DOCTORAT BRETAGNE LOIRE / MATHSTIC

Nantes Université

THÈSE DE DOCTORAT DE

NANTES UNIVERSITÉ

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité: *Informatique*

Par
Loïc PUJET

Computing with Extensionality Principles in Type Theory

Calculer avec des Principes d'Extensionnalité en Théorie des Types

Thèse présentée et soutenue à Nantes, le 13 décembre 2022
Unité de recherche : Laboratoire des Sciences du Numérique de Nantes

Rapporteurs avant soutenance :

Thierry COQUAND	Professor, University of Gothenburg
Conor MCBRIDE	Reader, University of Strathclyde

Composition du Jury :

Président :	Tom HIRSCHOWITZ	Directeur de Recherche, Université Savoie Mont-Blanc
Examinateurs :	Kuen-Bang HOU (Favonia)	Assistant Professor, University of Minnesota
	Ambrus KAPOSI	Associate Professor, Eötvös Loránd University
	Paige Randall NORTH	Honorary Research Fellow, University of Birmingham
	Jonathan STERLING	Postdoctoral Fellow, Aarhus University
Directeur de thèse :	Nicolas TABAREAU	Directeur de Recherche, INRIA Rennes

Thèse de Doctorat

Computing with Extensionality Principles in Dependent Type Theory

Loïc Pujet

13 décembre 2022

École Doctorale MathSTIC

À Vanille, mon lapin nain qui bien malgré lui, m'enseigna une des lois les plus fondamentales de l'univers: l'indifférence totale vis-à-vis de mon existence.

Acknowledgements

This thesis recounts part of my journey into the land of formal logic and computation. As is customary, the narrative is rather long-winded, and could certainly be described as fairly technical at times (it would not look very serious otherwise). But in the end, the past three years have been much more than a bunch of theorems and proofs, and I am grateful to all of you who played a part.

Thanks in particular to Nicolas, whom I had the pleasure to have as my advisor. For your availability, your insights, your patience, and your guidance. I had a lot of fun studying with you during these three years, and I learned a lot.

I am also grateful to Ambrus, Conor, Favonia, Jon, Paige, Thierry, and Tom for accepting to be part of my jury. It is both very flattering and a bit intimidating to be heard by so many people I admire.

Honestly, the entire Gallinette team was a blast to interact with during my stay in Nantes. I had many interesting conversations and pleasant chats with Assia, Étienne, Gaëtan, Guihem, Guillaume, Kazuhiko, Kenji, Marie, Matthieu (Piquerez), Matthieu (Sozeau), Pierre (Vial), Pierre-Marie and Yannick. I have particularly fond memories of my interactions with my senior PhD students, who taught me the ropes and made me feel at home: Ambroise for his humor and cheerfulness, Simon for making me discover Notre-Dame des Landes and the Jardin des Ronces, Théo for the leisurely walks along the Erdre, and Xavier for the board game nights. Meven, my thesis sibling, also played a huge part in making these three years unforgettable — thank you for being my protest comrade, for being my drinking comrade, and for everything you taught me!

But now, the time has come for the next generation of PhD students. I wish all the best to Martin, my favorite Toudou voice actor, to Pierre (Giraud) and his ability to know where is the closest piano at all times, and to Enzo and Hamza from the next office — hopefully you will get more time once I am not there to procrastinate with you anymore. And even though you are not yet PhD students, my wishes also apply to Yann and Arthur!

I am also grateful to the teaching team of the computer architecture course at the IUT de Nantes, and to the students I had the opportunity to teach. The teaching was certainly reciprocated, and I learned a lot from them.

Finally, I would like to thank Steve for welcoming me in CMU, and all of the people I had the occasion to meet there. Especially Anders, whom I had the pleasure to work with.

Outside of the world of academia, I am lucky to have found excellent friends in François, Vincent and Arthur, through ups and downs. They were such a great escape from working on my stubborn proofs! And during the monotonous times of lockdown and remote working, I could find a lot of fun and relief in a small, tight-knit online community — I am of course thinking of the Mauve crew, les grands philosophes du quotidien.

Last but not least, I want to thank Virginie Cornet-Berry and François Vellutini, two wonderful teachers whose passion for mathematics and love for transmission have accompanied me throughout my studies. And, of course, my family. Thank you mom, thank you dad, thank you sis. Love you all.

Contents

Introduction en Français | 1

This chapter is available in both French and English. If you would prefer to read the English version, you can jump to chapter 2.

1.1 L'égalité dans la Théorie des Types Intensionnelle

Si vous avez déjà utilisé un assistant à la preuve basé sur la théorie des types intensionnelle comme Coq, Agda ou Lean – et il y a de fortes chances que ce soit votre cas si vous êtes en train de lire cette thèse – alors vous savez sans doute que l'égalité peut être un véritable casse-tête.

D'une part, nous avons l'égalité définitionnelle (également appelée conversion), qui regroupe les équations que l'assistant à la preuve manipule silencieusement pour nous. Par exemple, les termes $2 + 2$ et 4 sont égaux définitionnellement, ce qui signifie que nous pouvons les utiliser indifféremment dans nos preuves sans avoir à nous soucier de prouver leur égalité à la main. Lorsque nous utilisons des types dépendants, cette quantité minimale d'automatisation est absolument vitale pour éviter de se retrouver à insérer des coercitions explicites à chaque fois que nous voulons, par exemple, utiliser un terme de type $\mathsf{Vector}\ (2 + 2)$ en tant que terme de type $\mathsf{Vector}\ 4$.

Dans un assistant à la preuve basé sur la théorie des ensembles ZF comme Metamath, le théorème $2 + 2 = 4$ demande une preuve. Et bien qu'elle soit élémentaire, une telle preuve demande une quantité surprenante de lemmes auxiliaires !

Mais malheureusement, l'assistant à la preuve n'est pas capable de nous mâcher le travail dans tous les cas. Les preuves mathématiques traitent fréquemment d'égalités entre des objets infinitaires tels que les fonctions et les ensembles et, bien évidemment, il n'existe pas d'algorithme capable de décider si deux fonctions de type $\mathbb{N} \to \mathbb{N}$ associent les mêmes images aux mêmes antécédents. Ainsi en pratique, l'égalité définitionnelle se limite aux égalités $\beta/\eta/\iota$, avec possiblement quelques extensions comme les règles de réécriture d'Agda [1].

[1]: Cockx et al. (2021), "The Taming of the Rew: A Type Theory with Computational Assumptions"

Puisque le système ne sera pas en mesure d'automatiser complètement toutes les égalités, nous avons besoin d'un moyen de raisonner à leur sujet. C'est pourquoi la théorie des types intensionnelle fournit une seconde notion d'égalité, appelée égalité *propositionnelle*. Contrairement à l'égalité définitionnelle, celle-ci est directement manipulable dans le langage de la théorie des types, et nous pouvons l'utiliser pour énoncer et prouver des théorèmes. Dans les trois principaux assistants à la preuve basés sur la théorie des types, l'égalité propositionnelle est un type inductif, d'après les travaux de Martin-Löf [2].

[2]: Martin-Löf (1975), "An Intiutionistic Theory of Types: Predicative Part"

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t =_A u} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{refl}_t : t =_A t}$$

L'égalité propositionnelle $t =_A u$ est un type, tandis que l'égalité définitionnelle $\Gamma \vdash t \equiv u : A$ est un jugement de typage.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash t : A \qquad \Gamma \vdash B : \Pi(x : A).\, t =_A x \to \mathbb{s} \qquad \Gamma \vdash u : B\ t\ \mathsf{refl}_t \qquad \Gamma \vdash t' : A \qquad \Gamma \vdash e : t =_A t'}{\Gamma \vdash \mathsf{J}(A, t, B, u, t', e) : B\ t'\ e}$$

$$\frac{[\ldots]}{\Gamma \vdash \mathsf{J}(A, t, B, u, t, \mathsf{refl}_t) \Rightarrow u : B\ t\ \mathsf{refl}_t}$$

Ces quelques règles suffisent à définir une relation d'équivalence qui contient l'égalité définitionnelle, peut être utilisée pour réécrire des termes et peut être utilisée pour obtenir un terme de type B à partir d'un terme de type A et d'une preuve de $A =_{\mathsf{Type}} B$.

Une différence importante avec l'égalité usuelle des mathématiques informelles est qu'en théorie des types, nous restons très attachés à la correspondance de Curry-Howard entre les preuves et les programmes : l'égalité est un type, et les preuves d'égalité sont des programmes. En tant que telles, les preuves d'égalité sont des valeurs de première classe au même titre que les entiers naturels ou les fonctions, et elles sont donc sujettes au raisonnement mathématique et peuvent être évaluées.

1.1.1 Types et Propositions

Bien que la possibilité d'évaluer les preuves soit un principe fondamental de la philosophie Curry-Howard, une preuve d'égalité n'a généralement pas un comportement calculatoire très intéressant. Le terme $\mathsf{J}(A, t, P, u, t', e)$ se contente d'attendre que e se réduise en une preuve par réflexivité, auquel cas t et t' sont convertibles, et le terme entier peut simplement être évalué en u.

L'extraction produit un programme dans un langage externe en effaçant une partie des types.

Par conséquent, les programmes extraits à partir de preuves qui utilisent du raisonnement équationnel ont tendance à passer beaucoup de temps à propager les preuves d'égalité, pour finir par les effacer lorsqu'elles ne sont plus nécessaires. Cette situation sous-optimale a conduit l'assistant à la preuve Coq à introduire une sorte spéciale Prop pour les types dont les preuves sont effacées lors de l'extraction [3]. En plaçant l'égalité propositionnelle dans Prop avec les autres contraintes logiques qui ne jouent aucun rôle calculatoire, Coq retrouve des performances raisonnables pour les programmes extraits.

[3]: Letouzey (2004), "Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq"

Techniquement, cette pratique va à l'encontre de la discipline de Curry-Howard, puisque Prop réintroduit une séparation entre les propositions et les données. Mais ne nous méprenons pas : les preuves de propositions jouent toujours leur rôle habituel dans les calculs de théorie des types, et en particulier, elles peuvent bloquer l'évaluation des termes ouverts. C'est seulement lorsque nous extrayons un programme *externe* que les propositions sont effacées.

L'assistant à la preuve Lean va encore un peu plus loin dans cette direction en introduisant la sorte sProp pour les *propositions strictes* [4]. Contrairement aux preuves des propositions en Coq, deux habitants quelconques d'une proposition stricte A sont toujours définitionnellement égaux (les propositions strictes sont dites *proof-irrelevant* en anglais).

[4]: de Moura et al. (2015), "The Lean Theorem Prover"

$$\frac{\Gamma \vdash A : \mathsf{sProp} \qquad \Gamma \vdash t, u : A}{\Gamma \vdash t \equiv u : A}$$

Mettre l'égalité propositionnelle dans sProp n'est pas aussi inoffensif qu'il n'y parait. En particulier, cela implique que toute preuve d'égalité propositionnelle entre deux termes convertibles est maintenant indiscernable d'une preuve par réflexivité, puisque les deux preuves ont le même type. En d'autres termes, Lean satisfait une version stricte du principe d'unicité des preuves d'égalité (abrégé en UIP pour *uniqueness of identity proofs*).

Avec ce principe, les preuves d'égalité non-réflexives ne sont plus capables de bloquer un calcul : le terme $\mathsf{J}(A, t, P, u, t, e)$ est convertible en $\mathsf{J}(A, t, P, u, t, \mathsf{refl}_t)$, qui devrait se réduire ! Par conséquent, l'éliminateur J de Lean se réduit dès qu'il est appliqué à deux termes définitionnellement égaux.

$$\text{J-conv}\quad \frac{[\ldots] \qquad \Gamma \vdash t \equiv t' : A}{\Gamma \vdash \mathsf{J}(A, t, B, u, t', e) \Rightarrow u : B\ t'\ e}$$

Mais bien que cette règle soit logiquement cohérente, Abel et Coquand ont montré qu'elle conduit à un échec de la normalisation pour les termes ouverts [5]. Et comme les algorithmes que nous utilisons pour vérifier la convertibilité reposent sur la normalisation, on en déduit que l'ajout de la règle J-conv casse notre procédure de décision pour l'égalité définitionnelle. Toutefois, comme l'ont remarqué Abel et Coquand, il n'est pas clair que cela provienne d'une incompatibilité fondamentale entre les propositions strictes de Lean et l'égalité propositionnelle, et qu'il n'existe pas une stratégie plus intelligente qui pourrait résoudre ce problème.

[5]: Abel et al. (2020), "Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality"

L'égalité définitionnelle de Lean est de toute manière indécidable pour des raisons orthogonales [6]. Remarquons que ça n'a pas empêché la communauté de Lean de développer une quantité impressionante de mathématiques.

1.1.2 L'Imprédicativité

En théorie des types dépendants, une sorte est dite *imprédicative* quand elle est close par produits dépendants indexés par n'importe quel type. Par exemple la sorte des propositions de Coq est imprédicative, car pour tout type A et toute fonction $B : A \to \mathsf{Prop}$ le produit dépendant $\Pi(x : A).\, B\ x$ est dans Prop.

