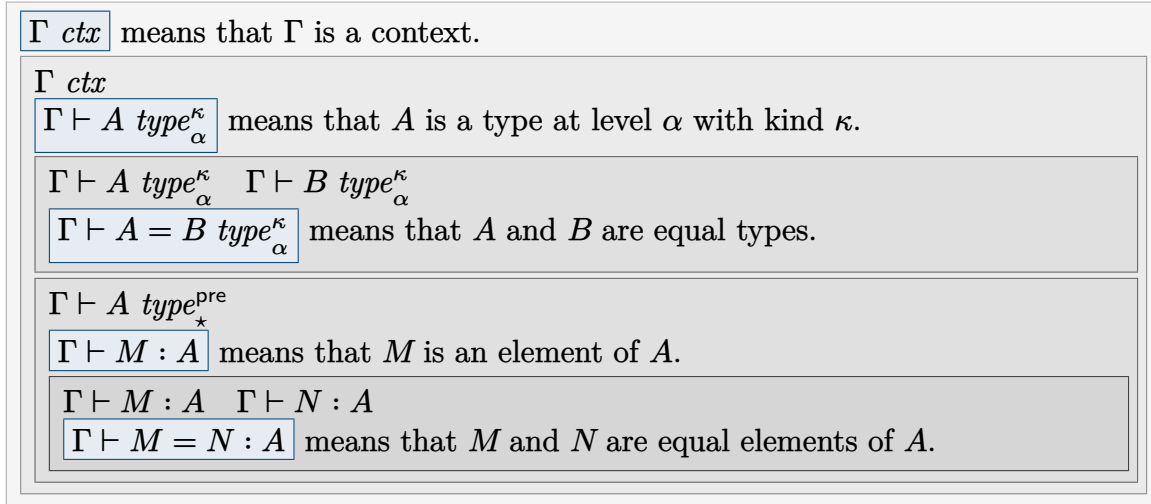


Figure A.2: A schematic presentation of the ordinary forms of judgment of RTT, as a version of dependent type theory with cumulative universes à la Coquand; the cubical extensions to standard dependent type theory are presented in Figure A.3.

by omitting the ones that can be reconstructed by a careful reader, and that we will mitigate at the conceptual level by passing later on to an *algorithmic* calculus in which the mode of inference is fixed qua logic programming.

Accordingly, we present RTT's main forms of judgment schematically in Figures A.2 and A.3; we will also make use of several abbreviations for judgments, summarized in Figure A.4.

### A.1.2 Contexts

Other presentations of  $\text{CuTT}_{\times}$  [Ang+19; Ang19] employ multiple levels of context, each fibered over the previous ones: dimension context, constraint context, and variable context. The split context style has some advantages semantically, but it was simpler to implement RTT with a unified context in the style of Cohen, Coquand, Huber, and Mörtberg [Coh+17].

$$\frac{\text{EMP}}{\cdot \text{ ctx}}$$

There are accordingly three ways to extend the context: with a dimension variable, with a dimension equation, and with a term variable.

$$\frac{\text{EXT/DIM} \quad \Gamma \text{ ctx}}{\Gamma, i : \mathbb{I} \text{ ctx}} \quad \frac{\text{EXT/EQ} \quad \Gamma \text{ ctx} \quad \Gamma \vdash r, r' : \mathbb{I}}{\Gamma, r = r' \text{ ctx}} \quad \frac{\text{EXT/TY} \quad \Gamma \text{ ctx} \quad \Gamma \vdash A \text{ type}_{\star}^{\text{pre}}}{\Gamma, x : A \text{ ctx}}$$



$\Gamma \text{ ctx}$	
$\Gamma \vdash r : \mathbb{N}$	means that $r$ is a dimension.
$\Gamma \vdash \vec{\xi} : \mathbb{F}$	means that $\vec{\xi}$ is a dimension constraint.
$\Gamma \vdash \vec{\xi} : \mathbb{F}$	
$\Gamma \mid \vec{\xi} \vdash \vec{A} \text{ type}_\alpha^\kappa$	means that $\vec{A}$ is a case-tree of types at level $\alpha$ and kind $\kappa$ for $\vec{\xi}$ .
$\Gamma \mid \vec{\xi} \vdash \vec{A} \text{ type}_\star^{\text{pre}}$	
$\Gamma \mid \vec{\xi} \vdash \vec{N} : \vec{A}$	means that $\vec{N}$ is a pointwise case-tree of elements of $\vec{A}$ , i.e. a <i>section</i> of $\vec{A}$ .
$\Gamma \vdash S \text{ type}_\kappa^\alpha$ means that $S$ is a <i>partial type</i> .	
$\Gamma \vdash A \text{ type}_\star^{\text{pre}}$	
$\Gamma \vdash S : A$	means that $S$ is a <i>partial element</i> of $A$ .

Figure A.3: A schematic presentation of the *cubical* extensions to the judgments of type theory in RTT.

$$\begin{array}{c}
\frac{\Gamma \mid \vec{\xi} \vdash B \text{ type}_\alpha^\kappa \quad \Gamma \vdash A \text{ type}_\alpha^\kappa \quad \overline{\Gamma, \xi \vdash A = B \text{ type}_\alpha^\kappa}}{\Gamma \vdash A \text{ type}_\alpha^\kappa \quad [\vec{\xi} \rightarrow \vec{B}]} \\
\frac{\Gamma \vdash A \text{ type}_\star^{\text{pre}} \quad \Gamma \mid \vec{\xi} \vdash \vec{N} : A \quad \Gamma \vdash M : A \quad \overline{\Gamma, \xi \vdash M = N : A}}{\Gamma \vdash M : A \quad [\vec{\xi} \rightarrow \vec{N}]} \\
\frac{\Gamma \vdash S \text{ type}_\star^{\text{pre}} \quad \Gamma \mid \vec{\xi} \vdash \vec{A} \text{ type}_\star^{\text{pre}} \quad \overline{\Gamma, \xi \vdash A \text{ type}_\star^{\text{pre}} [S]}}{\Gamma \mid \vec{\xi} \vdash \vec{A} \text{ type}_\star^{\text{pre}} [S]} \\
\frac{\Gamma \vdash A \text{ type}_\star^{\text{pre}} \quad \Gamma \mid \vec{\xi} \vdash \vec{N} : \overline{A \dots}}{\Gamma \mid \vec{\xi} \vdash \vec{N} : A}
\end{array}$$

Figure A.4: Abbreviations used in RTT's rules of inference, fixing  $\Gamma \vdash \vec{\xi} : \mathbb{F}$ .

<i>(types)</i>	$A, B$	$\dots$	$ $	$M$
<i>(terms)</i>	$M, N$	$\dots$	$ $	$A \mid U_\alpha^\kappa$

$\frac{n < m}{\bar{n} < \bar{m}}$	$\frac{}{\bar{n} < \omega}$	$\frac{}{\omega < \star}$	$\frac{\alpha < \beta \quad \beta \leq \gamma}{\alpha < \gamma}$
FORMATION $\frac{\alpha < \beta}{\Gamma \vdash U_\alpha^\kappa \text{ type}_\beta^\lambda}$	INTRODUCTION $\frac{\Gamma \vdash A \text{ type}_\alpha^\kappa}{\Gamma \vdash A : U_\alpha^\kappa}$	ELIMINATION $\frac{\Gamma \vdash A : U_\alpha^\kappa}{\Gamma \vdash A \text{ type}_\alpha^\kappa}$	

Figure A.5: The grammar and rules for cumulative universes à la Coquand.

In contrast to De Morgan Cubical Type Theory ( $\text{CuTT}_{\text{DM}}$ ) which allows a context to be extended by an arbitrary constraint [Coh+17], we only allow extension by an atomic equation; this corresponds to fixing in the syntax of RTT an  $\eta$ -long normal form for case trees based on a left inversion phase governed by the judgment  $\Gamma \mid \vec{\xi} \vdash \vec{N} : \vec{A}$ .

### A.1.3 Typehood, cumulativity, and universes

The typehood and type equality judgments of RTT are annotated in a universe level and a kind; the levels are the completion of the order of natural numbers with a maximal element  $\star$ , and the kinds are pre (for pretypes) and fib (for fibrant types). RTT expresses two dimensions of cumulativity:

- 1) If  $A$  is a type at level  $n$ , then it is a type at level  $\alpha$  for any  $\alpha \geq n$ .
- 2) If  $A$  is a fibrant type, then it is also a pretype.

RTT employs cumulative universes à la Coquand, which are similar to universes à la Russell, except that one pushes levels into the judgmental structure and then internalizes them in type structure (Figure A.5).

### A.1.4 The interval, constraint satisfaction, and cofibrations

*A.1.4.1 The interval* RTT employs the *cartesian* interval, consisting only of constant dimensions 0, 1 and generic dimensions  $i$  (Figure A.6).

*A.1.4.2 Dimension constraints and cofibrations* In RTT, dimension constraints are just finite disjunctions of equations of dimensions  $r = r'$  (schematically written  $\vec{\xi}$ ); as in Angiuli, Brunerie, Coquand, Hou (Favonia), Harper, and Licata [Ang+19], it is possible to consider richer notions of cofibration (including, for instance, conjunction). The role

$$\begin{array}{ccc}
\text{ZERO} & \text{ONE} & \text{GENERIC} \\
\frac{}{\Gamma \vdash 0 : \mathbb{I}} & \frac{}{\Gamma \vdash 1 : \mathbb{I}} & \frac{\Gamma \ni i : \mathbb{I}}{\Gamma \vdash i : \mathbb{I}}
\end{array}$$

Figure A.6: The cartesian interval of RTT; the judgmental equality of the interval  $\boxed{\Gamma \vdash r = r' : \mathbb{I}}$  is the equivalence closure of the relation  $\{(r, r') \mid \Gamma \ni r = r'\}$ .

$$\begin{array}{ccc}
\boxed{\Gamma \vdash \vec{\xi} : \mathbb{F}} & & \boxed{\Gamma \vdash \vec{\xi} \text{ true}} \\
\text{DISJUNCTION FORMATION} & & \text{DISJUNCTION INTRODUCTION} \\
\frac{\overline{\Gamma \vdash r : \mathbb{I}} \quad \overline{\Gamma \vdash r' : \mathbb{I}}}{\Gamma \vdash r = r' : \mathbb{F}} & & \frac{\Gamma \vdash r_k = r'_k : \mathbb{I}}{\Gamma \vdash r_i = r'_i \text{ true}}
\end{array}$$

Figure A.7: Formation and satisfaction of dimension constraints.

of constraints in cubical type theories is to specify the shape of a partial element of a type (in other words, the part of a cube on which an element is defined). The satisfaction of the disjunction constraint  $\vec{\xi}$  is handled by the judgment  $\boxed{\Gamma \vdash \vec{\xi} \text{ true}}$ , specified in Figure A.7.

### A.1.5 Case trees and partial elements

Given  $\Gamma \text{ ctx}$  and  $\Gamma \vdash \vec{\xi} : \mathbb{F}$ , we express the typing principle for a *case tree of types* through the form of judgment  $\boxed{\Gamma \mid \vec{\xi} \vdash \vec{A} \text{ type}_\alpha^\kappa}$ ; the sequence  $\vec{A}$  determines a type which is partially defined on the subshape determined by  $\vec{\xi}$ :

$$\frac{\Gamma, r = r' \vdash A \text{ type}_\alpha^\kappa \quad \Gamma \mid \vec{\xi} \vdash \vec{A} \text{ type}_\alpha^\kappa [r = r' \rightarrow A]}{\Gamma \mid \cdot \vdash \cdot \text{ type}_\alpha^\kappa}$$

In the above, we have used  $\overline{\Gamma \mid \vec{\xi} \vdash \vec{A} \text{ type}_\alpha^\kappa [S]}$  as an abbreviation for the premises  $\Gamma \mid \vec{\xi} \vdash \vec{A} \text{ type}_\alpha^\kappa$  and  $\overline{\Gamma, \xi \vdash A \text{ type}_\alpha^\kappa [S]}$ . A section of a case tree of types is, then, a case tree of terms:

$$\frac{\Gamma, r = r' \vdash N : A \quad \Gamma \mid \vec{\xi} \vdash \vec{N} : \vec{A} [r = r' \rightarrow N]}{\Gamma \mid \cdot \vdash \dots}$$

As above, we use the notation  $\overline{\Gamma \mid \vec{\xi} \vdash \vec{N} : \vec{A} [S]}$  to abbreviate the premises  $\Gamma \mid \vec{\xi} \vdash \vec{N} : \vec{A}$  and  $\overline{\Gamma, \xi \vdash N : A [S]}$ . As a further notational convenience, certain

operators on terms are extended over case trees in the following way:

$$\begin{aligned}
\text{app}_{x:A.B}(\overrightarrow{\xi \rightarrow M}, N) &\stackrel{\text{def}}{=} \overrightarrow{\xi \rightarrow \text{app}_{x:A.B}(M, N)} \\
\text{app}_{i.A|S}(\overrightarrow{\xi \rightarrow M}, \vec{r}) &\stackrel{\text{def}}{=} \overrightarrow{\xi \rightarrow \text{app}_{i.A|S}(M, \vec{r})} \\
\text{fst}_{x:A.B}(\overrightarrow{\xi \rightarrow M}) &\stackrel{\text{def}}{=} \overrightarrow{\xi \rightarrow \text{fst}_{x:A.B}(M)} \\
\text{snd}_{x:A.B}(\overrightarrow{\xi \rightarrow M}) &\stackrel{\text{def}}{=} \overrightarrow{\xi \rightarrow \text{snd}_{x:A.B}(M)} \\
[i.A] \downarrow_{r'}^r(\overrightarrow{\xi \rightarrow M}) &\stackrel{\text{def}}{=} \overrightarrow{\xi \rightarrow [i.A] \downarrow_{r'}^r M} \\
[i.\xi \rightarrow A] \downarrow_{r'}^r M &\stackrel{\text{def}}{=} \overrightarrow{\xi \rightarrow [A] \downarrow_{r'}^r M} && (i \neq \xi) \\
\lambda i. \overrightarrow{\xi \rightarrow M} &\stackrel{\text{def}}{=} \overrightarrow{\xi \rightarrow \lambda i. M}
\end{aligned}$$

**A.1.5.1 Partial types and partial elements** Presupposing  $\Gamma \text{ ctx}$ , we write  $\boxed{\Gamma \vdash S \text{ type}_\alpha^\kappa}$  to mean that  $S$  is a partial type at level  $\alpha$  of kind  $\kappa$ ; we will write  $\pi(S)$  for the constraint  $\vec{\xi}$  when  $S \equiv \overrightarrow{\xi \rightarrow A}$ . Presupposing  $\Gamma \vdash A \text{ type}_\star^{\text{pre}}$ , we further write  $\boxed{\Gamma \vdash S : A}$  to mean that  $S$  is a partial element of  $A$ .

$$\frac{\Gamma \mid \vec{\xi} \vdash \vec{A} \text{ type}_\alpha^\kappa}{\Gamma \vdash \overrightarrow{\xi \rightarrow \vec{A}} \text{ type}_\alpha^\kappa} \qquad \frac{\Gamma \vdash A \text{ type}_\star^{\text{pre}} \quad \Gamma \mid \vec{\xi} \vdash \vec{N} : A}{\Gamma \vdash \overrightarrow{\xi \rightarrow \vec{N}} : A}$$

We will use the notation  $\Gamma \vdash \overrightarrow{\xi \rightarrow \vec{N}} : A \ [T]$  to abbreviate the premises  $\Gamma \vdash \overrightarrow{\xi \rightarrow \vec{N}} : A$  and  $\Gamma, \xi \vdash N : A \ [T]$ .