L'imprédicativité introduit de l'autoréférence dans notre système de types : nous pouvons l'utiliser pour former une proposition P qui quantifie sur le type Prop de toutes les propositions, qui contient P. Un exemple classique est l'encodage imprédicatif de la proposition fausse, dont les habitants peuvent être éliminés dans n'importe quelle autre proposition :

$$\bot \ :=\ \Pi(X : \mathsf{Prop}).\, X \ :\ \mathsf{Prop}$$

Comparons cette situation avec la hiérarchie *prédicative* $(\mathsf{Type}_i)_{i \in \mathbb{N}}$ des types non-propositionnels. Dans le monde non-propositionnel, les produits dépendants sont forcés d'habiter un univers dont le niveau est à la fois plus grand que celui de leur domaine et plus grand que celui de leur codomaine : étant donné un type $A : \mathsf{Type}_i$ et une fonction $B : A \to \mathsf{Type}_j$, le produit dépendant $\Pi(x : A).\, B\ x$ atterrit dans l'univers $\mathsf{Type}_{\max(i,j)}$.

Dans l'assistant à la preuve Coq, les univers prédicatifs ne sont techniquement pas indexés par des entiers, mais le système maintient un graphe de contraintes de niveau d'univers qui remplit la même fonction.

Une hiérarche prédicative supprime les possibilités d'autoréférence : si nous essayons de reproduire l'encodage de la proposition fausse dans Type_0, le type résultant habite l'univers Type_1.

$$\bot' \;:=\; \Pi(X : \mathsf{Type}_0).\, X \;:\; \mathsf{Type}_1$$

Autoriser l'autoréférence revient à jouer avec le feu : les antinomies logiques dérivées du paradoxe de Russell restent tapies dans l'ombre, et pourraient nous attraper si nous sommes un peu trop gourmands avec nos règles logiques. Par exemple, le paradoxe de Berardi montre qu'avoir à la fois le tiers exclu et l'élimination large pour les booléens définis dans Prop est incohérent. L'imprédicativité joue également un rôle central dans le terme non-normalisant qu'Abel et Coquand ont obtenu à partir de la règle J-CONV [5].

[5]: Abel et al. (2020), "Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality"

En contrepartie de cette fragilité, l'imprédicativité augmente considérablement le pouvoir expressif de notre système logique, et nous permet de formuler des constructions mathématiques importantes comme le théorème du point fixe de Tarski ou des treillis complets non-triviaux [7]. De plus, certains théorèmes comme la normalisation de Système F ne peuvent tout simplement pas être prouvés dans une théorie prédicative.

[7]: Hur et al. (2013), "The Power of Parameterization in Coinductive Proof"

Les différents assistants à la preuve ont adopté des attitudes diverses en ce qui concerne l'imprédicativité. D'une part, Coq et Lean ont tous deux adopté l'imprédicativité pour leur sorte des propositions, qui cohabite avec la hiérarchie d'univers prédicatifs utilisée pour les types non propositionnels. [4, 8]. D'autre part, l'assistant à la preuve Agda a choisi de ne pas implémenter l'imprédicativité et utilise deux hiérarchies prédicatives à la place, une pour les propositions strictes et une pour les types [9].

Dans sa version initiale, Coq utilisait une sorte imprédicative pour les types non propositionnels, mais l'équipe de développement a préféré abandonner cette fonctionnalité pour rester compatible avec les mathématiques classiques.

[4]: de Moura et al. (2015), "The Lean Theorem Prover"
[8]: Coq Development Team (2016), *The Coq proof assistant reference manual*
[9]: Agda Development Team (2020), *Agda 2.6.1 documentation*

1.2 L'Extensionalité dans la Théorie des Types Intensionnelle

Malheureusement, l'égalité propositionnelle simple que nous avons définie avec un type inductif ne répond pas vraiment aux exigences du raisonnement mathématique. En mathématiques, deux fonctions sont considérées comme égales lorsqu'elles coïncident sur tous les antécédents – il s'agit du principe d'*extensionnalité des fonctions* – mais ce principe n'est pas démontrable pour l'égalité propositionnelle dans la théorie des types intensionnelle.

En particulier, il nous est impossible de montrer que les fonctions $\lambda\, n\,.\, n + 1$ et $\lambda\, n\,.\, 1 + n$ sont égales.

$$f =_{A \to B} g \quad \not\Leftarrow \quad \Pi(x : A).\, f\, x =_B g\, x$$

En fait, nous pouvons même prouver que toutes les preuves d'égalité propositionnelle dans un contexte clos se réduisent à l'égalité définitionnelle. C'est une conséquence directe de la normalisation des termes bien typés : la seule forme normale close pour une preuve d'égalité est refl_t, terme qui n'est typable que si les deux membres de l'égalité sont convertibles. Et comme nous l'avons expliqué précédemment,

l'égalité définitionnelle est incapable de gérer l'égalité naturelle des fonctions.

Développer des mathématiques complexes sans le principe d'extensionnalité des fonctions n'est pas une tâche facile. En conséquence, la communauté de la théorie des types a considéré plusieurs options pour retrouver une égalité plus conventionnelle.

Postuler des axiomes d'extensionnalité Le moyen le plus simple de récupérer un principe de raisonnement manquant est simplement de le postuler comme axiome. Par exemple, l'axiome functional_extensionality_dep est disponible dans la bibliothèque standard de Coq. L'utilisation d'axiomes a toutefois des inconvénients, car ils ne se comportent pas très bien vis-à-vis des propriétés calculatoires de la théorie des types intensionnelle : appliquer l'éliminateur J à une égalité obtenue via un axiome ne produira qu'un terme bloqué.

Remarquons qu'avec Coq et Lean, il est possible d'extraire un programme à partir d'une preuve qui postule l'axiome d'extensionnalité des fonctions, puisque cet axiome est une proposition et sera donc effacé lors de l'extraction.

Utiliser des sétoïdes Une technique standard pour récupérer les principes d'extensionnalité sans entrer en conflit avec la correspondance preuves-programmes est d'utiliser des *sétoïdes* [10], c'est-à-dire de remplacer les types par des couples $(|A|, e_A)$ d'un type $|A|$ et d'une relation d'équivalence e_A sur $|A|$ (l'*égalité sétoïdale* de A). Nous considérons alors uniquement les fonctions qui préservent l'égalité sétoïdale, et nous pouvons imposer l'extensionnalité dans l'égalité sétoïdale des types de fonctions.

[10]: Hofmann (1995), "Extensional concepts in intensional type theory"

Travailler avec des sétoïdes n'est toutefois pas très agréable, car nous devons accompagner chaque définition par des preuves bureaucratiques de préservation de l'égalité sétoïdale, alors même que toutes les constructions possibles en théorie des types respectent naturellement l'extensionnalité des fonctions [11]. En un mot, nous devons faire le travail d'un compilateur à la main. Coq implémente des tactiques pour traiter les sétoïdes plus efficacement, mais elles supportent difficilement le passage à l'échelle de développements conséquents – au point que la communauté utilise le terme *setoid hell* en référence à ces problèmes.

[11]: Altenkirch (1999), "Extensional Equality in Intensional Type Theory"

Changer la théorie Étant donné que les axiomes d'extensionnalité sont problématiques parce qu'ils bloquent le calcul, les théoriciens des types ont envisagé plusieurs manières d'étendre la théorie des types intensionnelle avec de nouvelles règles de calcul qui prennent en compte les axiomes souhaités. Les travaux dans cette direction les plus fructueux peuvent être divisés en deux groupes : d'une part ceux qui remplacent l'égalité propositionnelle inductive par une *égalité observationnelle*, et d'autre part ceux qui la remplacent par une *égalité univalente*.

1.2.1 L'Égalité Observationnelle

Les origines de l'égalité observationnelle remontent au modèle sétoïdal de la théorie des types, construit par Altenkirch suite aux explorations de Hofmann [10, 12]. En interprétant les types comme des sétoïdes,

Altenkirch a pu modéliser ITT+funext dans une théorie des types intensionnelle étendue avec des propositions strictes. Étant donné que ces propositions strictes n'entrent pas en conflit avec l'interprétation des preuves en tant que programmes, le modèle sétoïdal fournit un moyen d'évaluer les preuves qui utilisent l'axiome d'extensionnalité des fonctions en évaluant leur interprétation.

ITT+funext est une abréviation pour la théorie des types intensionnelle avec l'axiome d'extensionalité des fonctions.

L'ingrédient central du modèle sétoïdal est l'interprétation de l'univers. Construire un sétoïde qui contient tous les petits sétoïdes semble difficile en première approche, car la collection de tous les petits sétoïdes n'a pas d'égalité sétoïdale naturelle ; mais cela peut être résolu en utilisant un univers inductif-récursif de codes sur lequel on définit récursivement l'égalité sétoïdale ainsi qu'un opérateur de *type-casting*.

Une construction par induction-récursion permet de définir un type inductif A simultanément avec des fonctions définies par récursion sur A.

Suite à cette première étape, Altenkirch, McBride et Swierstra [13] ont développé la théorie des types observationnelle (OTT), une extension de la théorie des types intensionnelle qui ramène les idées du modèle sétoïdal dans le monde de la syntaxe : OTT incorpore un nouvel opérateur primitif appelé *égalité observationnelle* qui dote chaque type d'une structure de sétoïde définie par récursion sur l'univers. Grâce à cela, OTT implémente les principes d'UIP et d'extensionnalité des fonctions, et de plus Altenkirch *et al.* prouvent que la normalisation des termes d'OTT découle d'un résultat de normalisation conjecturé pour la théorie des types avec induction-récursion.

[13]: Altenkirch et al. (2007), "Observational equality, now!"

L'égalité observationnelle a connu un certain renouveau ces dernières années, avec notamment les travaux de Sterling *et al.* qui la revisitent sous l'angle de la théorie des types cubique [14] ou bien les travaux sur *setoid type theory* d'Altenkirch *et al.* [15]. Toutefois, la théorie des types observationnelle n'a pas encore atteint un niveau de maturité comparable à celui des théories des types univalentes. En particulier, il n'existe toujours pas de support pour l'égalité observationnelle dans les principaux assistants à la preuve, bien que des travaux aient été menés dans cette direction [16].

[14]: Sterling et al. (2022), "A Cubical Language for Bishop Sets"

[15]: Altenkirch et al. (2019), "Setoid type theory - a syntactic translation"

[16]: McBride (2020), *Epigram 2 - Autopsy, Obituary, Apology*

1.2.2 L'Égalité Univalente

La seconde famille de travaux, plus récente, remonte à la formulation de l'*axiome d'univalence* de Voevodsky [17, HoTT].

$$(A =_{\mathsf{Type}} B) \quad \simeq \quad (A \simeq B)$$

[17]: Kapulkin et al. (2018), "The simplicial model of Univalent Foundations (after Voevodsky)"

[HoTT]: Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

L'axiome d'univalence donne un sens nouveau à l'égalité entre les types : se donner un témoin d'égalité entre A et B revient à se donner un élément du type $A \simeq B$ des *isomorphismes* entre A et B. L'univalence peut être vue comme un principe d'extensionnalité pour l'univers des types, et l'ajouter à la théorie des types intensionnelle entraîne des conséquences remarquables. En particulier, l'axiome d'univalence implique le principe d'extensionnalité des fonctions.

Donner une bonne définition d'un isomorphisme entre deux types est plus subtil qu'il n'y paraît. Pour éviter trop de digressions dans notre introduction, nous supposerons que cette définition nous est donnée.

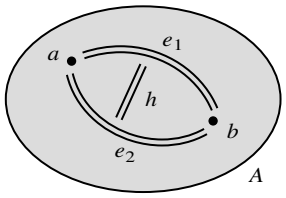
La Théorie des Types Homotopique L'axiome d'univalence s'éloigne de l'usage habituel de l'égalité en mathématiques, en ce sens que l'égalité ne peut plus être une proposition puisqu'elle contient désormais des informations essentielles pour le calcul. Par exemple, il existe deux

automorphismes distincts du type des booléens – l'un correspondant à l'identité et l'autre à la négation – et transporter un terme selon l'identité ne donne pas les mêmes résultats que le transport selon la négation.

$$\begin{aligned} \mathsf{J}(\mathsf{Type}, \mathsf{Bool}, \lambda X\,.\,X, \mathsf{true}, \mathsf{Bool}, \mathsf{id}) &=_{\mathsf{Bool}} \mathsf{true} \\ \mathsf{J}(\mathsf{Type}, \mathsf{Bool}, \lambda X\,.\,X, \mathsf{true}, \mathsf{Bool}, \mathsf{neg}) &=_{\mathsf{Bool}} \mathsf{false} \end{aligned}$$

Transporter une terme $t : A$ selon un isomorphisme $f : A \simeq B$ revient à appliquer l'isomorphisme à t, à une égalité propositionnelle près.

Ainsi lorsque nous utilisons l'axiome d'univalence, nous devons accepter que les types contiennent maintenant des informations importantes dans leurs types d'égalité. Mais ces types d'égalité contiennent également des informations dans leurs propres types d'égalité, et ainsi de suite. Nous nous retrouvons donc avec des tours infinies de types d'égalité potentiellement utiles pour le calcul, tous contenus dans la donnée d'un type.



$$\begin{aligned} &A : \mathsf{Type}_i \\ &a : A \\ &b : A \\ &e_1 : a =_A b \\ &e_2 : a =_A b \\ &h : e_1 =_{(a =_A b)} e_2 \end{aligned}$$

Figure 1.1: Représentation graphique d'un type avec des habitants et des égalités

Somme toute, avec l'axiome d'univalence les types ne se comportent plus tellement comme des ensembles. Voevodsky a remarqué qu'ils ressemblent bien plus à des ∞-*groupoïdes faibles*, un gadget catégorique qui joue un rôle important dans la théorie de l'homotopie, la branche des mathématiques qui étudie les espaces et leurs déformations. C'est le point de départ de la *théorie des types homotopique* (HoTT), une analogie profonde entre la théorie de l'homotopie et la théorie des types intensionnelle avec l'axiome d'univalence [HoTT].

Lorsque nous faisons de la théorie des types homotopique, nous considérons les types comme des espaces. Les habitants d'un type jouent le rôle de points, les égalités entre deux termes correspondent aux chemins entre les points, les égalités entre deux égalités sont des déformations continues entre les chemins correspondants, et ainsi de suite. Logiquement, les fonctions entre deux types sont continues (car elles préservent les égalités), et les isomorphismes correspondent aux équivalences d'homotopie.

La théorie des types homotopique étend également la notion de type inductif aux types inductifs supérieurs (abrégés en HIT pour *higher inductive types*), qui comportent des constructeurs de chemins/égalités en plus des constructeurs d'éléments. Les HIT fournissent des outils pour construire des espaces basiques comme le cercle, la sphère et le tore, mais également de quoi former des types quotients.

$$\begin{aligned} &\mathsf{Inductive}\ \ \mathsf{circle} : \mathsf{Type}_0\ \ := \\ &|\ \ \mathsf{base}\ \ :\ \ \mathsf{circle} \\ &|\ \ \mathsf{loop}\ \ :\ \ \mathsf{base} =_{\mathsf{circle}} \mathsf{base} \end{aligned}$$

$$\begin{aligned} &\mathsf{Inductive}\ \ \mathsf{quotient}\ (A : \mathsf{Type}_i)\ (R : A \to A \to \mathsf{Type}_i) : \mathsf{Type}_i\ \ := \\ &|\ \ \mathsf{emb}\ \ :\ \ A \to \mathsf{quotient}\ A\ R \\ &|\ \ \mathsf{quo}\ \ :\ \ \Pi(a\ b : A).\, R\ a\ b \to \mathsf{emb}\ a =_{\ldots} \mathsf{emb}\ b \end{aligned}$$

La définition du type circle peut être surprenante au premier abord, car elle ne ressemble absolument pas à l'espace attendu $\{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$. En effet les types en HoTT ne correspondent pas à des espaces *topologiques*, mais plutôt à une notion synthétique d'espaces construits à partir de points, de lignes, de faces de dimensions supérieures, *etc.* Ainsi le type circle est librement engendré par un point et un chemin qui relie ce point avec lui-même.

En utilisant des types inductifs supérieurs et l'axiome d'univalence, il devient possible de prouver un nombre surprenant de résultats classiques de la théorie de l'homotopie, reformulés pour parler de types et d'égalité : dans sa thèse, Brunerie a réussi à montrer que le quatrième groupe d'homotopie de la sphère tridimensionnelle a deux éléments [18]. Depuis lors, la communauté HoTT a développé des bibliothèques de mathématiques univalentes conséquentes, à l'aide des assistants à la preuve Coq [19] et Agda [20].

[18]: Brunerie (2016), "On the homotopy groups of spheres in homotopy type theory"

[19]: Bauer et al. (2017), "The HoTT Library: A Formalization of Homotopy Type Theory in Coq"

[20]: Agda-Unimath development team (2022), *The Agda-Unimath Library*

Toutefois, la théorie des types homotopique n'est pas une théorie qui *calcule*. L'axiome d'univalence est un axiome (comme le nom l'indique!), ce qui signifie que l'éliminateur J se contente de produire un terme bloqué lorsqu'il est appliqué à une égalité obtenue à partir d'un isomorphisme. Les types inductifs supérieurs ne sont pas compatibles avec le calcul non plus, car leurs principes d'élimination sont postulés comme des axiomes. Cette situation est d'autant plus décevante que les fonctionnalités introduites par HoTT semblent respecter les principes des mathématiques constructives.

La Théorie des Types Cubique Suite aux travaux de Voevodsky, la communauté des théoriciens des types s'est penchée sur la question du comportement calculatoire de l'égalité univalente.

En 2014, Bezem Coquand et Huber ont construit le premier modèle de l'axiome d'univalence dans une théorie des ensembles constructive [21]. Leur modèle interprète les types comme des *ensembles cubiques fibrants*, un objet combinatoire utilisé pour modéliser les ∞-groupoïdes faibles qui se prête bien à une présentation constructive. Le modèle de Bezem *et al.* s'est avéré être une manière fructueuse de réaliser le potentiel constructif de l'univalence, et d'évaluer les termes clos qui utilisent l'égalité univalente.

[21]: Bezem et al. (2014), "A Model of Type Theory in Cubical Sets"

Ce programme a donné naissance un an plus tard à la *théorie des types cubiques*, une théorie des types intensionnelle enrichie avec les idées apportées par les ensembles cubiques fibrants [22]. Ce nouveau système incorpore des primitives homotopiques directement dans les jugements de typage, et les équipe avec des règles de calcul adéquates. Grâce à cette structure supplémentaire, il devient possible de prouver l'univalence comme un théorème, ce qui fait de la théorie des types cubiques une théorie univalente sans axiome. Étant donné qu'elle satisfait une propriété de canonicité [23] et admet une fonction de normalisation pour les termes ouverts [24], la théorie des types cubique fournit une réponse solide à la question du comportement calculatoire de l'univalence. En plus de cela, elle peut être étendue par des types inductifs supérieurs avec des principes d'élimination et des règles de calcul naturels [25].

[22]: Cohen et al. (2015), "Cubical Type Theory: a constructive interpretation of the univalence axiom"

[23]: Huber (2019), "Canonicity for Cubical Type Theory"

[24]: Sterling et al. (2021), "Normalization for Cubical Type Theory"

[25]: Cavallo et al. (2019), "Higher Inductive Types in Cubical Computational Type Theory"

Dans les années qui ont suivi son introduction, la théorie des types cubiques a évolué en un domaine de recherche dynamique. De nombreuses variante du système original ont été étudiées sous un angle théorique [26, 27] et implémentées dans des assistants à la preuve [28, 29]. Parmi celles-ci, le mode `cubical` d'Agda est sans aucun doute l'implémentation qui a rencontré le plus franc succès, avec plusieurs projets de formalisation de grande envergure.

1.3 Contributions

Le Calcul des Constructions Observationnel Dans la première partie de cette thèse, nous étudions les propriétés méta-théoriques de l'égalité observationnelle. Pour ce faire, nous introduisons un calcul formel basé sur la théorie des types observationnelle d'Altenkirch *et al*, que nous appelons le calcul des constructions observationnel (abrégé en CC^{obs}). En plus des briques élémentaires de la théorie des types dépendants, CC^{obs} supporte une implémentation de l'égalité observationnelle et un opérateur primitif de *type-casting* à partir desquels nous pouvons dériver les principes d'unicité des preuves d'identité, d'extensionnalité des fonctions et d'extensionnalité des propositions. CC^{obs} permet également la formation de quotients d'un type par une relation d'équivalence stricte.

Nous équipons notre système formel d'une stratégie de réduction que nous utilisons pour prouver un théorème de normalisation, la canonicité des types de données de base et la décidabilité de la relation de typage. Ces résultats sont obtenus grâce à un modèle de normalisation, que nous construisons dans Agda en utilisant le cadre des relations logiques développé par Abel *et al.* [30]. La preuve de normalisation peut être consultée en ligne *[Everything.agda]*, et des liens vers le code seront fournis à intervalles réguliers dans les chapitres correspondants.

[30]: Abel et al. (2018), "Decidability of Conversion for Type Theory in Type Theory"

À notre connaissance, notre modèle apporte plusieurs innovations par rapport à la littérature existante sur les preuves de normalisation. La plus basique, mais aussi la plus importante, est que nous ne normalisons pas les preuves de propositions strictes, contrairement à l'approche adoptée par Gilbert et al *et al.* [6]. Cela nous permet non seulement de supporter les axiomes dans l'univers des propositions strictes, mais aussi de gérer l'imprédicativité sans avoir recours aux candidats de réductibilité usuels. Comme nous n'avons pas besoin des candidats de réductibilité, nous pouvons utiliser une relation logique *proof-irrelevant* pour notre preuve, ce qui nous permet d'ajouter des opérateurs non paramétriques tels que l'égalité observationnelle et le *type-casting*. En échange de cette flexibilité, notre refus de normaliser les preuves des propositions a pour conséquence que la normalisation n'implique pas directement la canonicité, et qu'une preuve de cohérence séparée est nécessaire pour obtenir la canonicité – comme suggéré par McBride *et al.* dans leur article sur la théorie des types observationnelle [31]. Nous comblerons cette lacune en construisant un modèle de CC^{obs} en théorie des ensembles.

[6]: Gilbert et al. (2019), "Definitional Proof-Irrelevance without K"

[31]: Altenkirch et al. (2007), "Observational Equality, Now!"

Ce travail résout plusieurs problèmes qui étaient encore ouverts à notre connaissance : tout d'abord, nous obtenons une preuve complète de la normalisation et de la décidabilité pour la théorie des types observationnelle, déjà conjecturée par Altenkirch *et al.* De plus, l'égalité de CC^{obs} vit dans une sorte imprédicative de propositions strictes, tout comme l'égalité de Lean. Nous montrons que nous pouvons interpréter l'éliminateur J de Lean dans une extension de CC^{obs}, tout en préservant la normalisation et la décidabilité du typage. Ceci répond à la question d'Abel et Coquand sur la compatibilité de l'imprédicativité et de l'égalité en tant que proposition stricte [5].

[5]: Abel et al. (2020), "Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality"

Vers une Traduction Cubique Dans la deuxième partie, nous étudions la possibilité d'une traduction de HoTT dans notre calcul des constructions observationnel. Puisque CC^{obs} est un système calculatoire, une telle traduction fournirait une nouvelle façon de calculer avec l'axiome d'univalence.

Le point de départ de cette partie est la traduction dite "préfasciste" de Pédrot qui interprète les types de la théorie de Martin-Löf comme des préfaisceaux dans une théorie des types intensionnelle avec des propositions strictes [32]. Les préfaisceaux sont une construction catégorique qui généralise de nombreux objets algébriques, y compris les ensembles cubiques que Coquand *et al.* ont utilisé pour modéliser l'univalence. Cependant il manque un ingrédient important à la traduction préfasciste comparé au modèle de Coquand *et al.* : les *structures de fibration*.

[32]: Pédrot (2020), "Russian Constructivism in a Prefascist Theory"

Nous ajoutons deux nouvelles pierres à cet édifice : dans un premier temps, nous présentons une dissection de la construction préfasciste de Pédrot, en expliquant comment elle contourne les difficultés bien connues des préfaisceaux en théorie des types intensionnelle. Dans un second temps, nous expliquons comment étendre la traduction préfasciste avec une notion de structure de fibration qui devrait permettre une interprétation de l'axiome d'univalence.

Homotopie Synthétique en Théorie des Types Cubique Enfin, dans la troisième partie de cette thèse, nous cherchons à montrer les avantages pratiques du calcul dans l'utilisation d'un système univalent. Pour ce faire, nous formalisons plusieurs résultats classiques de la théorie de l'homotopie en théorie des types cubiques, et nous comparons les preuves obtenues avec des formalisations analogues qui postulent l'univalence et les HITs en tant qu'axiomes. Notre panel de résultats inclut :

- L'équivalence entre le tore et le produit cartésien de deux cercles, avec un calcul de leurs groupes fondamentaux respectifs.
- L'équivalence entre une définition directe des sphères de basses dimensions et une définition alternative à base de suspensions itérées.
- La définition du pushout homotopique avec une preuve directe du lemme "3×3".
- La définition du joint de deux type et une preuve d'associativité. Avec ces éléments, nous obtenons deux preuves différentes que la sphère $\mathbb{S}^3$ est équivalente au joint de deux cercles.
- Une définition de la fibration de Hopf et une preuve que son espace total est équivalent à la sphère $\mathbb{S}^3$.