## A.1.6 Fibration structure and Kan operations

In cubical type theory, identifications between types are represented as types which depend on the interval  $\mathbb{I}$ ; for instance  $i : \mathbb{I} \vdash A \text{ type}_\star$  is an identification between  $[0/i]A$  and  $[1/i]A$ , and we call  $A$  a *cube* of types.

When a cube of types is equipped with enough structure to extend an element defined on only part of the cube to an element defined on the entire cube, we say that it is *fibrant* or has a *fibration structure*. In cubical type theory, fibration structures are expressed through a ‘‘generalized composition’’ operation which we write as follows:

$$\begin{array}{c}
\text{COMPOSITION FORMATION} \\
\Gamma \vdash r, s : \mathbb{I} \quad \Gamma \vdash \pi(S) : \mathbb{F} \quad \Gamma, i : \mathbb{I} \vdash A \text{ type}_\star^{\text{fib}} \\
\Gamma \vdash M : [r/i]A \quad \Gamma, i : \mathbb{I} \vdash S : A \quad [i = r \rightarrow \vec{M}] \\
\hline
\Gamma \vdash [i.A] \downarrow_s^r [M \mid i.S] : [s/i]A \quad [r = s \rightarrow M, \dots [s/i]S]
\end{array}$$

In practice, it is useful to decompose the composition operation into two different operations (called ‘‘Kan operations’’): coercion and homogeneous composition (Figure A.8) [CH19; CHM18].

$$\begin{array}{c}
\text{COERCION FORMATION} \\
\frac{\Gamma \vdash r, s : \mathbb{1} \quad \Gamma, i : \mathbb{1} \vdash A \text{ type}_\star^{\text{fib}} \quad \Gamma \vdash M : [r/i]A}{\Gamma \vdash [i.A] \downarrow_s^r M : [s/i]A \quad [r = s \rightarrow M]} \\
\\
\text{HOMOGENEOUS COMPOSITION FORMATION} \\
\frac{\Gamma \vdash r, s : \mathbb{1} \quad \Gamma \vdash \pi(S) : \mathbb{F} \quad \Gamma \vdash A \text{ type}_\star^{\text{fib}} \quad \Gamma \vdash M : A \quad \Gamma, i : \mathbb{1} \vdash S : A \quad [i = r \rightarrow \dot{M}]}{\Gamma \vdash A \downarrow_s^r [M | i.S] : A \quad [r = s \rightarrow M, \dots [s/i]S]} \\
\\
[i.A] \downarrow_{r'}^r [M | i.S] \stackrel{\text{def}}{=} [r'/i]A \downarrow_{r'}^r [(i.A) \downarrow_{r'}^r M] | i.[i.A] \downarrow_{r'}^i S
\end{array}$$

Figure A.8: Composition as a combination of coercion and homogeneous composition.

**A.1.6.1 Composite types** The collection of types itself, realized as a universe, must itself be fibrant; Angiuli, Hou (Favonia), and Harper [AHH17] showed how to achieve this by adding *formal composites* of types to the type theory. We summarize the basic rules for composite types in Figure A.9.

## A.1.7 Path types, line types and extension types

Most cubical type theories have a dependent path type connective  $\text{path}_{i.A}(N_0, N_1)$  as a primitive; given a type  $\Gamma, i : \mathbb{1} \vdash A \text{ type}_\star^{\kappa}$  and two elements at either side  $\overrightarrow{\Gamma \vdash N_\varepsilon : [\varepsilon/i]A}$ , the path type  $\text{path}_{i.A}(N_0, N_1)$  contains elements  $\lambda i.M$  such that  $\Gamma, i : \mathbb{1} \vdash M : A$  and  $\overrightarrow{[\varepsilon/i]M = N_\varepsilon}$ . Semantically, one can consider this to be the restriction of a line in  $A$  to a partial element defined on the boundary of  $i : \mathbb{1}$ . In practice, however, we have found that using only the path type to internalize higher-dimensional elements has its limitations: it is generally useful to be able to *partially* constrain an arbitrary  $n$ -cube (for instance, by constraining only one endpoint, or the diagonal of a cube).

A very common scenario is that one needs a type of  $n$ -cubes with *no* additional constraints at all. For this reason, we first introduced *line types* in RedPRL [Red18a], which correspond semantically to dependent products over the interval  $(i : \mathbb{1}) \rightarrow A$ ; RedPRL's line types were later adopted in Cubical Agda. Riehl and Shulman [RS17] have introduced *extension types*, which generalize all the use-cases considered above, expressing arbitrary constraints on  $n$ -cubes. `redtt` is the first proof assistant to implement extension types, which we detail in RTT below.

**A.1.7.1 Extension pretypes** Given an  $n$ -cube of pretypes  $\Gamma, \vec{i} : \vec{\mathbb{1}} \vdash A \text{ type}_\alpha^{\text{pre}}$  and a partial  $n$ -cube  $\Gamma, \vec{i} : \vec{\mathbb{1}} \vdash S : A$ , we have an *extension pretype*  $\Gamma \vdash \text{Ext}(\vec{i}.A | S) \text{ type}_\alpha^{\text{pre}}$  which classifies the  $n$ -cubes in  $A$  which restrict to the partial element  $S$ . An element of  $\text{Ext}(\vec{i}.A | S)$  is given by a lambda abstraction  $\lambda \vec{i}.M$ , where  $M$  is required to agree

(terms)  $M, N \quad \dots \mid [M \mid T] \mid \text{cap}_{\text{U}_\alpha^{\text{fib}} \downarrow_{r'} [A \mid i.S]} (M)$

INTRODUCTION

$$\frac{\Gamma \mid \vec{\xi} \vdash \vec{N} : \overrightarrow{[r'/i]B} \quad \Gamma \vdash M : A \quad \overrightarrow{[\xi \rightarrow [i.B] \downarrow_{r'} N]}}{\Gamma \vdash [M \mid \vec{\xi} \rightarrow \vec{N}] : \text{U}_\alpha^{\text{fib}} \downarrow_{r'} [A \mid i.\vec{\xi} \rightarrow \vec{B}] \quad [r = r' \rightarrow M, \xi \rightarrow \vec{N}]}$$

ELIMINATION

$$\frac{\Gamma \vdash M : \text{U}_\alpha^{\text{fib}} \downarrow_{r'} [A \mid i.S]}{\Gamma \vdash \text{cap}_{\text{U}_\alpha^{\text{fib}} \downarrow_{r'} [A \mid i.S]} (M) : A \quad [r = r' \rightarrow M, \dots [i.S] \downarrow_{r'} M]}$$

UNICITY


$$\frac{S \stackrel{\text{def}}{=} \vec{\xi} \rightarrow \vec{B} \quad \Gamma \vdash M : \text{U}_\alpha^{\text{fib}} \downarrow_{r'} [A \mid i.S]}{\Gamma \vdash M = [\text{cap}_{\text{U}_\alpha^{\text{fib}} \downarrow_{r'} [A \mid i.S]} (M) \mid \vec{\xi} \rightarrow \vec{B}] : \text{U}_\alpha^{\text{fib}} \downarrow_{r'} [A \mid i.S]}$$

COMPUTATION

$$\frac{S \stackrel{\text{def}}{=} \vec{\xi} \rightarrow \vec{B}}{\Gamma \vdash \text{cap}_{\text{U}_\alpha^{\text{fib}} \downarrow_{r'} [A \mid i.S]} ([M \mid \vec{\xi} \rightarrow \vec{B}]) = M : A}$$

Figure A.9: The rules for formal composite types; we omit the calculations of the Kan operations for these composite types, which are computed in detail by Angiuli [Ang19].

with the partial element  $S$ ; elements are eliminated by means of an application expression  $\text{app}_{i.A \mid S}(M, \vec{r})$  given a sequence of dimensions  $\vec{r}$  of the proper length, and the application agrees with the partial element  $[\vec{r}/i]S$  to judgmental equality.

**Example A.1.1** (Cube types). We will write  $\text{Ext}(\vec{i}.A)$  for the unconstrained extension type  $\text{Ext}(\vec{i}.A \mid \cdot)$ , which we call a *cube type*; the cube types correspond to iterations of **RedPRL**'s line types. 

**Example A.1.2** (Dependent path types). The dependent path types of  $\text{CuTT}_\times$  are an instance of the extension type connective, considering the boundary of a single dimension:

$$\text{path}_{i.A}(N_0, N_1) \stackrel{\text{def}}{=} \text{Ext}(i.A \mid i = 0 \rightarrow N_0, i = 1 \rightarrow N_1) \quad \img alt="heart icon" data-bbox="838 791 853 806"/>$$

**A.1.7.2** *Fibrant extension types* Given a (fibrant) type  $\Gamma, i : \mathbb{1} \vdash A \text{ type}_\alpha^{\text{fib}}$ , the pretype  $\text{Ext}(\vec{i}.A \mid S)$  described above cannot always be equipped with a coercion and composition operation. The coercion structure for  $\text{Ext}(\vec{i}.A \mid S)$  can only generally be obtained if the supporting cofibration  $\vec{\xi}$  depends only on the variables  $i : \mathbb{1}$  and not on

$\Gamma$ . We thank Carlo Angiuli for furnishing the following counterexample, which would exhibit a path between `tt` and `ff` in the booleans if  $\text{Ext}(\cdot.\text{bool} \mid j = 0 \rightarrow \text{tt}, j = 1 \rightarrow \text{ff})$  were fibrant in context  $\Gamma, i : \mathbb{1}, j : \mathbb{1}$ :

$$(\lambda i.[j.\text{Ext}(\cdot.\text{bool} \mid j = 0 \rightarrow \text{tt}, j = 1 \rightarrow \text{ff})] \downarrow_i^0 \text{tt}) : \text{path}_{\cdot.\text{bool}}(\text{tt}, \text{ff})$$

We therefore employ the following formation rule for *fibrant* extension types:

$$\frac{\text{TYPE FORMATION} \quad \Gamma, \vec{i} : \mathbb{1} \vdash A \text{ type}_{\alpha}^{\text{fib}} \quad \Gamma, \vec{i} : \mathbb{1} \vdash S : A \quad \vec{i} : \mathbb{1} \vdash \pi(S) : \mathbb{F}}{\Gamma \vdash \text{Ext}(\vec{i}.A \mid S) \text{ type}_{\alpha}^{\text{fib}}}$$

We present the remainder of the rules for extension (pre)types in Figure A.10.

### A.1.8 Partial element pretypes

The judgmental notions of partial types and their partial elements are internalized in RTT as a pretype connective. In fact, the version presented here significantly generalizes what is present in the actually-implemented versions of `redtt`, but its implementation is straightforward.

Given a partial pretype  $\Gamma \vdash S \text{ type}_{\alpha}^{\text{pre}}$ , the pretype of partial elements of  $S$  is written  $\text{Partial}(S)$ ; given an element  $M : \text{Partial}(S)$ , we have can extract a partial element  $\text{out}(M) : S$ . The formal rules for partial element pretypes are presented in Figure A.11

## A.2 Normalization by evaluation for RTT

Using a cubical extension to the standard normalization by evaluation algorithm (see Abel [Abe13]), we may heuristically obtain normal forms and check judgmental equality for RTT; the correctness (soundness, completeness) of `redtt`'s normalization algorithm is only conjectured, and would be a corollary of the proposed work for this dissertation. At present, we merely observe that the algorithm appears to work in practice.

In this section, we characterize the  $\beta/\eta$ -normal types and terms of RTT; terms of RTT (written in black) should be read as equivalence classes of well-typed terms; as a convention, we will write normal forms in `German Script`, construing them as a structure which lies over *judgmental equivalence classes* of the terms of RTT. The grammar of (left- and right-) normal representatives of RTT terms, as well as the judgments which characterize their well-formedness, are specified in Figure A.12.

For the sake of space we do *not* present in this document the (conjectured) normalization algorithm which we have implemented in the `redtt` proof assistant, emphasizing instead our conjectural characterization of the normal forms themselves; whereas our characterization of normal forms is likely to play an important role in the proposed

(types)  $A, B, \dots \mid \text{Ext}(\vec{i}.A \mid S)$   
 (terms)  $M, N \dots \mid \lambda\vec{i}.M \mid \text{app}_{\vec{i}.A \mid S}(M, \vec{r})$

PRETYPE FORMATION

$$\frac{\Gamma, \vec{i} : \vec{\mathbb{I}} \vdash A \text{ type}_\alpha^{\text{pre}} \quad \Gamma, \vec{i} : \vec{\mathbb{I}} \vdash S : A}{\Gamma \vdash \text{Ext}(\vec{i}.A \mid S) \text{ type}_\alpha^{\text{pre}}}$$

TYPE FORMATION

$$\frac{\Gamma, \vec{i} : \vec{\mathbb{I}} \vdash A \text{ type}_\alpha^{\text{fib}} \quad \vec{i} : \vec{\mathbb{I}} \vdash \pi(S) : \mathbb{F} \quad \Gamma, \vec{i} : \vec{\mathbb{I}} \vdash S : A}{\Gamma \vdash \text{Ext}(\vec{i}.A \mid S) \text{ type}_\alpha^{\text{fib}}}$$

Fixing  $\Gamma, \vec{i} : \vec{\mathbb{I}} \vdash A \text{ type}_\star^{\text{pre}}$  and  $\Gamma, \vec{i} : \vec{\mathbb{I}} \vdash S : A$ .