Ce travail a été formalisé dans l'assistant à la preuve Agda, et a été incorporé dans la bibliothèque standard de Cubical Agda [33].

[33]: Agda-Cubical development team (2022), *A Standard Library for Cubical Agda*

1.3.1 Publications

Cette thèse s'appuie sur trois articles revus et publiés dans des conférences internationales:

- Le travail sur le développement de l'homotopie synthétique en théorie cubique des types a été présenté dans un article co-écrit avec Anders Mörtberg et publié à CPP'20 [34].
- La construction d'une relation logique pour une version prédicative de la théorie des types observationnelle a été réalisée en collaboration avec Nicolas Tabareau, et a fait l'objet d'une publication à POPL'22 [35].
- Le travail sur l'imprédicativité et son interaction avec l'égalité observationnelle a également été réalisé en collaboration avec Nicolas Tabareau, et a été accepté à POPL'23 [36].

[34]: Mörtberg et al. (2020), "Cubical Synthetic Homotopy Theory"

[35]: Pujet et al. (2022), "Observational Equality: Now For Good"

[36]: Pujet et al. (2023), "Impredicative Observational Equality"

En plus de cela, la traduction cubique étudiée en partie 2 a été l'objet d'un résumé étendu presenté à ICMS'20.

1.4 Théories et Méta-Théories

Dans les chapitres suivants, nous manipulerons un certain nombre de systèmes formels – principalement des théories des types dépendants, mais également quelques théories du premier ordre. Étant donné que les conventions de nommages des théories des types semblent varier significativement d'un auteur à l'autre, nous fixons notre propre convention :

MLTT	La Théorie des Types Intensionnelle de Martin-Löf désigne un système avec des produits dépendents, une hiérarchie prédicative d'univers et une *certaine quantité* de types inductifs [2]. Nous demeurons intentionnellement vagues au sujet des types inductifs. Lorsque les inductifs jouent un rôle important, nous utiliserons des notations plus explicites que nous introduirons au besoin.
pCIC	Le Calcul prédicatif des Constructions Inductives étend MLTT avec une sorte imprédicative Prop, qui supporte l'élimination large des types inductifs sous-singletons [37].
$\mathsf{CC}^{\mathsf{obs}}$ / $\mathsf{CC}^{\mathsf{obs+}}$	Le Calcul des Constructions Observationnel désigne notre implémentation de la théorie des types observationnelle de McBride *et al.* Une présentation détaillée est donnée dans le chapitre 3. Le système étendu $\mathsf{CC}^{\mathsf{obs+}}$ ajoute une règle supplémentaire pour le calcul du *type-casting* sur les preuves d'égalité réflexives, décrite dans le chapitre 5.

Comme la majeure partie de cette thèse traite des propriétés méta-théoriques des théories des types, nous les traiterons souvent comme des objets mathématiques, c'est-à-dire comme une collection de lexèmes et de règles de typage. Mais les théories des types sont également des théories mathématiques à part entière, que nous pouvons utiliser pour prouver des théorèmes.

Et en effet, nous développerons une certaine quantité de mathématiques à l'intérieur de théories des types, en adoptant généralement un style informel pour des questions de lisibilité. Lorsque nous utilisons la théorie des types comme méta-théorie, nous adopterons la syntaxe et le schéma de couleurs de l'assistant à la preuve Agda, pour la simple raison que toutes nos formalisations sont effectuées en Agda. Si aucun

code en Agda n'apparaît, il est raisonnable de supposer que nous travaillons sans présupposer une méta-théorie précise, comme c'est le cas dans la majorité de la littérature mathématique.

Introduction in English 2

2.1 Equality in Intensional Type Theory

If you have ever used a proof assistant based on intensional type theory such as Coq, Agda or Lean—and there is a good chance that you have if you are reading this thesis—then you probably know that equality can be a real headache.

On the one hand we have the *definitional* equality (also called conversion), which records the equations that the proof assistant handles silently for us. For instance, the terms $2 + 2$ and 4 are definitionally equal, meaning that we can use them interchangeably in our proofs without having to worry about proving their equality by hand. In a dependently typed setting, this minimal amount of automation is absolutely vital; lest we have to insert explicit coercions every time we want to say, use a term of type Vector $(2+2)$ as a term of type Vector 4.

In proof assistants based on ZF set theory such as Metamath, the theorem $2 + 2 = 4$ does requires a proof. Although it is elementary, this proof involves a surprising amount of auxiliary lemmas!

But unfortunately not everything is a natural number, and mathematical proofs frequently deal with equalities between infinitary objects such as functions and sets. And of course there is no algorithm that can decide whether two functions of type $\mathbb{N} \to \mathbb{N}$ are pointwise equal. Thus in practice, the definitional equality is limited to $\beta/\eta/\iota$ equalities, with perhaps some extensions such as Agda's rewrite rules [1].

[1]: Cockx et al. (2021), "The Taming of the Rew: A Type Theory with Computational Assumptions"

Since the system will not be able to fully automate all equalities away, we need facilities to reason about them. This is why intensional type theories provide a second notion of equality, called the *propositional* equality. Contrary to the definitional equality, this one is available internally in the language of type theory, so that we may use it to state and prove theorems. In all three major proof assistants based on type theory, the propositional equality is implemented as an inductive type, after the pioneer work of Martin-Löf [2].

[2]: Martin-Löf (1975), "An Intiutionistic Theory of Types: Predicative Part"

The propositional equality $t =_A u$ is a type, while the definitional equality $\Gamma \vdash t \equiv u : A$ is a typing judgment.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t =_A u} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{refl}_t : t =_A t}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \qquad \Gamma \vdash t : A \qquad \Gamma \vdash B : \Pi(x : A).\, t =_A x \to \mathbb{s} \\ \Gamma \vdash u : B\ t\ \mathsf{refl}_t \qquad \Gamma \vdash t' : A \qquad \Gamma \vdash e : t =_A t'\end{array}}{\Gamma \vdash \mathsf{J}(A, t, B, u, t', e) : B\ t'\ e}$$

$$\frac{[\ldots]}{\Gamma \vdash \mathsf{J}(A, t, B, u, t, \mathsf{refl}_t) \Rightarrow u : B\ t\ \mathsf{refl}_t}$$

These few rules are enough to define an equivalence relation that contains the definitional equality, can be used to rewrite terms and can be used to coerce between equal types.

An important difference with the common practice of mathematics is the attachment to the Curry-Howard correspondence between proofs and programs: equality is a *type*, and equality proofs are *programs*. As

such, equality proofs are first class values like the natural numbers or the functions, and we can reason about them or evaluate them just as well.

2.1.1 Types and Propositions

While the possibility to evaluate proofs is a core tenet of the Curry-Howard doctrine, the typical equality proof does not exhibit a very interesting computational behavior. The term $\mathsf{J}(A, t, P, u, t', e)$ just waits for e to reduce to a proof by reflexivity, in which case t and t' are convertible and the whole term can simply evaluate to u.

As a consequence, the programs extracted from proofs that use equational reasoning tend to spend a lot of time passing around equality proofs just to end up discarding them. This less-than-ideal state of affairs led the Coq proof assistant to introduce a special sort Prop for types whose proofs are to be erased during program extraction [3]. By putting the propositional equality in Prop along with the other logical constraints that do not play any role in computation, Coq recovers reasonably good performances for extracted programs.

Extraction produces a program in an external programming language by erasing some type information.

[3]: Letouzey (2004), "Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq"

Technically this is a bit of an infrigement of the Curry-Howard discipline, as Prop effectively re-introduces a separation between propositions and data. But make no mistake: proofs of propositions still play their normal role in computations, and in particular they may block the evaluation of open terms. It is only when we extract an *external* program that propositions become irrelevant.

The Lean proof assistant goes a step further and introduces the sort sProp for *strict propositions* [4]. Contrary to Coq's propositions, the strict propositions are proof-irrelevant, meaning that any two inhabitants of a strict proposition A are definitionally equal simply by virtue of having type A.

$$\frac{\Gamma \vdash A : \mathsf{sProp} \qquad \Gamma \vdash t, u : A}{\Gamma \vdash t \equiv u : A}$$

The Lean community uses the name Prop for the sort of strict propositions, but we will use sProp to avoid confusion with non-strict propositions.

[4]: de Moura et al. (2015), "The Lean Theorem Prover"

Putting the propositional equality in sProp is not exactly innocuous. It means that any proof of propositional equality between two convertible terms is now undistinguishable from a proof by reflexivity since they have the same type. In other words Lean satisfies a strict version of the principle of *uniqueness of identity proofs* (UIP).

Now non-reflexive equality proofs can't block computation anymore: the term $\mathsf{J}(A, t, P, u, t, e)$ is convertible to $\mathsf{J}(A, t, P, u, t, \mathsf{refl}_t)$, which should reduce! As a consequence, the J eliminator of Lean reduces whenever it is applied to definitionally equal terms.

$$\text{J-conv}\ \frac{[\ldots] \qquad \Gamma \vdash t \equiv t' : A}{\Gamma \vdash \mathsf{J}(A, t, B, u, t', e) \Rightarrow u : B\ t'\ e}$$

But while this rule is logically sound, Abel and Coquand showed that it leads to a failure of normalization for open terms [5]. Since the algorithms we use to check for convertibility rely on normalization,

It is known that Lean's definitional equality is undecidable for other, independent reasons [6]. But this certainly did not prevent the Lean community from developing a large amount of mathematics.

it follows that adding rule J-CONV breaks our decision procedures for the definitional equality. As noted by Abel and Coquand, it is not clear whether this stems from a fundamental incompatibility between impredicative strict propositions and the propositional equality, or if a more clever strategy could solve this problem.

2.1.2 Impredicativity

In dependent type theory, a sort is said to be *impredicative* if it is closed under dependent products over any index type. For instance the sort of Coq propositions is impredicative, because for all types A and functions $B : A \to \mathsf{Prop}$ the dependent product $\Pi(x : A).\, B\, x$ is in Prop.

Impredicativity allows some amount of self-reference into the type system: we can use it to form a proposition P which quantifies over the type Prop of all propositions, a type that contains P. A classic example is the impredicative encoding of the false proposition, whose inhabitants can be eliminated into any other proposition:

$$\bot \;:=\; \Pi(X : \mathsf{Prop}).\, X \;:\; \mathsf{Prop}$$

Compare this situation with the *predicative* hierarchy $(\mathsf{Type}_i)_{i \in \mathbb{N}}$ of non propositional types. In the non-propositional world, dependent products are forced to inhabit a universe level that is higher than both the level of their domain and the level of their codomain: given a type $A : \mathsf{Type}_i$ and a function $B : A \to \mathsf{Type}_j$, their dependent product $\Pi(x : A).\, B\, x$ is in $\mathsf{Type}_{\max(i,j)}$.

In the case of Coq the predicative universes are technically not indexed by integers, but the system maintains a graph of level constraints that effectively does the same job.

Predicativity removes the possibility for self-reference: if we try to replicate the encoding of the false proposition in Type_0, the resulting type lands in the next universe.

$$\Pi(X : \mathsf{Type}_0).\, X \;:\; \mathsf{Type}_1$$

Playing with self-reference is always a dangerous game: logical antinomies born of Russell's paradox are lurking in the shadows, and might catch us if we get a bit too greedy with our logical principles. For instance, Berardi's paradox shows that having excluded middle and booleans with large elimination in Prop is inconsistent. Impredicativity also plays a central role in the non-normalizing term that Abel and Coquand derived from rule J-CONV [5].

In counterpart to this fragility, impredicativity greatly increases the expressive power of our logical system, and allows us to formulate important mathematical constructions such as Tarski's fixed point theorem or nontrivial complete lattices [7]. Plus some theorems such as the normalization of System F outright cannot be proved in a predicative theory.

Different proof assistants have adopted different attitudes toward impredicativity. On the one hand, Coq and Lean both have embraced impredicativity for their sort for propositions, which cohabits with a predicative universe hierarchy used for the computationally relevant types [4, 8]. On the other hand, the Agda proof assistant prefers not

[5]: Abel et al. (2020), "Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality"

[7]: Hur et al. (2013), "The Power of Parameterization in Coinductive Proof"

Back in the day the Coq proof assistant used impredicative sorts for both computational data and proofs, but the development team switched to a theory that is more compatible with classical mathematics.

[4]: de Moura et al. (2015), "The Lean Theorem Prover"

[8]: Coq Development Team (2016), *The Coq proof assistant reference manual*

to support impredicativity at all and uses two predicative hierarchies instead, one for strict propositions and one for types [9].

[9]: Agda Development Team (2020), *Agda 2.6.1 documentation*

2.2 Recovering Extensionality in Intensional Type Theory

The basic propositional equality that we defined as an inductive type does not quite meet the standards of mathematical reasoning. In mathematics, two functions are considered equal when they agree on all inputs—this is the principle of *function extensionality*—but this principle is not derivable for the propositional equality in intensional type theory.

In particular, it is not possible to prove that the functions $\lambda\ n\ .\ n + 1$ and $\lambda\ n\ .\ 1 + n$ are equal.

$$f =_{A \to B} g \quad \not\Leftarrow \quad \Pi(x : A).\, f\, x =_B g\, x$$

In fact, we can show that all proofs of propositional equality in an empty context reduce to the definitional equality. This is a straightforward consequence of the normalization of well-typed terms: the sole closed normal form for an equality proof is refl_t, which only type-checks if the two endpoints are convertible. And as we explained earlier, the definitional equality does not handle pointwise equality of functions.

Developing sophisticated mathematics without the principle of function extensionality is not an easy task. Consequently, the community has explored several options to recover a more conventional equality throughout the years.

Extensionality axioms The most obvious way to recover a missing reasoning principle is simply to postulate it as an axiom. For instance, the axiom functional_extensionality_dep is available in Coq's standard library. The downside of using axioms is that they do not play so well with the computational properties of intensional type theory: applying the J eliminator to an equality obtained *via* an axiom will only result in a stuck term.

Note that in Coq and Lean it is still possible to extract a program from a proof that postulates function extensionality, because the axiom is erased during extraction.

Setoids A standard technique to recover extensionality principles without breaking the proofs-as-programs correspondance is to use *setoids* [10], *i.e.* to replace types with pairs $(|A|, e_A)$ of a carrier type $|A|$ and an equivalence relation e_A on $|A|$ (the *setoid equality* of A). We then restrict our attention to functions that preserve the setoid equality, and we bake function extensionality into the setoid equality of function types.

[10]: Hofmann (1995), "Extensional concepts in intensional type theory"

Working with setoids is not exactly pleasant, as we need to supplement every definition with bureaucratic proofs of preservation of the setoid equalities, even though all available constructs in type theory do respect function extensionality [11]. Basically, we need to do the work of a compiler by hand. The Coq proof assistant provides some automation to deal with setoids through tactics, but these solutions do not scale painlessly to large developments—to the point that the community has coined the term *setoid hell* to refer to these issues.

[11]: Altenkirch (1999), "Extensional Equality in Intensional Type Theory"

Alternative type theories Since extensionality axioms are problematic because they block computation, type theorists have explored numerous ways to extend intensional type theory with new computational rules that handle the desired axioms. The most successful lines of work can be roughly divided into two groups: the ones that replace the inductive propositional equality with an *observational equality*, and the ones that replace it with a *univalent equality*.

2.2.1 Observational Equality

The observational equality has its roots in the setoid model of type theory, developed by Altenkirch after some explorations by Hofmann [10, 12]. By interpreting types as setoids, Altenkirch was able to model ITT+funext in an intensional type theory extended with strict propositions. Since these strict propositions do not conflict with the proofs-as-programs interpretation, the setoid model provides a way to evaluate proofs that use a function extensionality axiom by evaluating their interpretation.

ITT+funext stands for intensional type theory extended with function extensionality.

The central ingredient of the setoid model is the interpretation of the universe. Constructing a setoid of small setoids seemed difficult at first, because the collection of all small setoids does not have a natural setoid equality; but this can be solved by using an inductive-recursive universe of codes on which the setoid equality and type coercion operators are defined by recursion.

Induction-recursion is a powerful principle that allows us to define an inductive type A simultaneously with functions defined by recursion on A.

Following this first step, Altenkirch, McBride and Swierstra [13] developed observational type theory (OTT), an extension of intensional type theory that brings the main insights of the setoid model back to the world of syntax: OTT introduces a new primitive operator called the *observational equality* that equips every type with a setoid structure defined by recursion on the universe. The result is a type theory that supports the principles of UIP and function extensionality, and Altenkirch *et al.* prove that normalization of OTT terms follows from a conjectured normalization result for type theory with induction-recursion.

[13]: Altenkirch et al. (2007), "Observational equality, now!"

The observational equality has seen some new and exciting developments in recent years, such as the work of Sterling *et al.* who revisit it under the lens of cubical type theory [14] or the setoid type theory of Altenkirch *et al.* [15]. Still, observational type theory has yet to reach a level of maturity comparable to that of univalent type theories. In particular, there is still no support for the observational equality in major proof assistants, despite some significant work in that direction [16].

[14]: Sterling et al. (2022), "A Cubical Language for Bishop Sets"

[15]: Altenkirch et al. (2019), "Setoid type theory - a syntactic translation"

[16]: McBride (2020), *Epigram 2 - Autopsy, Obituary, Apology*

2.2.2 Univalent Equality

The other line of work is more recent and takes its roots in Voevodsky's *univalence axiom* [17, HoTT].

$$(A =_{\mathsf{Type}} B) \quad \simeq \quad (A \simeq B)$$

[17]: Kapulkin et al. (2018), "The simplicial model of Univalent Foundations (after Voevodsky)"
[HoTT]: Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

The univalence axiom gives a new meaning to the equality between types: a witness of equality between A and B is now the same as an element of the type $A \simeq B$ of *isomorphisms* between A and B. This can

The correct definition of an isomorphism between types is somewhat subtle, so we will simply take it as granted for the time being.

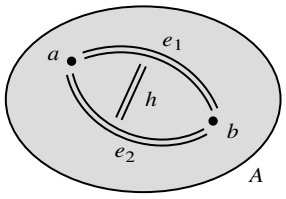
be seen as an extensionality principle for the universe of types, and adding it to intensional type theory has far reaching consequences. In particular, univalence implies the principle of function extensionality.

Homotopy Type Theory The univalence axiom departs from the standard mathematical usage of equality, in that equality can no longer be a proposition since it now contains essential computational information. For instance there are two distinct automorphisms of the type of booleans —one corresponding to the identity and the other to negation—and transporting along the identity is not the same as transporting along negation.

Transporting along an isomorphism amounts to an application of the isomorphism, up to a propositional equality.

$$\begin{aligned} \mathsf{J}(\mathsf{Type}, \mathsf{Bool}, \lambda X\,.\,X, \mathsf{true}, \mathsf{Bool}, \mathsf{id}) &=_{\mathsf{Bool}} \mathsf{true} \\ \mathsf{J}(\mathsf{Type}, \mathsf{Bool}, \lambda X\,.\,X, \mathsf{true}, \mathsf{Bool}, \mathsf{neg}) &=_{\mathsf{Bool}} \mathsf{false} \end{aligned}$$

Thus when we use the univalence axiom, we must accept that types now contain important information in their equality types. But these equality types also contain information in their own equality types, and so on. We end up with infinite towers of relevant equality types, all contained in the data of a type.



$$\begin{aligned} &A : \mathsf{Type}_i \\ &a : A \\ &b : A \\ &e_1 : a =_A b \\ &e_2 : a =_A b \\ &h : e_1 =_{(a =_A b)} e_2 \end{aligned}$$

Figure 2.1: Graphical representation of a type with inhabitants and equalities

All in all, these types do not really resemble sets anymore. Voevodsky noticed that they are much closer to *weak* ∞*-groupoids*, a higher categorical gadget that plays an important role in homotopy theory, the branch of math that studies spaces and their deformations. This is the starting point of *homotopy type theory* (HoTT), a far-reaching analogy between homotopy theory and intensional type theory with the univalence axiom [HoTT].

In HoTT, we think of types as spaces. Inhabitants of a type play the role of its points, equalities between two terms correspond to paths between the points, equalities between two equalities are continuous deformations between the corresponding paths, and so on. Naturally, functions between types are continuous in that they preserve equalities, and isomorphisms correspond to homotopy equivalences.

Homotopy type theory also extends the notion of inductive type to *higher inductive types* (HITs), which feature constructors for paths/equalities in addition to constructors for elements. HITs provide a variety of basic spaces, such as the circle, the sphere and the torus, but also a notion of quotient types.

This definition of the circle type might be surprising, as it looks nothing like the expected space $\{(x, y) \in \mathbb{R}^2 \mid x^2 + y^2 = 1\}$. This is because types in HoTT are not *topological* spaces, but rather synthetic spaces built out of points, paths and higher dimensional faces. As such, the circle is freely generated by a base point and a path from that base point to itself.

$$\begin{aligned} &\mathsf{Inductive}\ \ \mathsf{circle} : \mathsf{Type}_0\ \ := \\ &\mid\ \ \mathsf{base}\ \ :\ \ \mathsf{circle} \\ &\mid\ \ \mathsf{loop}\ \ :\ \ \mathsf{base} =_{\mathsf{circle}} \mathsf{base} \end{aligned}$$

$$\begin{aligned}&\mathsf{Inductive}\ \ \mathsf{quotient}\ (A : \mathsf{Type}_i)\ (R : A \to A \to \mathsf{Type}_i) : \mathsf{Type}_i\ \ :=\\ &\mid\ \ \mathsf{emb}\ \ :\ \ A \to \mathsf{quotient}\ A\ R\\ &\mid\ \ \mathsf{quo}\ \ :\ \ \Pi(a\ b : A).\, R\ a\ b \to \mathsf{emb}\ a =_{\dots} \mathsf{emb}\ b\end{aligned}$$

Using higher inductive types and the univalence axiom it becomes possible to prove a surprising amount of classical results from homotopy theory, rephrased to talk about types and equality: in his PhD thesis, Brunerie managed to show that the fourth homotopy group of the three-dimensional sphere has two elements [18]. Since then, the HoTT community has developed extensive libraries of formalized univalent mathematics in Coq [19] and Agda [20].

[18]: Brunerie (2016), "On the homotopy groups of spheres in homotopy type theory"

[19]: Bauer et al. (2017), "The HoTT Library: A Formalization of Homotopy Type Theory in Coq"

[20]: Agda-Unimath development team (2022), *The Agda-Unimath Library*

However, homotopy type theory is not exactly computational. The univalence axiom is, well, an axiom—which means that the J eliminator will not reduce when applied to an equality that we obtained from an isomorphism using univalence. Higher inductive types also block computation, because their elimination principles are simply postulated as axioms. This situation is all the more disappointing as the features introduced by HoTT seem to follow the principles of constructive mathematics.

Cubical Type Theory Subsequently, the type theory community directed its attention to the search for a computational interpretation of the univalent equality.

In 2014, Bezem Coquand and Huber built a model of the univalence axiom in a constructive set theory [21]. Their model interprets types as *fibrant cubical sets*, a combinatorial model of weak ∞-groupoids that lends itself well to a constructive presentation. Although this model is not developed in type theory, it allows evaluation of closed terms that make use of the univalent equality, and as such it paved the way for the understanding of the computational behavior of univalence.

[21]: Bezem et al. (2014), "A Model of Type Theory in Cubical Sets"

This program came to fruition a year later with the introduction of *cubical type theory*, an intensional type theory based on the insights provided by the fibrant cubical sets [22]. This new system incorporates primitives from homotopy theory directly in the typing judgments, and provides them with adequate computation rules. With this additional structure, it becomes possible to derive the principle of univalence as a theorem, which makes cubical type theory an axiom-free univalent theory. And since it satisfies a canonicity property [23] and supports a normalization function for open terms [24], cubical type theory can be deemed fully computational. On top of this, it supports higher inductive types with natural elimination principles and computation rules [25].

[22]: Cohen et al. (2015), "Cubical Type Theory: a constructive interpretation of the univalence axiom"

[23]: Huber (2019), "Canonicity for Cubical Type Theory"

[24]: Sterling et al. (2021), "Normalization for Cubical Type Theory"

[25]: Cavallo et al. (2019), "Higher Inductive Types in Cubical Computational Type Theory"

In the years following its introduction, cubical type theory flourished into a vibrant field of research. Many variations of the original system have been studied on paper [26, 27] and implemented in proof assistants [28, 29]. Among them, the `cubical` mode of the Agda proof assistant [38] is without a doubt the most successful implementation, and it has set the stage for several large-scale formalization projects.

[38]: Vezzosi et al. (2019), "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types"

2.3 Contributions

The Observational Calculus of Constructions In the first part of this thesis, we study the meta-theoretical properties of the observational equality. To this end, we introduce a formal calculus based on the observational type theory of Altenkirch *et al*, which we call the Observational Calculus of Constructions, or CC^{obs} for short. CC^{obs} is based on dependent type theory with inductive types and an impredicative universe of strict propositions. On top of this, it adds an implementation of the observational equality and a primitive type casting operator, from which we can derive the principles of uniqueness of identity proofs, of function extensionality and of proposition extensionality. CC^{obs} also supports the formation of the quotient of a type by a strict equivalence relation.

We equip our formal system with a reduction strategy that we use to prove a normalization theorem, canonicity of the basic datatypes, and decidability of the typing relation. These results are obtained through a normalization model, that we build in Agda using the logical relations framework developed by Abel *et al.* [30]. The normalization proof can be browsed at *[Everything.agda]*, but links to the code will be scattered throughout the relevant parts.