INTRODUCTION

$$\frac{\Gamma, \vec{i} : \vec{\mathbb{I}} \vdash M : A \ [S]}{\Gamma \vdash \lambda\vec{i}.M : \text{Ext}(\vec{i}.A \mid S)}$$

ELIMINATION

$$\frac{\Gamma \vdash M : \text{Ext}(\vec{i}.A \mid S) \quad \overline{\Gamma \vdash r : \vec{\mathbb{I}}}}{\Gamma \vdash \text{app}_{\vec{i}.A \mid S}(M, \vec{r}) : [\vec{r}/\vec{i}]A \ [S]}$$

UNICITY

$$\frac{\Gamma \vdash M : \text{Ext}(\vec{i}.A \mid S)}{\Gamma \vdash M = \lambda\vec{i}.\text{app}_{\vec{i}.A \mid S}(M, \vec{i}) : \text{Ext}(\vec{i}.A \mid S)}$$

COMPUTATION

$$\frac{\Gamma, \vec{i} : \vec{\mathbb{I}} \vdash M : A \quad \overline{\Gamma \vdash r : \vec{\mathbb{I}}}}{\Gamma \vdash \text{app}_{\vec{i}.A \mid S}(\lambda\vec{i}.M, \vec{r}) = [\vec{r}/\vec{i}]M : [\vec{r}/\vec{i}]A}$$

COERCION

$$\frac{C[s] \stackrel{\text{def}}{=} \text{Ext}(\vec{j}.[s/i]A \mid [s/i]S)}{\Gamma \vdash [i.C[i]] \downarrow_r^r, M = \lambda\vec{j}.[i.A] \downarrow_{r'}^r, [\text{app}(M, \vec{j}) \mid i.S] : C[r']}$$

HOMOGENEOUS COMPOSITION

$$\frac{C \stackrel{\text{def}}{=} \text{Ext}(\vec{j}.A \mid S)}{\Gamma \vdash C \downarrow_r^r, [M \mid i.T] = \lambda\vec{j}.A \downarrow_{r'}^r, [\text{app}(M, \vec{j}) \mid i.S, \text{app}(T, \vec{j})] : C}$$

Figure A.10: Rules for extension (pre)types.




(types)  $A, B \quad \dots \mid \text{Partial}(S)$   
 (terms)  $M, N \quad \dots \mid \text{partial}(S) \mid \text{out}_S(M)$

$$\begin{array}{c}
 \text{FORMATION} \\
 \frac{\Gamma \vdash S \text{ type}_\alpha^{\text{pre}}}{\Gamma \vdash \text{Partial}(S) \text{ type}_\alpha^{\text{pre}}} \\
 \\
 \text{INTRODUCTION} \\
 \frac{\Gamma \mid \vec{\xi} \vdash \vec{N} : \vec{A}}{\Gamma \vdash \text{partial}(\vec{\xi} \rightarrow \vec{N}) : \text{Partial}(\vec{\xi} \rightarrow \vec{A})} \\
 \\
 \text{ELIMINATION} \\
 \frac{\Gamma \vdash M : \text{Partial}(\vec{\xi} \rightarrow \vec{A}) \quad \Gamma \vdash \xi \text{ true}}{\Gamma \vdash \text{out}_{\vec{\xi} \rightarrow \vec{A}}(M) : A} \\
 \\
 \text{UNICITY} \\
 \frac{\Gamma \vdash M : \text{Partial}(\vec{\xi} \rightarrow \vec{A})}{\Gamma \vdash M = \text{partial}(\vec{\xi} \rightarrow \text{out}_{\vec{\xi} \rightarrow \vec{A}}(M)) : \text{Partial}(\vec{\xi} \rightarrow \vec{A})} \\
 \\
 \text{COMPUTATION} \\
 \frac{\Gamma \vdash \xi \text{ true}}{\Gamma \vdash \text{out}_{\vec{\xi} \rightarrow \vec{A}}(\text{partial}(\vec{\xi} \rightarrow N)) = N : A}
 \end{array}$$

Figure A.11: The rules for partial element pretypes in RTT.

work, the actual normalization algorithm (although equivalent) will have a very different appearance.

**Remark A.2.1** (Staging normal forms). It is tempting to characterize directly the normal elements of normal types; this perspective naturally gives rise to an algorithm to type check normal elements, but it is difficult to scale: when formulated in this way, the normal forms must be defined simultaneously with either a hereditary substitution operation or a normalization function.

A more tractable way to stage things is to characterize the normal elements of an *equivalence class* of ordinary types. The resulting definition is of course not algorithmic or even well-moded as a logic program, but we prefer to address algorithmic issues after developing a normalization algorithm and passing to elaboration. A common theme in both the prior work and the proposed work for this dissertation is to stage things in such a way that difficult proofs can be deferred until the intervening developments have rendered them tractable. 

(right norm. types)	$\mathbf{a}, \mathbf{b}$	$\mathbf{a}^+ \mid \mathbf{a}^-$
(right norm. pos. types)	$\mathbf{a}^+, \mathbf{b}^+$	$[\mathbf{e} : \text{univ}_\alpha^\kappa] \mid \text{univ}_\alpha^\kappa \mid \dots$
(right norm. neg. types)	$\mathbf{a}^-, \mathbf{b}^-$	$\text{pi}(x : \mathbf{a}. \mathbf{b}) \mid \text{sg}(x : \mathbf{a}. \mathbf{b}) \mid \dots$
(right norm. terms)	$\mathbf{m}, \mathbf{n}$	$[\mathbf{e} : \mathbf{a}^+] \mid \mathbf{a} \mid \text{lam}(x. \mathbf{m}) \mid \dots$
(right norm. partial elements)	$\mathbf{s}, \mathbf{t}$	$\xi \rightarrow \mathbf{n}$
(left norm. terms)	$\mathbf{e}, \mathbf{f}$	$x \mid \text{app}(\mathbf{e}, \mathbf{m}) \mid \text{fst}(\mathbf{e}) \mid \text{snd}(\mathbf{e}) \mid \dots$

$\Gamma \text{ ctx}$
$\Gamma \vdash A \text{ type}_\alpha^\kappa$
$\Gamma \vdash \text{type}_\alpha^\kappa \ni \mathbf{a} \text{ right } \rightsquigarrow [A]$ means that $\mathbf{a}$ is a <i>right-normal representative</i> of the type $A$ .
$\Gamma \vdash A \text{ type}_*^{\text{pre}} \quad \Gamma \vdash M : A$
$\Gamma \vdash A \ni \mathbf{m} \text{ right } \rightsquigarrow [M]$ means that $\mathbf{m}$ is a <i>right-normal representative</i> of $M : A$ .
$\Gamma \vdash A \ni \mathbf{e} \text{ left } \rightsquigarrow [M]$ means that $\mathbf{e}$ is a <i>left-normal representative</i> of $M : A$ .
$\Gamma \vdash S : A$
$\Gamma \vdash A \ni \mathbf{s} \text{ right/rgd } \rightsquigarrow [S]$ means that $\mathbf{s}$ is a <i>rigid right-normal representative</i> of the partial element $S$ . Rigidity means that the supporting constraint of the partial element is not true (but it does not have to be false).
$\Gamma \vdash A \ni \mathbf{s} \text{ right } \rightsquigarrow [S]$ means that $\mathbf{s}$ is a <i>normal representative</i> of the (not necessarily rigid) case tree $S$ .

Figure A.12: The grammar and judgments of RTT's normal forms.

### A.2.1 Core rules of normal forms

In this section, we summarize the main rules of normal forms which lie at the basis of cubical type theory with cumulative universes; it is this small collection of rules which is extended as further connectives (such as dependent sum and product types, extension types, partial element types, composite types, etc. are added). The left-normal forms are generated by the *variables*:

$$\frac{\Gamma \ni x : A}{\Gamma \vdash A \ni x \text{ left } \rightsquigarrow [x]}$$

The normal forms of RTT are meant to be  $\eta$ -long; therefore, one includes a *mode shift*  $[\mathbf{e} : \mathbf{a}^+]$  from left-normal to right-normal only at those types which lack an  $\eta$ -rule,

namely the positive types. In RTT (prior to extension by higher inductive types), the positive types are the universes and their left-normal elements.

$$\text{MODE SHIFT (TYPE)} \\ \frac{\Gamma \vdash U_\alpha^\kappa \ni \epsilon \text{ left } \rightsquigarrow [A] \quad \alpha \leq \beta \quad \kappa \leq \lambda}{\Gamma \vdash \text{type}_\beta^\lambda \ni [\epsilon : \text{univ}_\beta^\lambda] \text{ right } \rightsquigarrow [A]}$$

$$\text{MODE SHIFT (ELEMENT OF LEFT-NORMAL TYPE)} \\ \frac{\Gamma \vdash U_\alpha^\kappa \ni \epsilon_A \text{ left } \rightsquigarrow [A] \quad \Gamma \vdash A \ni \epsilon \text{ left } \rightsquigarrow [M]}{\Gamma \vdash A \ni [\epsilon : \epsilon_A] \text{ right } \rightsquigarrow [M]}$$

**Remark A.2.2** (Mode shift and subtyping). In contrast to some other presentations, we have found it particularly liberating to equip the different positive types with their own “mode shift” rules; for instance, the mode shift for types above incorporates cumulativity. No other subtyping principles are required for the normal forms, because the subsumption principle is otherwise admissible for terms in  $\eta$ -long form.  $\clubsuit$

*A.2.1.1 Universes* The connection between types of a particular level and elements of the corresponding universe is made in the rules below:

$$\frac{\alpha < \beta}{\Gamma \vdash \text{type}_\beta^\lambda \ni \text{univ}_\alpha^\kappa \text{ right } \rightsquigarrow [U_\alpha^\kappa]} \quad \frac{\Gamma \vdash \text{type}_\alpha^\kappa \ni a \text{ right } \rightsquigarrow [A] \quad \alpha < \star}{\Gamma \vdash U_\alpha^\kappa \ni a \text{ right } \rightsquigarrow [A]}$$

*A.2.1.2 Normal and rigid-normal partial elements* The normal partial elements (case trees) are characterized by the following rule:

$$\frac{\overrightarrow{\Gamma, \xi \vdash A \ni n \text{ right } \rightsquigarrow [N]}}{\Gamma \vdash A \ni \xi \rightarrow n \text{ right } \rightsquigarrow [\xi \rightarrow N]}$$

A normal partial element is called *rigid-normal* if its underlying constraint  $\vec{\xi}$  is not true; rigid-normal partial elements are important for characterizing the normal Kan operations.

$$\frac{\Gamma \not\vdash \vec{\xi} \text{ true} \quad \Gamma \vdash A \ni \overrightarrow{\xi \rightarrow n} \text{ right } \rightsquigarrow [\overrightarrow{\xi \rightarrow N}]}{\Gamma \vdash A \ni \overrightarrow{\xi \rightarrow n} \text{ right/rgd } \rightsquigarrow [\overrightarrow{\xi \rightarrow N}]}$$

## A.2.2 Normal forms for Kan operations

<i>(norm. neg. types)</i>	$a^-, b^-$	...		$\text{univ}_\alpha^\kappa \downarrow_{r'}^r, [m \mid i.s]$
<i>(norm. terms)</i>	$m, n$	...		$[m \mid t]$
<i>(neu. terms)</i>	$\epsilon$	...		$[i.e] \downarrow_{r'}^r, m \mid \epsilon \downarrow_{r'}^r, [m \mid i.s]$

The Kan operations must be treated carefully when determining appropriate normal forms. One source of left-normals is the coercions and compositions applied at left-normal types; a left-normal coercion or composition must be between two distinct dimensions  $r \neq r'$ , and a left-normal composition must additionally have a *rigid* tube.

## LEFT-NORMAL COERCION

$$\frac{\begin{array}{l} \Gamma \not\vdash r = r' : \mathbb{1} \\ \Gamma, i : \mathbb{1} \vdash U_{\alpha}^{\text{fib}} \ni \epsilon \text{ left } \rightsquigarrow [A] \\ \Gamma \vdash [r/i]A \ni m \text{ right } \rightsquigarrow [M] \end{array}}{\Gamma \vdash [r'/i]A \ni [i.\epsilon] \downarrow_{r'}^r, m \text{ left } \rightsquigarrow [[i.A] \downarrow_{r'}^r, M]}$$

## LEFT-NORMAL HOMOGENEOUS COMPOSITION

$$\frac{\begin{array}{l} \Gamma \not\vdash r = r' : \mathbb{1} \\ \Gamma \vdash U_{\alpha}^{\text{fib}} \ni \epsilon \text{ left } \rightsquigarrow [A] \\ \Gamma \vdash A \ni m \text{ right } \rightsquigarrow [M] \\ \Gamma, i : \mathbb{1} \vdash A \ni s \text{ right/rgd } \rightsquigarrow [S] \end{array}}{\Gamma \vdash A \ni \epsilon \downarrow_{r'}^r, [m \mid i.s] \text{ left } \rightsquigarrow [A \downarrow_{r'}^r, [M \mid i.S]]}$$

On the other hand, some compositions can be *normal*; for instance, a composition in the universe between distinct dimensions along a rigid tube is a *normal* type; as such, we must also consider the normal forms which arise from its introduction and elimination forms.