[30]: Abel et al. (2018), "Decidability of Conversion for Type Theory in Type Theory"

To the best of our knowledge, our model introduces several innovations compared to the existing literature on normalization proofs. The most basic, but also the most important one is that we do not normalize the proofs of strict propositions, contrary to the approach championed by Gilbert *et al.* [6]. This allows us to not only support proof-irrelevant axioms, but also to handle impredicativity without resorting to the usual reducibility candidates. Without the need for reducibility candidates, we can use a proof-irrelevant logical relation, which in turn lets us have non-parametric operators such as the observational equality and type-casting. In counterpart for this added flexibility, normalization does not directly imply canonicity anymore, and a separate proof of consistency of the theory is required to derive canonicity—as suggested by McBride *et al.* in their observational type theory paper [31]. We bridge that gap by building a set-theoretic model for CC^{obs}.

[6]: Gilbert et al. (2019), "Definitional Proof-Irrelevance without K"

[31]: Altenkirch et al. (2007), "Observational Equality, Now!"

This work solves several problems that were still open as far as we know: first and foremost, we get a proper proof of normalization and decidability for observational type theory, which was conjectured by Altenkirch *et al.* Moreover, the equality of CC^{obs} lives in an impredicative universe of strict propositions just like the equality of Lean. We show that we can interpret the J eliminator of Lean in an extension of CC^{obs}, while preserving normalization and decidability of the typechecking. This answers the question of Abel and Coquand about the compatibility of proof-irrelevant impredicativity and UIP [5].

[5]: Abel et al. (2020), "Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality"

Toward a Cubical Translation In the second part, we work toward translating HoTT into our observational calculus of constructions. Since CC^{obs} is a computational type theory, such a translation would provide a new way to compute with the univalence axiom.

The starting point of this part is the so-called "prefascist" translation of Pédrot that interprets every type of Martin-Löf type theory as a *presheaf*

in an intensional type theory with strict propositions [32]. Presheaves are a categorical construction that subsumes many important objects, and among them the cubical sets that Coquand *et al.* used to model univalence. However, the prefascist translation is missing an important ingredient of that model, called the *fibration structure*.

[32]: Pédrot (2020), "Russian Constructivism in a Prefascist Theory"

We add two new pages to this story: first, we present a dissection of the prefascist construction of Pédrot, explaining how it gets around the well-known difficulties of working with presheaves in intensional type theory. Then we explain how to extend the translation with a notion of fibration structure that should allow for an interpretation of the univalence axiom.

Synthetic Cubical Homotopy Theory Finally, in the third part of this thesis, we aim to show the practical gains of using a *computational* system that supports univalence and higher inductive types. To do so, we formalize a handful of classical results from homotopy theory in cubical type theory, and we compare the process with similar formalizations done in axiomatic HoTT. Our example results include:

- The equivalence of the torus and the product of two circles together with the computation of their respective fundamental groups.
- The equivalence between direct definitions of low dimensional spheres, and alternative definitions using iterated suspensions.
- Definition of pushout together with a direct proof of the "3×3 lemma".
- Definition of the join of two types and a proof that it is associative. Using this we get two proofs, one inspired by HoTT and a new direct cubical proof, that $\mathbb{S}^3$ is equivalent to the join of two circles.
- Definitions of the Hopf fibration and a proof that its total space is $\mathbb{S}^3$.

This work was done in the Agda proof assistant, and has been integrated to the standard library of cubical Agda [33].

[33]: Agda-Cubical development team (2022), *A Standard Library for Cubical Agda*

2.3.1 Publications

This thesis builds upon three peer-reviewed papers:

- The development of synthetic homotopy theory in cubical type theory was presented in a paper written with Anders Mörtberg published at CPP'20 [34].
- The construction of a logical relation for a predicative version of observational type theory is joint work with Nicolas Tabareau and was presented at POPL'22 [35].
- The treatment of impredicativity in observational type theory is also joint work with Nicolas Tabareau. It has been accepted at POPL'23 [36].

[34]: Mörtberg et al. (2020), "Cubical Synthetic Homotopy Theory"

[35]: Pujet et al. (2022), "Observational Equality: Now For Good"

[36]: Pujet et al. (2023), "Impredicative Observational Equality"

Additionally, the cubical translation of part 2 was the object of an extended abstract presented at ICMS'20.

2.4 Theories and Meta-Theories

In the following chapters, we will be juggling with a fair amount of formal systems—mostly dependent type theories, but also a few first-order theories. Since the mapping between type theories and names seems to be rather inconsistent in the literature, we fix our own convention:

MLTT — Intensional Martin-Löf Type Theory designates a type theory with dependent products, a predicative hierarchy of universes and *some amount* of inductive types [2]. We remain intentionally vague about which inductive types are included, and we will use more explicit notations such as $\mathsf{MLTT}^{\mathsf{ind}}$ when such concerns become relevant.

pCIC — The predicative Calculus of Inductive Constructions has all the features of MLTT, plus a universe Prop of impredicative propositions with large elimination of subsingleton inductive types [37].

$\mathsf{CC}^{\mathsf{obs}}$ / $\mathsf{CC}^{\mathsf{obs+}}$ — The Obervational Calculus of Constructions is our implementation of observational type theory, which is presented in great detail in chapter 3. The extended system $\mathsf{CC}^{\mathsf{obs+}}$ adds a rule for the computation of type-casting on reflexive identity proofs, as described in chapter 5.

As the bulk of this thesis deals with meta-theoretical properties of type theories, we will spend a lot of time treating them as mathematical objects, *i.e.* as a collection of syntax tokens and typing rules. But type theories are also mathematical frameworks in their own right, that we can use to prove theorems.

And indeed we will be doing a lot of mathematics inside type theories, often adopting an informal style to avoid overwhelming the reader with unsightly proof terms. When using type theory as our meta-theory, we will use the syntax and color scheme of the Agda proof assistant, for the simple reason that all of our formalization is done in Agda. When no Agda code appears, it is safe to assume that we are working without a specific framework in mind, as in most mathematical writing.

The Observational Equality

The Observational Calculus of Constructions

3

In this chapter we present the observational calculus of constructions (CC^{obs}), which is a flavor of Altenkirch, McBride and Swierstra's observational type theory [13]. CC^{obs} extends Martin-Löf Type Theory with support for impredicative propositions and an observational equality that satisfies UIP, function extensionality and proposition extensionality, while preserving the computational properties of MLTT.

We start in section 3.1 with some informal considerations about the system, then we delve into a detailed description of the typing rules in section 3.2, and finally we conclude with an overview of the meta-theoretical properties of CC^{obs} in section 3.3.

3.1 Overview of the Observational Calculus of Constructions

3.1.1 Constructions and Propositions

CC^{obs} separates types into two layers: a proof-relevant layer for *constructions* and a proof-irrelevant layer for *propositions*.

The proof-relevant types inhabit the universe hierarchy $\mathcal{U}_i$. This layer works exactly like usual Martin-Löf Type Theory, and contains familiar types such as the type of natural numbers, Π-types, Σ-types or universes; but also new types such as subset types and quotient types that we will introduce in due course. Generally speaking, proof-relevant objects correspond to mathematical constructions or programs, depending on which side of the Curry-Howard correspondance you are most familiar with.

In fact, MLTT can be embedded in the proof-relevant layer of CC^{obs} as we will see.

The second layer corresponds to the impredicative universe of propositions Ω, and it behaves rather differently. Indeed, propositions are *definitionally proof-irrelevant types*, which means that any two inhabitants of a type $A : \Omega$ are convertible simply by virtue of having type A:

$$\frac{\Gamma \vdash A : \Omega \qquad \Gamma \vdash t, u : A}{\Gamma \vdash t \equiv u : A}$$

We use the triple equal symbol $\equiv$ to denote convertibility, also referred to as definitional equality.

This corresponds to the universe of strict propositions sProp described in Gilbert et al. [6] and implemented in Coq and Lean. Propositions contain no computational information whatsoever, but they may be used to express logical constraints on proof-relevant objects. Examples of propositions will include True, False, logical connectors and the observational equality.

[6]: Gilbert et al. (2019), "Definitional Proof-Irrelevance without K"

These two layers are far from being hermetically sealed, though: the entire point of CC^{obs} is to leverage the flexibility of proof-irrelevance to extend the proof-relevant fragment with extensionality principles. For instance, the elimination principle for the observational equality allows

us to prove function extensionality even for the *inductive* equality of CC^{obs}, which lives in the proof-relevant layer!

3.1.2 The Observational Equality

A central feature of CC^{obs} is that every proof-relevant type A comes equipped with a proof-irrelevant *observational equality* type, that we write $t \sim_A u$.

Proof-irrelevant propositions, for their part, do not need to be equipped with an equality relation: since any two inhabitants are convertible, they would always be equal by reflexivity.

As suggested by its name, this equality is built around the idea that two objects should be identified when they behave similarly with respect to all obervations we can make on them. For instance, the observations we can make on a function $f : \mathbb{N} \to \mathbb{N}$ basically amount to applying f to some number n, and computing the result. Thus, two integer functions f and g are observationally equal when $f\,n \sim_{\mathbb{N}} g\,n$ for all $n : \mathbb{N}$, which is the principle of function extensionality.

In this regard, the observational equality is similar to the univalent equality found in Homotopy Type Theory [HoTT]. But it is also fundamentally different: by virtue of being a proof-irrelevant proposition, the observational equality validates the principle of Uniqueness of Identity Proofs (UIP), while the equality of HoTT equips types with the complex structure of a higher groupoid. In other words, CC^{obs} is a language for set-like objects and set-based mathematics, not higher mathematics. In exchange for the comfortable simplicity of UIP, the observational equality is not univalent, and in particular it does not identify isomorphic types.

[HoTT]: Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

From a computational perspective, the observational equality behaves very differently from the usual Martin-Löf Identity Type. The equality type $t \sim_A u$ should be understood as an *eliminator* that computes by reducing the type A to its normal form, and then pattern-matches on it to produce the equality relation which is appropriate for A. For instance, if A happens to be a function type, then the observational equality will reduce to the statement of function extensionality:

$$f \sim_{A \to B} g \quad \Longrightarrow \quad \Pi(x : A).\, f\,x \sim_B g\,x.$$

Likewise, all the other type formers have corresponding computation rules that prescribe the behavior of the observational equality on their inhabitants.

3.1.3 Eliminating Equality with Type Casting

For the observational equality to be of any interest, we need a way to use it in constructions and proofs: from an inhabitant of $P\,a$ and a proof of $a \sim b$, we should be able to derive an inhabitant of $P\,b$.

This principle is called the Leibniz rule. It should apply to both relevant and irrelevant predicates.

$$\frac{\Gamma \vdash a, b : A \qquad \Gamma \vdash e : a \sim_A b \qquad \Gamma \vdash P : A \to \mathcal{U} \qquad \Gamma \vdash t : P\,a \qquad \Gamma \vdash A : \mathcal{U}}{\Gamma \vdash ? : P\,b}$$

We use the light blue font to help distinguish CC^{obs} keywords from the body text. We do not use it for mathematical symbols however, since they are easy enough to identify as is.

To this end, CC^{obs} provides the two primitive eliminators transp and cast.

Transport Elimination of equalities into the proof-irrelevant layer is done with transp. Since the result is computationally irrelevant, this operator is not required to satisfy any kind of equation—in the proof-irrelevant world, all equations hold true as long as they are well-typed—and thus transp is merely an axiom that postulates the Leibniz rule for propositional predicates, with no computation rules whatsoever.

$$\frac{\Gamma \vdash A, b : A \qquad \Gamma \vdash e : a \sim_A b \qquad \Gamma \vdash P : A \to \Omega \qquad \Gamma \vdash t : P\, a \qquad \Gamma \vdash A : \mathcal{U}}{\Gamma \vdash \mathsf{transp}(A, a, P, t, b, e) : P\, b}$$

We can use it to derive important properties of the observational equality, such as the fact that it is an equivalence relation, or that it is preserved by functions:

$$\mathsf{ap} : \Pi(f : A \to B).\, x \sim_A y \to f\, x \sim_B f\, y.$$

The variables A, B, x, y are implicitly quantified over the appropriate types, which we omit to avoid clutter.

Type casting On the other hand, the cast operator eliminates equalities into the proof-relevant layer by allowing to cast terms between two observationally equal types.

$$\frac{\Gamma \vdash A, B : \mathcal{U} \qquad \Gamma \vdash e : A \sim_{\mathcal{U}} B \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{cast}(A, B, e, t) : B}$$

Given a proof $e : A \sim_{\mathcal{U}} B$ and an inhabitant $t : A$, the term $\mathsf{cast}(A, B, e, t)$ produces an inhabitant of B by computing the normal forms of the types A and B, and then picking an appropriate computation rule according to their head constructors. For instance, if A and B are two function types, then the cast will reduce to a function that casts back and forth:

$$\begin{aligned} &\mathsf{cast}(A \to B, A' \to B', e, f) \\ &\quad \Longrightarrow \quad \lambda(x : A').\, \mathsf{cast}(B, B', e_2, f\, \mathsf{cast}(A', A, {e_1}^{-1}, x)). \end{aligned}$$

In this computation, e is a proof of $A \to B \sim A' \to B'$, which is actually convertible to $(A \sim A') \wedge (B \sim B')$. We write e_1 and e_2 for its projections.

Just like with the observational equality, every type constructor comes with rules that describe the behavior of cast on its elements. Note that the equality proof e plays no computational role, and is just here to ensure that the cast is consistent. Thus cast is really an eliminator of the universe of proof-relevant types $\mathcal{U}$ rather than an eliminator of the observational equality.

The cast operator, despite its apparent simplicity, is in fact enough to derive the J eliminator of MLTT. Indeed, it is well-known that the J eliminator is equivalent to the Leibniz rule and the contractibility of singletons, but the contractibility of singletons is implied by proof irrelevance of the equality, and the Leibniz rule is a consequence of ap and cast. The corresponding term is spelled out in Subsection 3.2.11.

3.2 The Formal System $\mathrm{CC}^{\mathrm{obs}}$

3.2.1 Syntax

The syntax of the sorts, contexts, terms and types of the system is specified in figure 3.1. In order to distinguish them from the body text more easily, we will write keywords of $\mathrm{CC}^{\mathrm{obs}}$ in blue.

The formal system features dependent products $\Pi^{\mathfrak{s},\mathfrak{s}'}(x : A).B$, and dependent sums $\Sigma^{\mathfrak{s},\mathfrak{s}'}(x : A).B$. In an attempt to maximize readability, we will omit the sort annotations on Π and Σ when they can be inferred from the context, and we write them as respectively $A \to B$ and $A \times B$ when B does not depend on A. On top of this, the system supports some inductive types, which include the standard examples of natural numbers $\mathbb{N}$ and inductive equality Id but also the less standard quotient types and box types.

We will introduce some syntactic sugar for constructions of $\mathrm{CC}^{\mathrm{obs}}$, such as $\top$ for the true proposition or ap for the proof that functions preserve the observational equality (cf Subsection 3.1.3). Such notations will be introduced throughout the chapter, but for convenience they are collected in appendix A.1. The capture-avoiding substitution of a variable x in a term A by the term t is noted $B[x := t]$.

i, j	$\in$	$\mathbb{N}$	Universe levels
$\mathfrak{s}$	$::=$	$\mathscr{U}_i \mid \Omega$	Universes
Γ, Δ	$::=$	$\bullet \mid \Gamma, x : A : \mathfrak{s}$	Contexts
t, u, v, m, n, e, A, B	$::=$	$x \mid \mathfrak{s}$	Variables and Universes
	$\mid$	$\bot\text{-elim}(A, t) \mid \bot$	Empty type
	$\mid$	$t \sim_A u \mid \mathrm{refl}(t) \mid \mathrm{transp}(A, t, B, u, t', e)$	Observational equality
	$\mid$	$\mathrm{cast}(A, B, e, t) \mid \mathrm{castrefl}(A, t)$	Type cast
	$\mid$	$\mathbb{N} \mid 0 \mid \mathrm{S}\, t \mid \mathbb{N}\text{-elim}(A, t, u, n)$	Natural numbers
	$\mid$	$\Pi^{\mathfrak{s},\mathfrak{s}'}(x : A).B \mid \lambda(x : A).t \mid t\, u$	Dependent products
	$\mid$	$\Sigma^{\mathfrak{s},\mathfrak{s}'}(x : A).B \mid \langle t\, ;\, u\rangle \mid \mathrm{proj}_1(t) \mid \mathrm{proj}_2(t)$	Dependent sums
	$\mid$	$A/B \mid \pi(t) \mid \mathrm{Q}\text{-elim}(A, t, u, v)$	Quotient Types
	$\mid$	$\Box A \mid \diamond\, t \mid \Box\text{-elim}(t)$	Box Types
	$\mid$	$\mathrm{Id}(A, t, u) \mid \mathrm{Idrefl}(t) \mid \mathrm{Idpath}(t) \mid \mathrm{J}(A, t, B, u, v, e)$	Inductive equality

Figure 3.1: Syntax of $\mathrm{CC}^{\mathrm{obs}}$

3.2.2 Structure of the Rules

The typing rules of $\mathrm{CC}^{\mathrm{obs}}$ are based on four different judgments:

$\vdash \Gamma$	the context Γ is well-formed
$\Gamma \vdash t : A : \mathfrak{s}$	the term t has type A in context Γ
$\Gamma \vdash t \equiv u : A : \mathfrak{s}$	t and u are convertible at type A in context Γ
$\Gamma \vdash t \Rightarrow u : A : \mathscr{U}_i$	the term t reduces to u at type A in context Γ

In all the judgments, $\mathfrak{s}$ is either $\mathscr{U}_i$ or Ω. Note that the reduction only makes sense for proof-relevant terms, so it will never apply to a term in Ω.

The typing and convertibility rules define the algebraic theory of $\mathrm{CC}^{\mathrm{obs}}$, while the reduction rules describe a weak-head normalization algorithm that we will use to compute normal forms for the well-typed

terms (see Subsection 3.2.12). In case the type is a sort $\mathfrak{s}$, we shorten $\Gamma \vdash A : \mathfrak{s} : \mathfrak{s}^+$ to $\Gamma \vdash A : \mathfrak{s}$, and we similarly shorten conversion and reduction.

We use $\mathfrak{s}^+$ to represent the next bigger sort to $\mathfrak{s}$. It is defined by

$$\Omega^+ := \mathscr{U}_0$$
$$\mathscr{U}_i^+ := \mathscr{U}_{i+1}$$

As is often the case in type theory, the rules are organized in a very systematic structure: every type former comes with rules that describe its formation, introduction, elimination and computation. In $\mathrm{CC}^{\mathrm{obs}}$, every type former must also add rules that describe the observational equality on top of this.

3.2.3 Generic Rules

Contexts Rules Ctx-Nil, Ctx-Cons and Var describe the usual formation of contexts and typing of variables.

$$\textsc{Ctx-Nil}\ \frac{}{\vdash \bullet} \qquad \textsc{Ctx-Cons}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathfrak{s}}{\vdash \Gamma, x : A : \mathfrak{s}} \qquad \textsc{Var}\ \frac{\vdash \Gamma \qquad x : A : \mathfrak{s} \in \Gamma}{\Gamma \vdash x : A : \mathfrak{s}}$$

Conversion Conversion is an equivalence relation that is preserved by typing, and it subsumes the reduction of proof-relevant terms and the convertibility of proof-irrelevant terms.

$$\textsc{conv}\ \frac{\Gamma \vdash t : A : \mathfrak{s} \qquad \Gamma \vdash A \equiv B : \mathfrak{s}}{\Gamma \vdash t : B : \mathfrak{s}} \qquad \textsc{Refl}\ \frac{\Gamma \vdash t : A : \mathfrak{s}}{\Gamma \vdash t \equiv t : A : \mathfrak{s}}$$

$$\textsc{Sym}\ \frac{\Gamma \vdash t \equiv u : A : \mathfrak{s}}{\Gamma \vdash u \equiv t : A : \mathfrak{s}} \qquad \textsc{Trans}\ \frac{\Gamma \vdash t \equiv t' : A : \mathfrak{s} \qquad \Gamma \vdash t' \equiv u : A : \mathfrak{s}}{\Gamma \vdash t \equiv u : A : \mathfrak{s}}$$

$$\textsc{Red-Conv}\ \frac{\Gamma \vdash t \Rightarrow u : A : \mathscr{U}_i}{\Gamma \vdash t \equiv u : A : \mathscr{U}_i} \qquad \textsc{Proof-Irrelevance}\ \frac{\Gamma \vdash t, u : A : \Omega}{\Gamma \vdash t \equiv u : A : \Omega}$$

On top of this, we need to add bureaucratic congruence rules which make sure that each constructor preserves the judgmental equality (say for instance, if $t \equiv u$ then $\mathsf{S}\ t \equiv \mathsf{S}\ u$). Since these rules present little interest, we do not reproduce them here and refer the interested reader to appendix A.5.

3.2.4 The Logical Layer

Impredicative Π-Types Given a type A and a proof-irrelevant family $B : A \to \Omega$ on it, the system allows the formation of a proof-irrelevant Π-Type, with the usual introduction and elimination rules.

There is no need for β-reduction or an η rule. Since this type lives in the proof-irrelevant world, both conversion rules are a consequence of proof irrelevance.

$$\Pi\textsc{-Irr-Form}\ \frac{\Gamma, x : A : \mathfrak{s} \vdash B : \Omega}{\Gamma \vdash \Pi^{\mathfrak{s},\Omega}(x : A).\, B : \Omega} \qquad \textsc{Fun-Irr}\ \frac{\Gamma, x : A : \mathfrak{s} \vdash t : B : \Omega}{\Gamma \vdash \lambda(x : A).\, t : \Pi(x : A).\, B : \Omega}$$

$$\textsc{App-Irr}\ \frac{\Gamma \vdash t : \Pi(x : A).\, B : \Omega \qquad \Gamma \vdash u : A : \mathfrak{s}}{\Gamma \vdash t\ u : B[x := u] : \Omega}$$

If the domain A is proof-relevant, then the Π-type plays the role of a universal quantification. If A is a proposition instead, then the Π-type plays the role of a dependent implication between two propositions.

Note that this rule makes Ω into an impredicative sort. In particular, it allows the definition of self-referential propositions, which quantify over the type Ω of all propositions and may thus be applied to themselves. In addition to providing a tremendous amount of raw logical power, impredicativity is a remarkably flexible tool to define new propositions and abstractions. This versatility makes impredicativity a crucial ingredient of numerous tools in constructive mathematics, such as Tarski's fixed point theorem or lattice theory [7].

A sort is said to be impredicative if it is closed under dependent products over any index type.

[7]: Hur et al. (2013), "The Power of Parameterization in Coinductive Proof"

Dependent Conjunctions and Existential Types As a testament to the expressive power of impredicativity, we use it to derive new logical primitives, using the well-known technique of impredicative encodings.

Given a proposition A and a second proposition B that depends on A, we form the dependent conjunction of A and B as follows

$$(x : A) \,\&\, B \quad := \quad \Pi(X : \Omega).\, (\Pi(x : A).\, B \to X) \to X.$$

In presence of proof-irrelevance, this encoding provides everything we might ask of a dependent conjunction: on the one hand, from a proof a of A and a proof b of $B[x := a]$, we can derive a proof of $(x : A) \,\&\, B$ as a simple lambda abstraction

$$\langle a, b \rangle \quad := \quad \lambda\, X\, H\, .\, H\, a\, b$$

and conversely, given a proof t of $(x : A) \,\&\, B$ we define the following eliminators:

$$\begin{array}{rcll} \mathsf{fst}(t) & := & t\, A\, (\lambda\, a\, b\, .\, a) & : \quad A \\ \mathsf{snd}(t) & := & t\, (B\, \mathsf{fst}(t))\, (\lambda\, a\, b\, .\, b) & : \quad B\, \mathsf{fst}(t). \end{array}$$

When A is a proof-relevant type and B is a proof-irrelevant proposition that depends on A, we use the same construction to define the existential quantifier $\exists(x : A).\, B$.

$$\exists(x : A).\, B \quad := \quad \Pi(X : \Omega).\, (\Pi(x : A).\, B \to X) \to X.$$

In the case of an existential quantifier, the projections fst and snd are ill-typed, because we cannot use A as an argument of type Ω. This is only fair, since being able to recover an inhabitant of A from an a proof of propositional existence would require to extract computational content from a proof-irrelevant object.

Truth and Falsity To represent absurdity, $\mathrm{CC}^{\mathrm{obs}}$ features the false proposition $\bot$, from which we can define the true proposition $\top$.

Besides its formation rule (Rule $\bot$-Form), the false proposition comes with an elimination principle (Rule $\bot$-Elim) that applies to both proof-irrelevant and proof-relevant types.

This large elimination rule means that we could not replace $\bot$ with some impredicative encoding.

$$\frac{\vdash \Gamma}{\Gamma \vdash \bot : \Omega}\ \bot\text{-Form} \qquad \frac{\Gamma \vdash A : \mathfrak{s} \qquad \Gamma \vdash t : \bot : \Omega}{\Gamma \vdash \bot\text{-elim}(A, t) : A : \mathfrak{s}}\ \bot\text{-Elim}$$

In the theory implemented by Coq, this rule constitutes the unique connection between the proof-irrelevant and relevant types. This means that the only way to use information from a proof-irrelevant term to build a proof-relevant term is by using a proof of $\bot$, which amounts to proving that actually we are in an inaccessible branch or impossible case. In CC$^{\text{obs}}$ however, there is another way of using proof irrelevant information to build a proof relevant term, by transporting (or casting) along a proof of equality.