## RIGHT-NORMAL COMPOSITE TYPE

$$\frac{\begin{array}{l} \Gamma \not\vdash r = r' : \mathbb{1} \quad \alpha < \beta \\ \Gamma \vdash U_{\alpha}^{\kappa} \ni a \text{ right } \rightsquigarrow [A] \quad \Gamma, i : \mathbb{1} \vdash A \ni s \text{ right/rgd } \rightsquigarrow [S] \end{array}}{\Gamma \vdash \text{type}_{\beta}^{\lambda} \ni \text{univ}_{\alpha}^{\kappa} \downarrow_{r'}^r, [a \mid i.s] \text{ right } \rightsquigarrow [U_{\alpha}^{\kappa} \downarrow_{r'}^r, [A \mid i.S]]}$$

## RIGHT-NORMAL COMPOSITE TYPE ELEMENT

$$\frac{\begin{array}{l} \Gamma \not\vdash r = r' : \mathbb{1} \quad \Gamma \not\vdash \vec{\xi} \text{ true} \\ \Gamma \vdash A \ni m \text{ right } \rightsquigarrow [M] \quad \Gamma, \xi \vdash [r'/i]B \ni n \text{ right } \rightsquigarrow [N] \end{array}}{\Gamma \vdash U_{\alpha}^{\text{fib}} \downarrow_{r'}^r, [A \mid i.\vec{\xi} \rightarrow \vec{B}] \ni [m \mid \vec{\xi} \rightarrow n] \text{ right } \rightsquigarrow [[M \mid \vec{\xi} \rightarrow N]]}$$

## LEFT-NORMAL COMPOSITE TYPE ELIMINATION

$$\frac{\begin{array}{l} \Gamma \not\vdash r = r' : \mathbb{1} \quad \Gamma \not\vdash \vec{\xi} \text{ true} \quad \Gamma \vdash U_{\alpha}^{\text{fib}} \downarrow_{r'}^r, [A \mid i.\vec{\xi} \rightarrow \vec{B}] \ni \epsilon \text{ left } \rightsquigarrow [M] \end{array}}{\Gamma \vdash A \ni \text{cap}(\epsilon) \text{ left } \rightsquigarrow [\text{cap}_{U_{\alpha}^{\text{fib}} \downarrow_{r'}^r, [A \mid i.\vec{\xi} \rightarrow \vec{B}]}(M)]}$$

### A.2.3 Normal forms for connectives

Normal forms for dependent sum and product types are standard, but we present them here for completeness.

$$\begin{array}{c}
\frac{\Gamma \vdash \text{type}_\alpha^\kappa \ni \mathfrak{a} \text{ right } \rightsquigarrow [A] \quad \Gamma, x : A \vdash \text{type}_\alpha^\kappa \ni \mathfrak{b} \text{ right } \rightsquigarrow [B]}{\Gamma \vdash \text{type}_\alpha^\kappa \ni \text{sg}(x : \mathfrak{a}.\mathfrak{b}) \text{ right } \rightsquigarrow [(x : A) \times B]} \\
\Gamma \vdash \text{type}_\alpha^\kappa \ni \text{pi}(x : \mathfrak{a}.\mathfrak{b}) \text{ right } \rightsquigarrow [(x : A) \rightarrow B] \\
\frac{\Gamma, x : A \vdash B \ni \mathfrak{m} \text{ right } \rightsquigarrow [M]}{\Gamma \vdash (x : A) \rightarrow B \ni \text{lam}(x.\mathfrak{m}) \text{ right } \rightsquigarrow [\lambda x.M]} \\
\frac{\Gamma \vdash A \ni \mathfrak{m} \text{ right } \rightsquigarrow [M] \quad \Gamma \vdash [M/x]B \ni \mathfrak{n} \text{ right } \rightsquigarrow [N]}{\Gamma \vdash (x : A) \times B \ni \text{pair}(\mathfrak{m}, \mathfrak{n}) \text{ right } \rightsquigarrow [\langle M, N \rangle]} \\
\frac{\Gamma \vdash (x : A) \times B \ni \mathfrak{e} \text{ left } \rightsquigarrow [M]}{\Gamma \vdash A \ni \text{fst}(\mathfrak{e}) \text{ left } \rightsquigarrow [\text{fst}_{x:A.B}(M)]} \\
\Gamma \vdash [\text{fst}_{x:A.B}(M)/x]B \ni \text{snd}(\mathfrak{e}) \text{ left } \rightsquigarrow [\text{snd}_{x:A.B}(M)] \\
\frac{\Gamma \vdash (x : A) \rightarrow B \ni \mathfrak{e} \text{ left } \rightsquigarrow [M] \quad \Gamma \vdash A \ni \mathfrak{n} \text{ right } \rightsquigarrow [N]}{\Gamma \vdash \text{app}_{x:A.B}(M, N) \ni \text{app}(\mathfrak{e}, \mathfrak{n}) \text{ left } \rightsquigarrow [[N/x]B]}
\end{array}$$

### A.3 Elaboration relative to a boundary

`redtt`'s surface language contains *holes* ?, which stand for incomplete parts of a construction. Elaboration with holes has been treated many times (see McBride [McB99], Norell [Nor07], Asperti, Ricciotti, Coen, and Tassi [Asp+11], and Brady [Bra13]), but off-the-shelf accounts lead to poor results in the context of cubical type theories, as we will explain below.

In ordinary type theory, typing rules do not contain equality judgments as premises;<sup>1</sup> for this reason, it is easy to treat the elaboration of a hole as a new top-level declaration of particular (pre)type, maintaining McBride's invariant that the partial proof term is always well-typed. In cubical type theory, however, equality judgments appear in the premises to typing rules in two situations:

- 1) In the introduction rule for extension (path) types, the body of the abstraction must have the appropriate boundary.
- 2) In the rule for forming compositions, the cap and the tube must coincide at the corners, and moreover, intersecting parts of the tube must coincide.

Using standard accounts of holes (such as McBride [McB99]) without additionally altering `redtt`'s judgmental structure would make it impossible in most cases to place a hole underneath a dimension binder, because the synchronous boundary checks would immediately fail: a hole is completely generic, and does not match any boundary except by virtue of an  $\eta$ -law.

<sup>1</sup>Except for the conversion rule, which is rendered differently in *algorithmic* presentations of type theory.

Our boundary-sensitive elaboration algorithm is similar to other bidirectional elaboration algorithms, but differs in some critical ways from the traditional formulations; in particular, we locate the *mode switch* between synthesis and checking differently. Whereas this mode switch is traditionally activated on the basis of the syntax of the term being elaborated, in *redtt* we activate the mode-switch in a type-directed fashion, based on *polarity*.

As we will see in Appendix A.1.7, the polarized account of bidirectional elaboration corresponds to building  $\eta$ -expansion into the elaborator, using the intuition that the negative types are the ones which are characterized by  $\eta$ -laws. Our account leads to a smoother implementation of subtyping for extension types of increasingly refined boundaries, a *façon de parler* which is essential for usability.

### A.3.1 Basic elaboration structure and rules

The main aspects of the bidirectional elaboration algorithm can be presented prior to introducing any connectives. *Constructors* are terms which can be checked against a specific type, and embed the *destructors* which determine their own type:

(dimensions)	$r$	$0 \mid 1 \mid i$
(constructors)	$M$	$E \mid \dots$
(partial elements)	$S$	$\cdot \mid r = r' \rightarrow M, S$
(destructors)	$E$	$(M : A) \mid x \mid \dots$

The main forms of elaboration judgment are summarized in Figure A.13. These judgments allow elements of any type to be checked relative to a boundary through the judgment form  $\Gamma \vdash M \Leftarrow \alpha [S] \rightsquigarrow [M]$ ; for negative types (types with  $\eta$ -laws), this is particularly useful because the boundary for the premises can be uniformly reconstructed. This corresponds to an *asynchronous* account of boundary checking.

On the other hand, the boundaries of elements of positive type cannot be uniformly transformed from conclusion to premise without unification. Rather than developing unification for cubical type theory, we simply elaborate elements of positive type using a boundary-free form of judgment  $\Gamma \vdash M \Leftarrow \alpha^+ \rightsquigarrow [M]$ ; the interface between these two forms of judgment is governed by the following rule, which performs a *synchronous* boundary check:

$$\frac{(\Gamma \vdash \text{type}_*^{\text{pre}} \ni \alpha^+ \text{ right } \rightsquigarrow [A]) \quad \Gamma \vdash M \Leftarrow \alpha^+ \rightsquigarrow [M] \quad \overline{\Gamma, \xi \vdash M = N : A}}{\Gamma \vdash M \Leftarrow \alpha^+ [\xi \rightarrow \vec{N}] \rightsquigarrow [M]}$$

Types are elaborated using the judgment form  $\Gamma \vdash A \Leftarrow \text{type}_\alpha^{\text{c}} \rightsquigarrow [A]$ ; this differs from  $\Gamma \vdash A \Leftarrow \text{univ}_\alpha^{\text{c}} \rightsquigarrow [A]$  because in the latter, we of course have the restriction  $\alpha < \star$ . Types, elements of the universe, and their cumulativity are all negotiated using the following three rules of elaboration:

$\Gamma \text{ ctx}$
$\Gamma \vdash r : \mathbb{I}$ $\Gamma \vdash \mathbf{r} \Leftarrow \mathbb{I} \rightsquigarrow [r]$ means that the dimension code $\mathbf{r}$ elaborates to the dimension $r$ .
$\Gamma \vdash A \text{ type}_{\alpha}^{\kappa}$ $\Gamma \vdash \mathbf{A} \Leftarrow \text{type}_{\alpha}^{\kappa} \rightsquigarrow [A]$ means that the constructor code $\mathbf{A}$ elaborates to the judgmental equivalence class of the type $A$ at level $\alpha$ and kind $\kappa$ .
$\Gamma \vdash A \text{ type}_{\star}^{\text{pre}} \quad \Gamma \vdash \text{type}_{\star}^{\text{pre}} \ni \mathfrak{a} \text{ right } \rightsquigarrow [A] \quad \Gamma \vdash S : A$
$\Gamma \vdash M : A \ [S]$ $\Gamma \vdash \mathbf{M} \Leftarrow \mathfrak{a} [S] \rightsquigarrow [M]$ means that the constructor code $\mathbf{M}$ elaborates at type $\mathfrak{a}$ to the judgmental equivalence class of the term $M$ , matching the boundary $S$ .
$\Gamma \vdash T : A \ [S]$ $\Gamma \vdash \mathbf{T} \Leftarrow \mathfrak{a} [S] \rightsquigarrow [T]$ means that the partial element code $\mathbf{T}$ elaborates a type $\mathfrak{a}$ to the judgmental equivalence class of the partial element $T$ , which agrees with the partial element $S$ .
$\Gamma \vdash \vec{\xi} : \mathbb{F} \quad \Gamma \mid \vec{\xi} \vdash \vec{M} : A$ $\Gamma \mid \vec{\xi} \vdash \vec{\mathbf{M}} \Leftarrow \mathfrak{a} [S] \rightsquigarrow [\vec{M}]$ means that the sequence of constructor codes $\vec{\mathbf{M}}$ elaborates at type $\mathfrak{a}$ to the judgmental equivalence class of the case tree $\vec{\xi} \rightarrow \vec{M}$ , agreeing with the partial element $S$ .
$\Gamma \vdash A \text{ type}_{\star}^{\text{pre}} \quad \Gamma \vdash \text{type}_{\star}^{\text{pre}} \ni \mathfrak{a}^{+} \text{ right } \rightsquigarrow [A] \quad \Gamma \vdash M : A$
$\Gamma \vdash \mathbf{M} \Leftarrow \mathfrak{a}^{+} \rightsquigarrow [M]$ means that the constructor code $\mathbf{M}$ elaborates at type $\mathfrak{a}^{+}$ to the judgmental equivalence class of the term $M$ .
$\Gamma \vdash \mathbf{E} \Rightarrow \mathfrak{a} \rightsquigarrow [M]$ means that the destructor code $\mathbf{E}$ elaborates to the judgmental equivalence class of the term $M$ of type $\mathfrak{a}$ .

Figure A.13: The elaboration judgments of `redtt`.

$$\frac{\Gamma \vdash E \Rightarrow \text{univ}_\alpha^\kappa \rightsquigarrow [A] \quad \alpha \leq \beta \quad \kappa \leq \lambda}{\Gamma \vdash E \Leftarrow \text{type}_\beta^\lambda \rightsquigarrow [A]} \quad \frac{\Gamma \vdash M \Leftarrow \text{type}_\alpha^\kappa \rightsquigarrow [M]}{\Gamma \vdash M \Leftarrow \text{univ}_\alpha^\kappa \rightsquigarrow [M]}$$

$$\frac{\Gamma \vdash E \Rightarrow [f : \text{univ}_\alpha^\kappa] \rightsquigarrow [M]}{\Gamma \vdash E \Leftarrow [f : \text{univ}_*^{\text{pre}}] \rightsquigarrow [M]}$$

Type-annotated cuts in the surface language are elaborated through a mode switch:

$$\frac{(\Gamma \vdash \text{type}_*^{\text{pre}} \ni a \text{ right} \rightsquigarrow [A]) \quad \Gamma \vdash A \Leftarrow \text{type}_*^{\text{pre}} \rightsquigarrow [A] \quad \Gamma \vdash M \Leftarrow a \rightsquigarrow [M]}{\Gamma \vdash (M : A) \Rightarrow a \rightsquigarrow [M]}$$

*A.3.1.1 Elaborating partial elements* A partial element defined on a constraint  $\vec{\xi}$  is a family of elements  $M_\xi$ , such that along each overlap  $\xi, \xi'$  we have  $M_\xi = M_{\xi'}$ . Whereas in the declarative calculus for RTT it is natural to treat the coherence of partial elements at intersections as a side condition, in a calculus oriented toward *interactive* proof it is much better to divide up the obligation among the subgoals.

This can be done in two phases; first we elaborate the constraint, and then we recursively elaborate the case tree, accumulating increasingly many constraints in the partial boundary:

$$\frac{\overline{\Gamma \vdash r \Leftarrow \mathbb{1} \rightsquigarrow [r]} \quad \overline{\Gamma \vdash s \Leftarrow \mathbb{1} \rightsquigarrow [s]} \quad \Gamma \mid \overline{r = s} \vdash \vec{M} \Leftarrow a [T] \rightsquigarrow [\vec{M}]}{\Gamma \vdash \overline{r = s} \rightarrow \vec{M} \Leftarrow a [T] \rightsquigarrow [r = s \rightarrow \vec{M}]}$$

CONS

$$\frac{\Gamma \vdash \text{type}_*^{\text{pre}} \ni a \text{ right} \rightsquigarrow [A] \quad \Gamma, r = s \vdash \text{type}_*^{\text{pre}} \ni a_{r=s} \text{ right} \rightsquigarrow [A] \quad \Gamma, r = s \vdash M \Leftarrow a_{r=s} [T] \rightsquigarrow [M] \quad \Gamma \mid \overline{r = s} \vdash M, \vec{M} \Leftarrow a [r = s \rightarrow M, T] \rightsquigarrow [M, \vec{M}]}{\Gamma \mid r = s, \overline{r = s} \vdash M, \vec{M} \Leftarrow a [T] \rightsquigarrow [M, \vec{M}]}$$

NIL

$$\frac{}{\Gamma \mid \cdot \vdash \cdot \Leftarrow a [T] \rightsquigarrow [\cdot]}$$

*A.3.1.2 Elaborating extension types* Extension types simultaneously exhibit nearly all the important aspects of *redtt*'s elaboration algorithm; therefore, we explain them in detail in this section, beginning with an appropriate extension to the grammar of the *redtt* surface language.

$$\begin{array}{l} \text{(constructors)} \quad M, A \quad \dots \mid \text{Ext}(\vec{1} \rightarrow A \mid S) \mid \text{Ext}(\vec{1} \rightarrow A) \mid \setminus \vec{1} \rightarrow M \\ \text{(destructors)} \quad E \quad \dots \mid E \cdot \vec{r} \end{array}$$

*Formation rules* Just as in Appendix A.1.7, we must provide three rules of elaboration depending on whether we are elaborating an extension pretype, a fibrant extension type along a cofibration, or a fibrant cube type. For the sake of illustration, we present only the elaboration rule for a fibrant extension type along a cofibration.



$$\frac{\begin{array}{l} \overrightarrow{i : \mathbb{I} \vdash r} \Leftarrow \mathbb{I} \rightsquigarrow [r] \quad \overrightarrow{i : \mathbb{I} \vdash s} \Leftarrow \mathbb{I} \rightsquigarrow [s] \quad \Gamma, \overrightarrow{i : \mathbb{I} \vdash A} \Leftarrow \text{type}_\alpha^{\text{fib}} \rightsquigarrow [A] \\ \Gamma, \overrightarrow{i : \mathbb{I} \vdash \text{type}_\alpha^{\text{fib}} \ni a} \text{ right } \rightsquigarrow [A] \quad \Gamma, \overrightarrow{i : \mathbb{I} \mid \vec{\xi} \vdash \vec{M}} \Leftarrow a \rightsquigarrow [M] \end{array}}{\Gamma \vdash \text{Ext}(\overrightarrow{\mathbb{I}} \rightarrow A \mid r = r' \rightarrow \vec{M}) \Leftarrow \text{type}_\alpha^{\text{fib}} \rightsquigarrow [\text{Ext}(\vec{i}.A \mid r = r' \rightarrow \vec{M})]}$$

*Type checking (constructors)* The rule for checking an introduction form of the extension type exposes the main idea of boundary-sensitive elaboration. To elaborate  $\overrightarrow{\mathbb{I}} \rightarrow \vec{M}$  along the partial element  $\Gamma \vdash T : \text{Ext}(\vec{i}.A \mid S)$ , we extend the context by  $\overrightarrow{i : \mathbb{I}}$  and elaborate  $\vec{M}$  along the partial element  $[S, \dots \text{app}_{\vec{i}.A \mid S}(T, \vec{i})]$ , where by the latter we mean the system of constraints  $S$  extended by  $\overrightarrow{\xi \rightarrow \text{app}_{\vec{i}.A \mid S}(N, \vec{i})}$  where  $T \stackrel{\text{def}}{=} \overrightarrow{\xi \rightarrow N}$ .

$$\frac{\begin{array}{l} (\dots \quad \Gamma, \overrightarrow{i : \mathbb{I} \vdash A} \ni s \text{ right } \rightsquigarrow [S]) \\ \Gamma, \overrightarrow{i : \mathbb{I} \vdash \vec{M}} \Leftarrow a [S, \dots \text{app}_{\vec{i}.A \mid S}(T, \vec{i})] \rightsquigarrow [M] \end{array}}{\Gamma \vdash \overrightarrow{\mathbb{I}} \rightarrow \vec{M} \Leftarrow \text{ext}(\vec{i}.a \mid s) [T] \rightsquigarrow [\lambda \vec{i}.M]}$$

This technique is initially easier to grasp when considering the special case of the path type connective.

$$\frac{\begin{array}{l} (\dots \quad \Gamma \vdash [0/i]A \ni n_0 \text{ right } \rightsquigarrow [N_0] \quad \Gamma \vdash [1/i]A \ni n_1 \text{ right } \rightsquigarrow [N_1]) \\ \Gamma, \overrightarrow{i : \mathbb{I} \vdash \vec{M}} \Leftarrow a [i = 0 \rightarrow N_0, i = 1 \rightarrow N_1, \dots \text{app}_{(\dots)}(T, i)] \rightsquigarrow [M] \end{array}}{\Gamma \vdash \overrightarrow{\mathbb{I}} \rightarrow \vec{M} \Leftarrow \text{path}_{i.a}(n_0, n_1) [T] \rightsquigarrow [\lambda i.M]}$$

The partial element from the conclusion is therefore adjusted in two ways in the premise:

- 1) First, the original partial element  $T : \text{Ext}(\vec{i}.A \mid S)$  is turned into a partial element of  $A$  by applying it pointwise to the abstracted variables.
- 2) Then, it is intersected with the partial element  $S$ ; in the case of the path type, this is a partial element defined along the boundary  $i = 0, i = 1$ .

*Type synthesis (destructors)* The destructor term of the extension type is elaborated in the traditional way, adding only the missing annotations.

$$\frac{\begin{array}{l} \Gamma \vdash \text{type}_\star^{\text{pre}} \ni a \text{ right } \rightsquigarrow [A] \quad \Gamma \vdash \text{type}_\star^{\text{pre}} \ni a_{\vec{r}} \text{ right } \rightsquigarrow [[\vec{r}/\vec{i}]A] \\ \Gamma \vdash E \Rightarrow \text{ext}(\vec{i}.a \mid s) \rightsquigarrow [M] \quad \Gamma \vdash r \Leftarrow \mathbb{I} \rightsquigarrow [r] \end{array}}{\Gamma \vdash E \cdot \vec{r} \Rightarrow a_{\vec{r}} \rightsquigarrow [\text{app}_{\vec{i}.A \mid S}(M, \vec{r})]}$$

*Type checking (destructors)* In traditional bidirectional type checking algorithms, to check a destructor  $E$  against a type  $\alpha$ , one switches to synthesizing the type  $\alpha'$  of  $E$ , and then checks  $\alpha'$  is equal to (or a subtype of)  $\alpha$  [Coq96]. However, we have found that there is a more refined perspective that avoids the need for complicated subtyping rules in *redtt*.

We instead locate the *mode switch* from checking to synthesis only at types which lack an  $\eta$ -rule (such as left-normal types, universes, and inductive types), and on types which are characterized by an  $\eta$ -rule (function, product, extension, partial element, and composite types), we continue checking even on destructors, by *extending* the spine with a corresponding elimination form.

$$\frac{\dots \quad \Gamma, i : \mathbb{1} \vdash E \cdot \vec{1} \Leftarrow \alpha [S, \dots \text{app}_{i.A|S}(T, \vec{i})] \rightsquigarrow [M]}{\Gamma \vdash E \Leftarrow \text{ext}(\vec{i}.\alpha \mid s) [T] \rightsquigarrow [\lambda \vec{i}.M]}$$

This algorithm enforces an  $\eta$ -long normal form for bidirectional typing *derivations*, and is reminiscent of the way that the boundary between left- and right-normal elements of a type in Appendix A.2 is determined by whether that type possesses an  $\eta$ -rule.

The up-shot is that the surface language of *redtt* exhibits a kind of *virtual subtyping* for extension types, in which an element of a more constrained extension type can be transparently used as if it were an element of a less constrained extension type (without requiring any modification to the RTT core language). For instance, the *redtt* term  $\backslash p \rightarrow p$  can be elaborated at the normal type  $c \stackrel{\text{def}}{=} \text{path}_{i.A}(x_0, x_1) \rightarrow \text{ext}(i.A)$  as follows (letting  $\Gamma \stackrel{\text{def}}{=} A : U_\alpha^\kappa, x_0 : A, x_1 : A$ ):

$$\frac{\frac{\frac{\frac{\Gamma, p : \text{path}_{i.A}(x_0, x_1), i : \mathbb{1} \vdash p \Rightarrow \text{path}_{i.A}(x_0, x_1) \rightsquigarrow [p] \quad \dots, i : \mathbb{1} \vdash i \Leftarrow \mathbb{1} \rightsquigarrow [i]}{\Gamma, p : \text{path}_{i.A}(x_0, x_1), i : \mathbb{1} \vdash p \cdot i \Rightarrow A \rightsquigarrow [\text{app}_{i.A|i=0 \rightarrow x_0, i=1 \rightarrow x_1}(p, i)]}}{\Gamma, p : \text{path}_{i.A}(x_0, x_1), i : \mathbb{1} \vdash p \cdot i \Leftarrow A [\cdot] \rightsquigarrow [\text{app}_{i.A|i=0 \rightarrow x_0, i=1 \rightarrow x_1}(p, i)]}}{\Gamma, p : \text{path}_{i.A}(x_0, x_1) \vdash p \Leftarrow \text{ext}(i.A) [\cdot] \rightsquigarrow [\lambda i.\text{app}_{i.A|i=0 \rightarrow x_0, i=1 \rightarrow x_1}(p, i)]}}{\Gamma \vdash \backslash p \rightarrow p \Leftarrow c [\cdot] \rightsquigarrow [\lambda p.\lambda i.\text{app}_{i.A|i=0 \rightarrow x_0, i=1 \rightarrow x_1}(p, i)]}}$$

**A.3.1.3 Elaborating dependent sum types** We present the rules for elaborating dependent sum types here; the only difference from standard algorithms is located in the existence of *checking*-rules for destructors, as described in Appendix A.3.1.2

$$\begin{array}{ll} \text{(constructors)} & M, A \quad \dots \mid (x : A) \times B \mid \langle M, N \rangle \\ \text{(destructors)} & E \quad \dots \mid E.\text{fst} \mid E.\text{snd} \end{array}$$

$$\frac{\frac{\Gamma \vdash A \Leftarrow \text{type}_\alpha^\kappa \rightsquigarrow [A] \quad \Gamma, x : A \vdash B \Leftarrow \text{type}_\alpha^\kappa \rightsquigarrow [B]}{\Gamma \vdash (x : A) \times B \Leftarrow \text{type}_\alpha^\kappa \rightsquigarrow [(x : A) \times B]}}$$

Fixing  $\Gamma \vdash \text{type}_\star^{\text{pre}} \ni \mathfrak{a} \text{ right } \rightsquigarrow [A]$  and  $\Gamma, x : A \vdash \text{type}_\star^{\text{pre}} \ni \mathfrak{b} \text{ right } \rightsquigarrow [B]$ :

$$\frac{\begin{array}{l} \Gamma \vdash \mathfrak{M} \Leftarrow \mathfrak{a} [\text{fst}_{x:A.B}(S)] \rightsquigarrow [M] \\ \Gamma \vdash \text{type}_\star^{\text{pre}} \ni \mathfrak{b}_M \text{ right } \rightsquigarrow [[M/x]B] \\ \Gamma \vdash \mathfrak{N} \Leftarrow \mathfrak{b}_M [\text{snd}_{x:A.B}(S)] \rightsquigarrow [N] \end{array}}{\Gamma \vdash \langle \mathfrak{M}, \mathfrak{N} \rangle \Leftarrow \text{sg}(x : \mathfrak{a}. \mathfrak{b}) [S] \rightsquigarrow [\langle M, N \rangle]}$$

$$\frac{\begin{array}{l} \Gamma \vdash \mathfrak{E} \text{fst} \Leftarrow \mathfrak{a} [\text{fst}_{x:A.B}(S)] \rightsquigarrow [M] \\ \Gamma \vdash \text{type}_\star^{\text{pre}} \ni \mathfrak{b}_M \text{ right } \rightsquigarrow [[M/x]B] \\ \Gamma \vdash \mathfrak{E} \text{snd} \Leftarrow \mathfrak{b}_M [\text{snd}_{x:A.B}(S)] \rightsquigarrow [N] \end{array}}{\Gamma \vdash \mathfrak{E} \Leftarrow \text{sg}(x : \mathfrak{a}. \mathfrak{b}) [S] \rightsquigarrow [\langle M, N \rangle]}$$

$$\frac{\Gamma \vdash \mathfrak{E} \Rightarrow \text{sg}(x : \mathfrak{a}. \mathfrak{b}) \rightsquigarrow [M]}{\Gamma \vdash \mathfrak{E} \text{fst} \Rightarrow \mathfrak{a} \rightsquigarrow [\text{fst}_{x:A.B}(M)]}$$

$$\frac{\begin{array}{l} \Gamma \vdash \mathfrak{E} \Rightarrow \text{sg}(x : \mathfrak{a}. \mathfrak{b}) \rightsquigarrow [M] \\ \Gamma \vdash \text{type}_\star^{\text{pre}} \ni \mathfrak{b}_{M_0} \text{ right } \rightsquigarrow [[\text{fst}_{x:A.B}(M)/x]B] \end{array}}{\Gamma \vdash \mathfrak{E} \text{snd} \Rightarrow \mathfrak{b}_{M_0} \rightsquigarrow [\text{snd}_{x:A.B}(M)]}$$

**A.3.1.4** *Elaborating dependent product types* The elaboration of dependent product types presented here is likewise mostly standard, aside from the checking-mode elaboration of destructors.

$$\begin{array}{l} (\text{constructors}) \quad \mathfrak{M}, \mathfrak{A} \quad \dots \mid (x : \mathfrak{A}) \rightarrow \mathfrak{A} \mid \backslash x \rightarrow \mathfrak{M} \\ (\text{destructors}) \quad \mathfrak{E} \quad \dots \mid \mathfrak{E} \cdot \mathfrak{M} \end{array}$$

$$\frac{\begin{array}{l} \Gamma \vdash \mathfrak{A} \Leftarrow \text{type}_\alpha^\kappa \rightsquigarrow [A] \\ \Gamma, x : A \vdash \mathfrak{B} \Leftarrow \text{type}_\alpha^\kappa \rightsquigarrow [B] \end{array}}{\Gamma \vdash (x : \mathfrak{A}) \rightarrow \mathfrak{A} \Leftarrow \text{type}_\alpha^\kappa \rightsquigarrow [(x : A) \rightarrow B]}$$

Fixing  $\Gamma \vdash \text{type}_\star^{\text{pre}} \ni \mathfrak{a} \text{ right } \rightsquigarrow [A]$  and  $\Gamma, x : A \vdash \text{type}_\star^{\text{pre}} \ni \mathfrak{b} \text{ right } \rightsquigarrow [B]$ :

$$\frac{\begin{array}{l} \Gamma, x : A \vdash \mathfrak{M} \Leftarrow \mathfrak{b} [\text{app}_{x:A.B}(S, x)] \rightsquigarrow [M] \\ \Gamma \vdash \backslash x \rightarrow \mathfrak{M} \Leftarrow \text{pi}(x : \mathfrak{a}. \mathfrak{b}) [S] \rightsquigarrow [\lambda x. M] \end{array}}{\Gamma, x : A \vdash \mathfrak{E} \cdot x \Leftarrow \mathfrak{B} [\text{app}_{x:A.B}(S, x)] \rightsquigarrow [M]}$$

$$\frac{\Gamma \vdash \mathfrak{E} \Leftarrow \text{pi}(x : \mathfrak{a}. \mathfrak{b}) [S] \rightsquigarrow [\lambda x. M]}{\Gamma \vdash \mathfrak{E} \Leftarrow \text{pi}(x : \mathfrak{a}. \mathfrak{b}) [S] \rightsquigarrow [\lambda x. M]}$$

$$\frac{\begin{array}{l} \Gamma \vdash M : (x : A) \rightarrow B \quad \Gamma \vdash \mathfrak{E} \Rightarrow \text{pi}(x : \mathfrak{a}. \mathfrak{b}) \rightsquigarrow [M] \\ \Gamma \vdash \mathfrak{N} \Leftarrow \mathfrak{a} [\cdot] \rightsquigarrow [N] \quad \Gamma \vdash \text{type}_\star^{\text{pre}} \ni \mathfrak{b}_N \text{ right } \rightsquigarrow [[N/x]B] \end{array}}{\Gamma \vdash \mathfrak{E} \cdot \mathfrak{N} \Rightarrow \mathfrak{b}_N \rightsquigarrow [\text{app}_{x:A.B}(M, N)]}$$

## A.3.2 Interactive theorem proving and the elaboration of holes

The main advantage of `redtt`'s elaboration structure is that it is possible to place a hole `?` in nearly any location that expects a constructor and receive an appropriate subgoal:

$$(\text{constructors}) \quad \mathfrak{M} \quad \dots \mid \boxed{?}$$

Intuitively, this is possible because the entire elaboration environment (context, type, boundary) can be reified into a *pretype*, and a generic element of this pretype satisfies the typing and boundary conditions automatically. This reification is enabled by RTT's extension pretypes and partial element pretypes, thereby extending the treatment of interactive proof/program development pioneered by Epigram [Bra+11] and Idris [Bra13] to the higher-dimensional setting.

**A.3.2.1 Reifying goals** We will define a meta-operation  $\mathbf{ReifyGoal}_\Delta(\Gamma; C; S)$ , given  $\Delta, \Gamma \text{ ctx}$  and  $\Delta, \Gamma \vdash C \text{ type}_*^{\text{pre}}$  and  $\Delta, \Gamma \vdash S : C$ . Because some parameters remain fixed throughout the definition, it will be convenient to write  $R_\Delta(\Gamma)$  for  $\mathbf{ReifyGoal}_\Delta(\Gamma; C; S)$ . We define  $\Delta \vdash \mathbf{ReifyGoal}_\Delta(\Gamma; C; S) \text{ type}_*^{\text{pre}}$  by (backward) recursion on  $\Gamma$ :

$$\begin{aligned} R_\Delta(\cdot) &= \text{Ext}(\cdot.C \mid S) \\ R_\Delta(x : A, \Gamma) &= (x : A) \rightarrow R_{\Delta, x:A}(\Gamma) \\ R_\Delta(i : \mathbb{1}, \Gamma) &= \text{Ext}(i.R_{\Delta, i:\mathbb{1}}(\Gamma)) \\ R_\Delta(\xi, \Gamma) &= \text{Partial}(\xi \rightarrow R_{\Delta, \xi}(\Gamma)) \end{aligned}$$

**A.3.2.2 Instantiating goals** Given  $\Delta \vdash M : R_\Delta(\Gamma)$ , we define the *instantiation* of  $M$  to be a term  $\Delta, \Gamma \vdash \mathbf{InstGoal}_\Delta(\Gamma; C; S; M) : C [S]$ ; this term is obtained by recursion on  $\Gamma$ , applying the appropriate elimination rules. Locally, we write  $I_\Delta(\Gamma; M)$  for this term:

$$\begin{aligned} I_\Delta(\cdot; M) &= \text{app}_{\cdot.C \mid S}(M, \cdot) \\ I_\Delta(x : A, \Gamma; M) &= I_{\Delta, x:A}(\Gamma; \text{app}_{x:A.R_{\Delta, x:A}(\Gamma)}(M, x)) \\ I_\Delta(i : \mathbb{1}, \Gamma; M) &= I_{\Delta, i:\mathbb{1}}(\Gamma; \text{app}_{i.R_{\Delta, i:\mathbb{1}}(\Gamma)}(M, i)) \\ I_\Delta(\xi, \Gamma; M) &= I_{\Delta, \xi}(\Gamma; \text{out}_{\xi \rightarrow R_{\Delta, \xi}(\Gamma)}(M)) \end{aligned}$$

**A.3.2.3 Elaborating holes** We are now equipped to elaborate hole terms  $\boxed{?}$  in the three major *checking* modes.

$$\begin{array}{c} \frac{\Gamma \vdash \text{type}_*^{\text{pre}} \ni \text{a right} \rightsquigarrow [A] \quad \Sigma := \Sigma, \alpha : \mathbf{ReifyGoal}_{(\cdot)}(\Gamma; A; S)}{\Gamma \vdash \boxed{?} \Leftarrow \text{a} [S] \rightsquigarrow [\mathbf{InstGoal}_{(\cdot)}(\Gamma; A; S; \alpha)]} \\ \frac{\Gamma \vdash \text{type}_*^{\text{pre}} \ni \text{a}^+ \text{right} \rightsquigarrow [A] \quad \Sigma := \Sigma, \alpha : \mathbf{ReifyGoal}_{(\cdot)}(\Gamma; A; \cdot)}{\Gamma \vdash \boxed{?} \Leftarrow \text{a}^+ \rightsquigarrow [\mathbf{InstGoal}_{(\cdot)}(\Gamma; A; \cdot; \alpha)]} \\ \frac{\Sigma := \Sigma, \alpha : \mathbf{ReifyGoal}_{(\cdot)}(\Gamma; \mathbb{U}_{\alpha \sqcap \omega}^\kappa; \cdot)}{\Gamma \vdash \boxed{?} \Leftarrow \text{type}_\alpha^\kappa \rightsquigarrow [\mathbf{InstGoal}_{(\cdot)}(\Gamma; \mathbb{U}_{\alpha \sqcap \omega}^\kappa; \cdot; \alpha)]} \end{array}$$

**A.3.2.4 Guesses for synchronous boundary checks** At the boundary between boundary-sensitive checking and checking at positive types, there is a synchronous boundary

check; this poses a problem for interactive development because a partial proof may not yet satisfy the boundary. Inspired by the “guess” terms from Epigram and Idris’s development calculi, we include a special kind of hole construct which can be used to *protect* a partial proof term which does not yet satisfy the boundary constraint.

(constructors)  $\mathbb{M}$  ... |  $\boxed{\mathbb{M}}$

When one writes  $\boxed{\mathbb{M}}$  instead of  $\mathbb{M}$ , a new hole is created for the current goal and the body  $\mathbb{M}$  is elaborated relative to the empty boundary, and thrown away. This allows incremental development in the presence of boundaries, even when boundaries cannot be decomposed.

$$\frac{\begin{array}{l} \Gamma \vdash \text{type}_*^{\text{pre}} \ni \mathfrak{a}^+ \text{ right } \rightsquigarrow [A] \\ \Gamma \vdash \mathbb{M} \leftarrow \mathfrak{a}^+ \rightsquigarrow [M] \quad \overline{\Gamma, \xi \not\vdash M = N : A} \quad \Sigma := \Sigma, \alpha : \mathbf{ReifyGoal}_{(\cdot)}(\Gamma; A; \overline{\xi \rightarrow N}) \end{array}}{\Gamma \vdash \boxed{\mathbb{M}} \leftarrow \mathfrak{a}^+ [\overline{\xi \rightarrow N}] \rightsquigarrow [\mathbf{InstGoal}_{(\cdot)}(\Gamma; A; \overline{\xi \rightarrow N}; \alpha)]}$$



## Acronyms

- CHTT*** Computational Higher Type Theory (pp. 23, 26)  
***CoC*** Calculus of Constructions (p. 3)  
***CTT*** Computational Type Theory (pp. 25, 26, 35)  
***CuTT<sub>×</sub>*** Cartesian Cubical Type Theory (pp. 20, 23, 28, 39, 45–50, 52, 58)  
***CuTT<sub>DM</sub>*** De Morgan Cubical Type Theory (p. 52)  
***ELF*** Edinburgh Logical Framework (p. 9)  
***ETT*** Extensional Type Theory (pp. 5, 39)  
***GAT*** Generalized Algebraic Theory (p. 9)  
***ITT*** Intensional Type Theory (p. 5)  
***MLLF*** Martin-Löf's Logical Framework (p. 9)  
***MLTT*** Martin-Löf Type Theory (pp. 9, 24–26, 28, 39, 50)  
***pCIC*** Predicative Calculus of Inductive Constructions (p. 20)





# Bibliography

- [AAD07] Andreas Abel, Klaus Aehlig, and Peter Dybjer. “Normalization by Evaluation for Martin-Löf Type Theory with One Universe”. In: *Electron. Notes Theor. Comput. Sci.* 173 (Apr. 2007), pp. 17–39. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2007.02.025 (cit. on p. 6).
- [Abe09] Andreas Abel. “Extensional normalization in the logical framework with proof irrelevant equality”. In: *2009 Workshop on Normalization by Evaluation*. 2009 (cit. on p. 6).
- [Abe13] Andreas Abel. “Normalization by Evaluation: Dependent Types and Impredicativity”. Habilitation. Ludwig-Maximilians-Universität München, 2013 (cit. on pp. 6, 59).
- [ACK17] Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. *Two-Level Type Theory and Applications*. 2017. arXiv: 1705.03307 (cit. on p. 30).
- [ACP09] Andreas Abel, Thierry Coquand, and Miguel Pagano. “A Modular Type-Checking Algorithm for Type Theory with Singleton Types and Proof Irrelevance”. In: *Typed Lambda Calculi and Applications*. Ed. by Pierre-Louis Curien. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 5–19. ISBN: 978-3-642-02273-9 (cit. on pp. 3, 6, 30).
- [AGV72] Michael Artin, Alexander Grothendieck, and Jean-Louis Verdier. *Théorie des topos et cohomologie étale des schémas*. Séminaire de Géométrie Algébrique du Bois-Marie 1963–1964 (SGA 4), Dirigé par M. Artin, A. Grothendieck, et J.-L. Verdier. Avec la collaboration de N. Bourbaki, P. Deligne et B. Saint-Donat, Lecture Notes in Mathematics, Vol. 269, 270, 305. Berlin: Springer-Verlag, 1972 (cit. on p. 16).
- [AHH17] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. *Computational Higher Type Theory III: Univalent Universes and Exact Equality*. 2017. arXiv: 1712.01800 (cit. on pp. 6, 10, 23, 27, 28, 31, 34, 36, 39, 57).
- [AHH18] Carlo Angiuli, Kuen-Bang Hou (Favonia), and Robert Harper. “Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities”. In: *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. Ed. by Dan Ghica and Achim Jung. Vol. 119. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Ger-

- many: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 6:1–6:17. ISBN: 978-3-95977-088-0. DOI: 10.4230/LIPIcs.CSL.2018.6 (cit. on p. 30).
- [AJ19] Mathieu Anel and André Joyal. *Topo-logie*. Preprint. Mar. 2019. URL: <http://mathieu.anel.free.fr/mat/doc/Anel-Joyal-Topo-logie.pdf> (cit. on p. 36).
- [AK16] Thorsten Altenkirch and Ambrus Kaposi. “Normalisation by Evaluation for Dependent Types”. In: *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Ed. by Delia Kesner and Brigitte Pientka. Vol. 52. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 6:1–6:16. ISBN: 978-3-95977-010-1. DOI: 10.4230/LIPIcs.FSCD.2016.6 (cit. on p. 11).
- [All87] Stuart Frazier Allen. “A Non-Type-Theoretic Definition of Martin-Löf’s Types”. In: *Proceedings of the Symposium on Logic in Computer Science (LICS ’87)*. Ithaca, NY: IEEE, June 1987, pp. 215–221 (cit. on p. 36).
- [Alt+01] T. Altenkirch, P. Dybjer, M. Hofmann, and P. Scott. “Normalization by Evaluation for Typed Lambda Calculus with Coproducts”. In: *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 303–. DOI: 871816.871869 (cit. on pp. 14, 47).
- [AMB13] Guillaume Allais, Conor McBride, and Pierre Boutillier. “New Equations for Neutral Terms: A Sound and Complete Decision Procedure, Formalized”. In: *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*. DTP ’13. Boston, Massachusetts, USA: ACM, 2013, pp. 13–24. ISBN: 978-1-4503-2384-0. DOI: 10.1145/2502409.2502411 (cit. on p. 6).
- [Ane19] Mathieu Anel. *Descent & Univalence*. Talk given to the HoTTEST seminar. 2019. URL: <https://www.uwo.ca/math/faculty/kapulkin/seminars/hotttestfiles/Anel-2019-05-2-HoTTEST.pdf> (cit. on p. 36).
- [Ang+18a] Carlo Angiuli, Evan Cavallo, Kuen-Bang Hou (Favonia), Robert Harper, Anders Mörtberg, and Jonathan Sterling. *reddt: implementing Cartesian cubical type theory*. Dagstuhl Seminar 18341: Formalization of Mathematics in Type Theory. 2018. URL: <http://www.jonmsterling.com/pdfs/dagstuhl.pdf>.
- [Ang+18b] Carlo Angiuli, Evan Cavallo, Kuen-Bang Hou (Favonia), Robert Harper, and Jonathan Sterling. “The RedPRL Proof Assistant (Invited Paper)”. In: *Proceedings of the 13th International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice, LFMTP@FSCD 2018, Oxford, UK, 7th July 2018*. 2018, pp. 1–10. DOI: 10.4204/EPTCS.274.1 (cit. on p. 23).

- [Ang+19] Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Kuen-Bang Hou (Favonia), Robert Harper, and Daniel R. Licata. *Syntax and Models of Cartesian Cubical Type Theory*. Preprint. Feb. 2019. URL: <https://github.com/dlicata335/cart-cube> (cit. on pp. 28, 31, 32, 34, 52, 54).
- [Ang19] Carlo Angiuli. “Computational Semantics of Cartesian Cubical Type Theory”. PhD thesis. Carnegie Mellon University, 2019 (cit. on pp. 10, 23, 26, 27, 52, 58).
- [AÖV17] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. “Decidability of Conversion for Type Theory in Type Theory”. In: *Proc. ACM Program. Lang.* 2 (Dec. 2017), 23:1–23:29. ISSN: 2475-1421. DOI: 10.1145/3158111 (cit. on p. 12).
- [Asp+11] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. “The Matita Interactive Theorem Prover”. In: *Automated Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings*. 2011, pp. 64–69 (cit. on p. 65).
- [AVW17] Andreas Abel, Andrea Vezzosi, and Theo Winterhalter. “Normalization by Evaluation for Sized Dependent Types”. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017), 33:1–33:30. ISSN: 2475-1421 (cit. on p. 6).
- [Awo14] Steve Awodey. “Structuralism, Invariance, and Univalence”. In: *Philosophia Mathematica* 22.1 (2014), pp. 1–11 (cit. on p. 41).
- [Awo15] Steve Awodey. “Notes on cubical models of type theory”. 2015. URL: <http://www.github.com/awodey/math/blob/master/Cubical/cubical.pdf> (cit. on p. 31).
- [Awo18a] Steve Awodey. “A cubical model of homotopy type theory”. In: *Annals of Pure and Applied Logic* 169.12 (2018). Logic Colloquium 2015, pp. 1270–1294. ISSN: 0168-0072. DOI: 10.1016/j.apal.2018.08.002 (cit. on p. 32).
- [Awo18b] Steve Awodey. “Natural models of homotopy type theory”. In: *Mathematical Structures in Computer Science* 28.2 (2018), pp. 241–286. DOI: 10.1017/S0960129516000268 (cit. on p. 41).
- [Awo19] Steve Awodey. *A Quillen model structure on the category of cartesian cubical sets*. A talk presented at the First International Conference on Homotopy Type Theory (HoTT 2019). 2019. URL: <https://hott.github.io/HoTT-2019/conf-slides/Awodey.pdf> (cit. on p. 32).
- [Bau+16] Andrej Bauer, Gaëtan Gilbert, Philipp Haselwarter, Matija Pretnar, and Christopher A. Stone. “Design and Implementation of the Andromeda proof assistant”. TYPES. 2016. URL: <http://www.types2016.uns.ac.rs/images/abstracts/bauer2.pdf> (cit. on p. 25).

- [BCH14] Marc Bezem, Thierry Coquand, and Simon Huber. “A Model of Type Theory in Cubical Sets”. In: *19th International Conference on Types for Proofs and Programs (TYPES 2013)*. Ed. by Ralph Matthes and Aleksy Schubert. Vol. 26. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 107–128. ISBN: 978-3-939897-72-9. DOI: 10.4230/LIPIcs.TYPES.2013.107 (cit. on p. 32).
- [Bra+11] Edwin Brady, James Chapman, Pierre-Évariste Dagand, Adam Gundry, Conor McBride, Peter Morris, Ulf Norell, and Nicolas Oury. *An Epigram Implementation*. Feb. 2011 (cit. on pp. 6, 72).
- [Bra13] Edwin Brady. “Idris, a general-purpose dependently typed programming language: Design and implementation”. In: *Journal of Functional Programming* 23.5 (Sept. 2013), pp. 552–593 (cit. on pp. 6, 65, 72).
- [Car78] John Cartmell. “Generalised Algebraic Theories and Contextual Categories”. PhD thesis. Oxford University, Jan. 1978 (cit. on pp. 8, 9).
- [Car86] John Cartmell. “Generalised Algebraic Theories and Contextual Categories”. In: *Annals of Pure and Applied Logic* 32 (1986), pp. 209–243. ISSN: 0168-0072 (cit. on pp. 9, 39).
- [CFM16] Pierre-Louis Curien, Marcelo Fiore, and Guillaume Munch-Maccagnoni. “A Theory of Effects and Resources: Adjunction Models and Polarised Calculi”. In: *Proc. POPL*. 2016. DOI: 10.1145/2837614.2837652 (cit. on p. 49).
- [CFS88] Aurelio Carboni, Peter J. Freyd, and André Scedrov. “A Categorical Approach to Realizability and Polymorphic Types”. In: *Mathematical Foundations of Programming Language Semantics*. Ed. by M. Main, A. Melton, M. Mislove, and D. Schmidt. Vol. 298. Lectures Notes in Computer Science. New Orleans: Springer-Verlag, 1988, pp. 23–42 (cit. on p. 36).
- [CH09] Karl Cray and Robert Harper. *Mechanized Definition of Standard ML (alpha release)*. 2009. URL: <https://www.cs.cmu.edu/~cray/papers/2009/mldef-alpha.tar.gz> (cit. on pp. 6, 20).
- [CH19] Evan Cavallo and Robert Harper. “Higher Inductive Types in Cubical Computational Type Theory”. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 1:1–1:27. ISSN: 2475-1421. DOI: 10.1145/3290314 (cit. on pp. 28, 34, 56).
- [CHM18] Thierry Coquand, Simon Huber, and Anders Mörtberg. “On Higher Inductive Types in Cubical Type Theory”. In: *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. Oxford, United Kingdom: ACM, 2018, pp. 255–264. ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209197 (cit. on p. 56).

- [CHS19] Thierry Coquand, Simon Huber, and Christian Sattler. “Homotopy canonicity for cubical type theory”. In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. ISBN: 978-3-95977-107-8 (cit. on p. 11).
- [Coh+17] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. “Cubical Type Theory: a constructive interpretation of the univalence axiom”. In: *IfCoLog Journal of Logics and their Applications* 4.10 (Nov. 2017), pp. 3127–3169. URL: <http://www.collegepublications.co.uk/journals/ifcolog/?00019> (cit. on pp. 32, 34, 52, 54).
- [Coh+18] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. *cubicaltt: Experimental implementation of Cubical Type Theory*. 2018. URL: <https://github.com/mortberg/cubicaltt> (cit. on p. 36).
- [Con+86] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986. ISBN: 0-13-451832-2 (cit. on pp. 23, 25, 35, 36).
- [Coq+09] Thierry Coquand, Yoshiki Kinoshita, Bengt Nordström, and Makoto Takeyama. “A simple type-theoretic language: Mini-TT”. In: *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*. Ed. by Yves Bertot, Gérard Huet, Jean-Jacques Lévy, and Gordon Plotkin. Cambridge University Press, 2009, pp. 139–164. DOI: 10.1017/CB09780511770524.007 (cit. on p. 36).
- [Coq16] The Coq Development Team. *The Coq Proof Assistant Reference Manual*. 2016 (cit. on p. 2).
- [Coq17] Thierry Coquand. *Universe of Bishop sets*. Feb. 2017. URL: <http://www.cse.chalmers.se/~coquand/bishop.pdf> (cit. on p. 40).
- [Coq18] Thierry Coquand. *Canonicity and normalisation for Dependent Type Theory*. Oct. 2018. arXiv: 1810.09367 (cit. on pp. 11, 42).
- [Coq19a] Thierry Coquand. “Canonicity and normalization for dependent type theory”. In: *Theoretical Computer Science* 777 (2019). In memory of Maurice Nivat, a founding father of Theoretical Computer Science - Part I, pp. 184–191. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2019.01.015. arXiv: 1810.09367 (cit. on pp. 7, 11, 39).
- [Coq19b] Thierry Coquand. *Some constructive models of univalent type theory*. A talk presented at the First International Conference on Homotopy Type Theory (HoTT 2019). 2019. URL: <https://hott.github.io/HoTT-2019/conf-slides/Coquand.pdf> (cit. on p. 32).

- [Coq96] Thierry Coquand. “An algorithm for type-checking dependent types”. In: *Science of Computer Programming* 26.1 (1996), pp. 167–177. ISSN: 0167-6423. DOI: 10.1016/0167-6423(95)00021-6 (cit. on pp. 50, 70).
- [Cro93] R. L. Crole. *Categories for Types*. Cambridge Mathematical Textbooks. New York: Cambridge University Press, 1993. ISBN: 978-0-521-45701-9 (cit. on pp. 9, 10).
- [CZ84] Robert L. Constable and Daniel R. Zlatin. “The Type Theory of PL/CV3”. In: *ACM Trans. Program. Lang. Syst.* 6.1 (Jan. 1984), pp. 94–117. ISSN: 0164-0925. DOI: 10.1145/357233.357238 (cit. on p. 5).
- [Dyb96] Peter Dybjer. “Internal type theory”. In: *Types for Proofs and Programs: International Workshop, TYPES '95 Torino, Italy, June 5–8, 1995 Selected Papers*. Ed. by Stefano Berardi and Mario Coppo. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 120–134. ISBN: 978-3-540-70722-6 (cit. on p. 41).
- [FH10] Marcelo Fiore and Chung-Kil Hur. “Second-Order Equational Logic (Extended Abstract)”. English. In: *Computer Science Logic*. Ed. by Anuj Dawar and Helmut Veith. Vol. 6247. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 320–335. ISBN: 978-3-642-15204-7 (cit. on p. 8).
- [Fio02] Marcelo Fiore. “Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus”. In: *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*. PPDP '02. Pittsburgh, PA, USA: ACM, 2002, pp. 26–37. ISBN: 1-58113-528-9. DOI: 10.1145/571157.571161 (cit. on pp. 7, 9, 10, 13, 15).
- [FM10] Marcelo Fiore and Ola Mahmoud. “Second-Order Algebraic Theories”. In: *Mathematical Foundations of Computer Science 2010: 35th International Symposium, MFCS 2010, Brno, Czech Republic, August 23-27, 2010. Proceedings*. Ed. by Petr Hliněný and Antonín Kučera. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 368–380. ISBN: 978-3-642-15155-2 (cit. on p. 8).
- [Fre78] Peter Freyd. “On proving that  $\mathbf{1}$  is an indecomposable projective in various free categories”. Unpublished manuscript. 1978 (cit. on pp. 10, 39).
- [Gen35a] Gerhard Gentzen. “Untersuchungen über das logische Schließen I”. In: *Mathematische Zeitschrift* 39 (1935), pp. 176–210. URL: <http://eudml.org/doc/168546> (cit. on p. 13).
- [Gen35b] Gerhard Gentzen. “Untersuchungen über das logische Schließen II”. In: *Mathematische Zeitschrift* 39 (1935), pp. 405–431. URL: <http://eudml.org/doc/168556> (cit. on p. 13).

- [GSB19a] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. *blott: an experimental type checker for a modal dependent type theory*. GitHub repository. July 2019. URL: <https://github.com/jozefg/blott/> (cit. on p. 36).
- [GSB19b] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. “Implementing a Modal Dependent Type Theory”. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (July 2019), 107:1–107:29. ISSN: 2475-1421. DOI: 10.1145/3341711 (cit. on pp. 6, 12, 25).
- [GSB19c] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. *Normalization by Evaluation for Modal Dependent Type Theory*. Technical report. July 2019. URL: <http://www.jonmsterling.com/pdfs/modal-mltt-tr.pdf> (cit. on p. 12).
- [Har92] Robert Harper. “Constructing Type Systems over an Operational Semantics”. In: *Journal of Symbolic Computation* 14.1 (July 1992), pp. 71–84. ISSN: 0747-7171 (cit. on p. 36).
- [Heg69] Georg Wilhelm Friedrich Hegel. *Science of Logic*. Trans. by A. V. Miller. London: Allen & Unwin, 1969 (cit. on p. 7).
- [HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. “A Framework for Defining Logics”. In: *J. ACM* 40.1 (Jan. 1993), pp. 143–184. ISSN: 0004-5411. DOI: 10.1145/138027.138060 (cit. on p. 9).
- [Hic01] Jason J. Hickey. “The MetaPRL Logical Programming Environment”. PhD thesis. Ithaca, NY: Cornell University, Jan. 2001 (cit. on p. 23).
- [HL07] Robert Harper and Daniel R. Licata. “Mechanizing Metatheory in a Logical Framework”. In: *Journal of Functional Programming* 17.4-5 (July 2007), pp. 613–673. ISSN: 0956-7968. DOI: 10.1017/S0956796807006430 (cit. on p. 48).
- [HP05] Robert Harper and Frank Pfenning. “On Equivalence and Canonical Forms in the LF Type Theory”. In: *ACM Trans. Comput. Logic* 6.1 (Jan. 2005), pp. 61–101. ISSN: 1529-3785. DOI: 10.1145/1042038.1042041 (cit. on p. 12).
- [HS00] Robert Harper and Christopher Stone. “A Type-theoretic Interpretation of Standard ML”. In: *Proof, Language, and Interaction*. Ed. by Gordon Plotkin, Colin Stirling, and Mads Tofte. Cambridge, MA, USA: MIT Press, 2000, pp. 341–387. ISBN: 0-262-16188-5. URL: <http://dl.acm.org/citation.cfm?id=345868.345906> (cit. on pp. 6, 20, 50).
- [Hub18] Simon Huber. “Canonicity for Cubical Type Theory”. In: *Journal of Automated Reasoning* (June 13, 2018). ISSN: 1573-0670. DOI: 10.1007/s10817-018-9469-1 (cit. on pp. 6, 10, 27, 34, 39).

- [Ike19] Mirai Ikebuchi. “A Lower Bound of the Number of Rewrite Rules Obtained by Homological Methods”. In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 24:1–24:17. ISBN: 978-3-95977-107-8. DOI: 10.4230/LIPIcs.FSCD.2019.24 (cit. on p. 9).
- [Joh02] Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium: Volumes 1 and 2*. Oxford Logical Guides 43. Oxford Science Publications, 2002 (cit. on p. 9).
- [JT93] Achim Jung and Jerzy Tiuryn. “A new characterization of lambda definability”. In: *Typed Lambda Calculi and Applications*. Ed. by Marc Bezem and Jan Friso Groote. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 245–257. ISBN: 978-3-540-47586-6 (cit. on p. 15).
- [KHS19] Ambrus Kaposi, Simon Huber, and Christian Sattler. “Gluing for type theory”. In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. ISBN: 978-3-95977-107-8 (cit. on pp. 9, 11, 42, 49).
- [Law04] F. William Lawvere. “Functorial Semantics of Algebraic Theories”. In: *Reprints in Theory and Applications of Categories* 4 (2004), pp. 1–121. URL: <http://tac.mta.ca/tac/reprints/articles/5/tr5.pdf> (cit. on p. 1).
- [Law63] F. William Lawvere. “Functorial Semantics of Algebraic Theories”. PhD thesis. Columbia University, 1963 (cit. on pp. 7–9, 16).
- [Law94] F. William Lawvere. “Tools for the advancement of objective logic: closed categories and toposes”. In: *The logical foundations of cognition*. Ed. by John Macnamara and Gonzalo Reyes. 4. Oxford University Press on Demand, 1994 (cit. on p. 7).
- [LCH07] Daniel K. Lee, Karl Crary, and Robert Harper. “Towards a Mechanized Metatheory of Standard ML”. In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Nice, France: ACM, 2007, pp. 173–184. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190245 (cit. on pp. 20, 50).
- [Lev03] Paul Blain Levy. “Adjunction Models For Call-By-Push-Value With Stacks”. In: *Electronic Notes in Theoretical Computer Science* 69 (2003). CTCS’02, Category Theory and Computer Science, pp. 248–271. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)80568-1 (cit. on p. 49).



- [LH12] Daniel R. Licata and Robert Harper. “Canonicity for 2-dimensional Type Theory”. In: *SIGPLAN Not.* 47.1 (Jan. 2012), pp. 337–348. ISSN: 0362-1340. DOI: 10.1145/2103621.2103697 (cit. on p. 6).
- [Lic+18] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. “Internal Universes in Models of Homotopy Type Theory”. In: *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*. 2018, 22:1–22:17. DOI: 10.4230/LIPIcs.FSCD.2018.22 (cit. on p. 46).
- [LS09] F. William Lawvere and Stephen H. Schanuel. *Conceptual Mathematics: A First Introduction to Categories*. 2nd. New York, NY, USA: Cambridge University Press, 2009. ISBN: 0-521-89485-9 (cit. on p. 7).
- [LW15] Peter LeFanu Lumsdaine and Michael A. Warren. “The Local Universes Model: An Overlooked Coherence Construction for Dependent Type Theories”. In: *ACM Trans. Comput. Logic* 16.3 (July 2015), 23:1–23:31. ISSN: 1529-3785. DOI: 10.1145/2754931.
- [Mao37] Mao Tse-Tung. “On Practice: On the Relation Between Knowledge and Practice, Between Knowing and Doing”. In: *Selected Works of Mao Tse-tung*. Vol. 1. marxists.org, July 1937. URL: [https://www.marxists.org/reference/archive/mao/selected-works/volume-1/mswv1\\_16.htm](https://www.marxists.org/reference/archive/mao/selected-works/volume-1/mswv1_16.htm) (cit. on p. 23).
- [Mar70] Per Martin-Löf. *Notes on Constructive Mathematics*. Stockholm: Almqvist and Wiksell, 1970.
- [Mar75a] Per Martin-Löf. “About Models for Intuitionistic Type Theories and the Notion of Definitional Equality”. In: *Proceedings of the Third Scandinavian Logic Symposium*. Ed. by Stig Kanger. Vol. 82. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 81–109 (cit. on p. 10).
- [Mar75b] Per Martin-Löf. “An Intuitionistic Theory of Types: Predicative Part”. In: *Logic Colloquium ’73*. Ed. by H. E. Rose and J. C. Shepherdson. Vol. 80. Studies in Logic and the Foundations of Mathematics. Elsevier, 1975, pp. 73–118. DOI: 10.1016/S0049-237X(08)71945-1.
- [Mar79] Per Martin-Löf. “Constructive Mathematics and Computer Programming”. In: *6th International Congress for Logic, Methodology and Philosophy of Science*. Published by North Holland, Amsterdam. 1982. Hanover, Aug. 1979, pp. 153–175 (cit. on p. 36).
- [Mar84] Per Martin-Löf. *Intuitionistic type theory. Notes by Giovanni Sambin*. Vol. 1. Studies in Proof Theory. Bibliopolis, 1984, pp. iv+91. ISBN: 88-7088-105-9 (cit. on pp. 36, 41).
- [Mar86] Per Martin-Löf. *Amendment to Intuitionistic Type Theory*. Notes from a lecture given in Göteborg. 1986.

- [Mar87a] Per Martin-Löf. *The Logic of Judgements*. Workshop on General Logic, Laboratory for Foundations of Computer Science. Feb. 22, 1987.
- [Mar87b] Per Martin-Löf. “Truth of a Proposition, Evidence of a Judgement, Validity of a Proof”. In: *Synthese* 73.3 (1987), pp. 407–420.
- [Mar92] Per Martin-Löf. *Substitution calculus*. Notes from a lecture given in Göteborg. 1992.
- [Mar94] Per Martin-Löf. “Analytic and Synthetic Judgements in Type Theory”. In: *Kant and Contemporary Epistemology*. Ed. by Paolo Parrini. Dordrecht: Springer Netherlands, 1994, pp. 87–99. ISBN: 978-94-011-0834-8.
- [Mar96] Per Martin-Löf. “On the meanings of the logical constants and the justifications of the logical laws”. In: *Nordic Journal of Philosophical Logic* 1.1 (1996), pp. 11–60.
- [McB99] Conor McBride. “Dependently typed functional programs and their proofs”. PhD thesis. University of Edinburgh, 1999 (cit. on pp. 6, 65).
- [Mil+97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997 (cit. on p. 6).
- [MM16] Philippe Malbos and Samuel Mimram. “Homological computations for term rewriting systems”. In: *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*. Ed. by Delia Kesner and Brigitte Pientka. Vol. 52. Leibniz International Proceedings in Informatics (LIPIcs). Porto, Portugal: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 27:1–27:17. ISBN: 978-3-95977-010-1. URL: <https://hal.archives-ouvertes.fr/hal-01678175> (cit. on p. 9).
- [Mou+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. “The Lean Theorem Prover (System Description)”. In: *Automated Deduction - CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6 (cit. on pp. 2, 36).
- [MS93] John C. Mitchell and Andre Scedrov. “Notes on scoping and relators”. In: *Computer Science Logic*. Ed. by E. Börger, G. Jäger, H. Kleine Büning, S. Martini, and M. M. Richter. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 352–378. ISBN: 978-3-540-47890-4 (cit. on p. 9).
- [Nog02] Aleksey Nogin. “Quotient Types: A Modular Approach”. In: *Theorem Proving in Higher Order Logics*. Ed. by Victor A. Carreño, César A. Muñoz, and Sofiène Tahar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 263–280. ISBN: 978-3-540-45685-8 (cit. on p. 35).

- [Nor07] Ulf Norell. “Towards a practical programming language based on dependent type theory”. PhD thesis. SE-412 96 Göteborg, Sweden: Department of Computer Science and Engineering, Chalmers University of Technology, Sept. 2007 (cit. on pp. 2, 65).
- [NPS90] Bengt Nordström, Kent Peterson, and Jan M. Smith. *Programming in Martin-Löf’s Type Theory*. Vol. 7. International Series of Monographs on Computer Science. NY: Oxford University Press, 1990 (cit. on p. 9).
- [OP16] Ian Orton and Andrew M. Pitts. “Axioms for Modelling Cubical Type Theory in a Topos”. In: *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*. 2016, 24:1–24:19 (cit. on p. 46).
- [Pfe16] Frank Pfenning. *Lecture Notes on Sequent Calculus*. From a lecture given to 15-317 (Constructive Logic) at Carnegie Mellon University. 2016 (cit. on p. 13).
- [PV07] E. Palmgren and S. J. Vickers. “Partial Horn logic and cartesian categories”. In: *Annals of Pure and Applied Logic* 145.3 (2007), pp. 314–353. ISSN: 0168-0072. DOI: 10.1016/j.apal.2006.10.001 (cit. on p. 8).
- [Red18a] The RedPRL Development Team. *RedPRL – the People’s Refinement Logic*. 2018. URL: <http://www.redprl.org/> (cit. on p. 57).
- [Red18b] The RedPRL Development Team. *redtt*. 2018. URL: <http://www.github.com/RedPRL/redtt>.
- [Red18c] The RedPRL Development Team. *ua-beta.red*. The *redtt* mathematical library. 2018. URL: <https://github.com/RedPRL/redtt/blob/8eff9849b5e1e73287cd02c94a45d598d746a8a5/library/cool/ua-beta.red> (cit. on p. 34).
- [Red18d] The RedPRL Development Team. *univalence.red*. The *redtt* mathematical library. 2018. URL: <https://github.com/RedPRL/redtt/blob/8eff9849b5e1e73287cd02c94a45d598d746a8a5/library/prelude/univalence.red> (cit. on p. 34).
- [Rie19] Emily Riehl. *The equivariant uniform Kan fibration model of cubical homotopy type theory*. A talk presented at the First International Conference on Homotopy Type Theory (HoTT 2019). 2019 (cit. on p. 32).
- [Rij19] Egbert Rijke. “Classifying Types”. PhD thesis. Carnegie Mellon University, 2019. arXiv: 1906.09435 (cit. on p. 36).
- [RS17] Emily Riehl and Michael Shulman. “A type theory for synthetic  $\infty$ -categories”. In: *Higher Structures* 1 (2017) (cit. on pp. 29, 57).
- [SA20] Jonathan Sterling and Carlo Angiuli. “Gluing models of type theory along flat functors”. Unpublished draft. 2020 (cit. on pp. 10, 11, 43, 49).

- [SAG19] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. “Cubical Syntax for Reflection-Free Extensional Equality”. In: *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019)*. Ed. by Herman Geuvers. Vol. 131. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 31:1–31:25. ISBN: 978-3-95977-107-8. DOI: 10.4230/LIPIcs.FSCD.2019.31. arXiv: 1904.08562 (cit. on pp. 10, 11, 39, 41–43).
- [SAG20] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. “A cubical language for Bishop sets”. In preparation. 2020 (cit. on pp. 11, 46).
- [Sch00] Stephen H. Schanuel. “Objective number theory and the retract chain condition”. In: *Journal of Pure and Applied Algebra* 154.1 (2000). Category Theory and its Applications, pp. 295–298. ISSN: 0022-4049. DOI: [https://doi.org/10.1016/S0022-4049\(99\)00185-1](https://doi.org/10.1016/S0022-4049(99)00185-1) (cit. on p. 7).
- [SGR15] Jonathan Sterling, Daniel Gratzer, and Vincent Rahli. “JonPRL”. 2015. URL: <http://www.jonprl.org/> (cit. on p. 23).
- [SH00] Christopher A. Stone and Robert Harper. “Deciding Type Equivalence in a Language with Singleton Kinds”. In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Boston, MA, USA: Association for Computing Machinery, 2000, pp. 214–227. ISBN: 1-58113-125-9. DOI: 10.1145/325694.325724 (cit. on p. 3).
- [SH06] Christopher A. Stone and Robert Harper. “Extensional equivalence and singleton types”. In: *ACM Transactions on Computational Logic* (2006) (cit. on pp. 3, 30).
- [Shu13] Michael Shulman. *Scones, Logical Relations, and Parametricity*. Blog. 2013. URL: [https://golem.ph.utexas.edu/category/2013/04/scones\\_logical\\_relations\\_and\\_p.html](https://golem.ph.utexas.edu/category/2013/04/scones_logical_relations_and_p.html).
- [Shu15] Michael Shulman. “Univalence for inverse diagrams and homotopy canonicity”. In: *Mathematical Structures in Computer Science* 25.5 (2015), pp. 1203–1277. DOI: 10.1017/S0960129514000565 (cit. on p. 11).
- [Shu17] Michael Shulman. *Categorical Logic from a Categorical Point of View*. Lecture notes. 2017. URL: <https://github.com/mikeshulman/catlog>.
- [Shu19] Michael Shulman. *All  $(\infty, 1)$ -toposes have strict univalent universes*. Apr. 2019. arXiv: 1904.07004.
- [SS18] Jonathan Sterling and Bas Spitters. *Normalization by gluing for free  $\lambda$ -theories*. Sept. 2018. arXiv: 1809.08646 [cs.LO] (cit. on p. 9).
- [Ste18] Jonathan Sterling. *Algebraic Type Theory and Universe Hierarchies*. Dec. 2018. arXiv: 1902.08848 [math.LO] (cit. on p. 11).

- [Str91] Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results*. Cambridge, MA, USA: Birkhauser Boston Inc., 1991. ISBN: 0-8176-3594-7 (cit. on p. 6).
- [Tai67] W. W. Tait. “Intensional Interpretations of Functionals of Finite Type I”. In: *The Journal of Symbolic Logic* 32.2 (1967), pp. 198–212. ISSN: 00224812. URL: <http://www.jstor.org/stable/2271658> (cit. on p. 11).
- [Tay99] Paul Taylor. *Practical Foundations of Mathematics*. Cambridge studies in advanced mathematics. Cambridge, New York (N. Y.), Melbourne: Cambridge University Press, 1999. ISBN: 0-521-63107-6 (cit. on p. 9).
- [Uem19] Taichi Uemura. *A General Framework for the Semantics of Type Theory*. 2019. arXiv: 1904.04097 (cit. on pp. 8–10, 46).
- [Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: <https://homotopytypetheory.org/book>, 2013 (cit. on p. 36).
- [VMA19] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. “Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types”. In: *Proceedings of the 24th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’19. Boston, Massachusetts, USA: ACM, 2019. DOI: 10.1145/3341691 (cit. on p. 30).
- [Voe10] Vladimir Voevodsky. *The equivalence axiom and univalence models of type theory*. Talk at CMU. Feb. 4, 2010. URL: [https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/CMU\\_talk.pdf](https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/CMU_talk.pdf) (cit. on p. 34).
- [Wat+04] Kevin Watkins, Iliano Cervesato, Frank Pfenning, and David Walker. “A Concurrent Logical Framework: The Propositional Fragment”. In: *Types for Proofs and Programs*. Ed. by Stefano Berardi, Mario Coppo, and Ferruccio Damiani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 355–377. ISBN: 978-3-540-24849-1 (cit. on p. 47).
- [WB18] Paweł Wieczorek and Dariusz Biernacki. “A Coq Formalization of Normalization by Evaluation for Martin-Löf Type Theory”. In: *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA: ACM, 2018, pp. 266–279. ISBN: 978-1-4503-5586-5. DOI: 10.1145/3167091 (cit. on p. 12).
- [Wra74] Gavin Wraith. “Artin glueing”. In: *Journal of Pure and Applied Algebra* 4.3 (1974), pp. 345–348. ISSN: 0022-4049. DOI: [https://doi.org/10.1016/0022-4049\(74\)90014-0](https://doi.org/10.1016/0022-4049(74)90014-0) (cit. on p. 16).