The true proposition on the other hand only requires a formation rule and a constructor. It does not require an eliminator, since the usual one can be derived from proof irrelevance. Thus, we can simply define them as

$$\begin{array}{lcll} \top & := & \bot \to \bot & : \quad \Omega \\ * & := & \lambda (x : \bot).\, x & : \quad \top. \end{array}$$

The Observational Equality As we explained in Subsection 3.1.2, a central feature of CC$^{\text{obs}}$ is that every proof-relevant type comes equipped with a proof-irrelevant equality type, noted $t \sim_A u$ and a canonical way to inhabit it, with the reflexivity constructor $\mathsf{refl}(t)$.

$$\frac{\Gamma \vdash t : A : \mathscr{U}_i \qquad \Gamma \vdash u : A : \mathscr{U}_i}{\Gamma \vdash t \sim_A u : \Omega}\ \text{Eq-Form} \qquad \frac{\Gamma \vdash t : A : \mathscr{U}_i}{\Gamma \vdash \mathsf{refl}(t) : t \sim_A t : \Omega}\ \text{Refl}$$

The equality type of CC$^{\text{obs}}$ should not be seen as a type former like the identity type of MLTT, but rather as an eliminator that produces a proposition by case analysis on the head constructor of the type A.

To transport a proof of a proposition along an observational equality, CC$^{\text{obs}}$ provides the eliminator transp.

$$\frac{\begin{array}{c}\Gamma \vdash t, t' : A : \mathscr{U}_i \quad \Gamma, x : A \vdash B : \Omega \\ \Gamma \vdash u : B[x := t] : \Omega \quad \Gamma \vdash e : t \sim_A t' : \Omega\end{array}}{\Gamma \vdash \mathsf{transp}(A, t, B, u, t', e) : B[x := t'] : \Omega}\ \text{Transport-}\Omega$$

Since it only ever constructs computationally irrelevant terms, this eliminator does not need any computational rule. Using transp, we can prove that the observational equality is in fact an equivalence relation, and we note e^{-1} for the inverse of e, $e \cdot e'$ for the transitivity, and $\mathsf{ap}\, f\, e$ for the preservation of equality by non-dependent function.

To deal with elimination of equality in a proof relevant context, CC$^{\text{obs}}$ provides a cast primitive that handles transport between two propositionally equal proof relevant types.

$$\frac{\Gamma \vdash e : A \sim_{\mathfrak{s}} B : \Omega \quad \Gamma \vdash t : A : \mathfrak{s}}{\Gamma \vdash \mathsf{cast}(A, B, e, t) : B : \mathfrak{s}}\ \text{Cast}$$

Note that our cast operation also applies to proof-irrelevant types. This is technically unnecessary as transp subsumes this case, but it will allow us to write more uniform reduction rules. The term $\mathsf{cast}(A, B, e, t)$ computes by case analysis on A and B, with specific computation rules for each type former (which are described in Subsection 3.2.8). The equality proof e plays no role in the computation whatsoever, and is only here to make sure the cast is consistent.

By default, cast applied to reflexivity only reduces to the identity function if the terms A and B are closed. To make up for this state of affairs, $\mathrm{CC}^{\mathrm{obs}}$ provides a proof-irrelevant axiom castrefl which asserts that casting along reflexivity is the identity.

$$\textsc{Cast-Refl}\quad \frac{\Gamma \vdash t : A : \mathscr{U}_i \qquad \Gamma \vdash e : A \sim_{\mathscr{U}_i} A : \Omega}{\Gamma \vdash \mathsf{castrefl}(A, t) : t \sim_A \mathsf{cast}(A, A, e, t) : \Omega}$$

In chapter 5, we study a slight variant of $\mathrm{CC}^{\mathrm{obs}}$ dubbed $\mathrm{CC}^{\mathrm{obs}+}$ where the rule Cast-Refl holds up to definitional equality.

3.2.5 Dependent products

From a type A and a proof-relevant dependent family $B : A \to \mathscr{U}_i$, we can form a proof-relevant dependent product, no matter what is the sort of the domain A. The resulting type has a universe level equal to the maximum of the level of the domain and the level of the codomain (where Ω is considered to have level 0).

The notation $\mathscr{U}_{\max(i,\mathfrak{s})}$ treats Ω as having universe level 0. Basically, it is equal to $\mathscr{U}_{\max(i,j)}$ if $\mathfrak{s}$ is $\mathscr{U}_j$, and to $\mathscr{U}_i$ if $\mathfrak{s}$ is Ω.

$$\Pi\textsc{-Rel-Form}\quad \frac{\Gamma, x : A : \mathfrak{s} \vdash B : \mathscr{U}_i}{\Gamma \vdash \Pi^{\mathfrak{s},\mathscr{U}_i}(x : A).\, B : \mathscr{U}_{\max(i,\mathfrak{s})}}$$

The introduction and elimination rules for dependent products correspond to the usual lambda-abstraction and application, with β-reduction and η-equality.

$$\textsc{Fun-Rel}\quad \frac{\Gamma, x : A : \mathfrak{s} \vdash t : B : \mathscr{U}_i}{\Gamma \vdash \lambda(x : A).\, t : \Pi^{\mathfrak{s},\mathscr{U}_i}(x : A).\, B : \mathscr{U}_{\max(i,\mathfrak{s})}}$$

$$\textsc{App-Rel}\quad \frac{\Gamma \vdash t : \Pi^{\mathfrak{s},\mathscr{U}_i}(x : A).\, B : \mathscr{U}_{\max(i,\mathfrak{s})} \qquad \Gamma \vdash u : A : \mathfrak{s}}{\Gamma \vdash t\, u : B[x := u] : \mathscr{U}_i}$$

$$\beta\textsc{-red}\quad \frac{\Gamma, x : A : \mathfrak{s} \vdash t : B : \mathscr{U}_i \qquad \Gamma \vdash u : A : \mathfrak{s}}{\Gamma \vdash (\lambda(x : A).\, t)\, u \Rightarrow t[x := u] : B[x := u] : \mathscr{U}_i}$$

Because reduction implies conversion (rule Red-Conv), rule β-red introduces both a reduction rule and a definitional equality.

$$\eta\textsc{-Eq}\quad \frac{\Gamma \vdash t, u : \Pi^{\mathfrak{s},\mathscr{U}_i}(x : A).\, B : \mathscr{U}_{\max(i,\mathfrak{s})} \qquad \Gamma, x : A : \mathfrak{s} \vdash t\, x \equiv u\, x : B : \mathscr{U}_i}{\Gamma \vdash t \equiv u : \Pi^{\mathfrak{s},\mathscr{U}_i}(x : A).\, B : \mathscr{U}_{\max(i,\mathfrak{s})}}$$

Furthermore, since dependent products are proof-relevant types, we must add rules that specify how the observational equality computes on values of type $\Pi(x : A).\, B$. In this case, it reduces to the point-wise

equality of the two functions, which bakes the principle of function extensionality in the system.

$$\textsc{Eq-Fun}\quad \frac{\Gamma \vdash f, g : \Pi^{\mathbb{s},\mathcal{U}_i}(x : A).\, B : \mathcal{U}_{\max(i,\mathbb{s})}}{\Gamma \vdash f \sim_{\Pi AB} g \Rightarrow \Pi^{\mathbb{s},\Omega}(x : A).\, f\, x \sim_B g\, x : \Omega}$$

3.2.6 Dependent Sums

$\mathrm{CC}^{\mathrm{obs}}$ supports proof-relevant dependent sums, with the usual rules from Martin-Löf Type Theory. We chose to implement η-extensional negative dependent sums, but a positive version would be just as straightforward.

$$\textsc{Σ-Form}\quad \frac{\Gamma, x : A : \mathcal{U}_i \vdash B : \mathcal{U}_j}{\Gamma \vdash \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}$$

$$\textsc{Pair}\quad \frac{\Gamma, x : A : \mathcal{U}_i \vdash B : \mathcal{U}_j \qquad \Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash u : B[x := t] : \mathcal{U}_j}{\Gamma \vdash \langle t\,;\, u\rangle : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}$$

$$\textsc{Fst}\quad \frac{\Gamma \vdash t : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}{\Gamma \vdash \mathsf{proj}_1(t) : A : \mathcal{U}_i} \qquad \textsc{Snd}\quad \frac{\Gamma \vdash t : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}{\Gamma \vdash \mathsf{proj}_2(t) : B[x := \mathsf{proj}_1(t)] : \mathcal{U}_j}$$

$$\textsc{Fst-Pair}\quad \frac{\Gamma, x : A : \mathcal{U}_i \vdash B : \mathcal{U}_j \qquad \Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash u : B[x := t] : \mathcal{U}_j}{\Gamma \vdash \mathsf{proj}_1(\langle t\,;\, u\rangle) \Rightarrow t : A : \mathcal{U}_i}$$

$$\textsc{Snd-Pair}\quad \frac{\Gamma, x : A : \mathcal{U}_i \vdash B : \mathcal{U}_j \qquad \Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash u : B[x := t] : \mathcal{U}_j}{\Gamma \vdash \mathsf{proj}_2(\langle t\,;\, u\rangle) \Rightarrow u : B[x := t] : \mathcal{U}_j}$$

$$\eta\textsc{-Depsum}\quad \frac{\Gamma \vdash t : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}{\Gamma \vdash t \equiv \langle \mathsf{proj}_1(t)\,;\, \mathsf{proj}_2(t)\rangle : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}$$

The observational equality between two inhabitants of a dependent sum reduces to the equality of the first and second projections—modulo a type casting, which is necessary for the second equality to type-check.

$$\textsc{Eq-Pair}\quad \frac{\begin{array}{c}\Gamma \vdash t, u : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)} \\ b' := \mathsf{cast}(B[x := \mathsf{proj}_1(t)], B[x := \mathsf{proj}_1(u)], \mathsf{ap}\ B\ e, \mathsf{proj}_2(t))\end{array}}{\Gamma \vdash \begin{array}{l} t \sim_{\Sigma AB} u \Rightarrow \\ (e : \mathsf{proj}_1(t) \sim_A \mathsf{proj}_1(u)) \,\&\, (b' \sim_B \mathsf{proj}_2(u))\end{array} : \Omega}$$

Note that in this rule, b' is only an abreviation, and not a premise — in particular, b' contains the variable e, which is not in scope yet. Although this might be a controversial choice, we will use this sort of abreviations fairly often for cast variables.

3.2.7 Natural numbers

The proof-relevant type of natural numbers $\mathbb{N}$ is the usual inductive datatype from MLTT. Note that its eliminator (rule $\mathbb{N}$-Elim) applies

indifferently to proof-relevant types and propositions, but the computation rules are only necessary in the proof-relevant case.

$$\textsc{$\mathbb{N}$-Form}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} : \mathscr{U}_0} \qquad \textsc{Zero}\ \frac{\vdash \Gamma}{\Gamma \vdash 0 : \mathbb{N} : \mathscr{U}_0} \qquad \textsc{Suc}\ \frac{\Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathsf{S}\, n : \mathbb{N} : \mathscr{U}_0}$$

$$\textsc{$\mathbb{N}$-Elim}\ \frac{\Gamma \vdash A : \mathbb{N} \to \mathfrak{s} \qquad \Gamma \vdash t_0 : A\ 0 : \mathfrak{s} \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).A\ n \to A\ (\mathsf{S}\, n) : \mathfrak{s} \qquad \Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathbb{N}\text{-elim}(A, t_0, t_S, n) : A\ n : \mathfrak{s}}$$

$$\textsc{$\mathbb{N}$-Elim-Zero}\ \frac{\Gamma \vdash A : \mathbb{N} \to \mathscr{U}_i \qquad \Gamma \vdash t_0 : A\ 0 : \mathscr{U}_i \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).A\ n \to A\ (\mathsf{S}\, n) : \mathscr{U}_i}{\Gamma \vdash \mathbb{N}\text{-elim}(A, t_0, t_S, 0) \Rightarrow t_0 : A\ 0 : \mathscr{U}_i}$$

$$\textsc{$\mathbb{N}$-Elim-Suc}\ \frac{\Gamma \vdash A : \mathbb{N} \to \mathscr{U}_i \qquad \Gamma \vdash t_0 : A\ 0 : \mathscr{U}_i \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).A\ n \to A\ (\mathsf{S}\, n) : \mathscr{U}_i \qquad \Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathbb{N}\text{-elim}(A, t_0, t_S, \mathsf{S}\, n) \Rightarrow t_S\ n\ \mathbb{N}\text{-elim}(A, t_0, t_S, n) : A\ (\mathsf{S}\, n) : \mathscr{U}_i}$$

Remark that rule $\mathbb{N}$-Elim applies to both proof relevant and proof irrelevant predicates, but the computation rules $\mathbb{N}$-Elim-Zero and $\mathbb{N}$-Elim-Suc only apply to proof relevant predicates. Reduction rules are pointless in the computationally irrelevant layer.

The observational equality between two natural numbers computes by recursively checking that their head constructors are compatible. If the comparison reaches $0 \sim 0$, the equality type reduces to $\top$, but in case it encounters two incompatible constructors, it returns $\bot$.

$$\textsc{Eq-zero}\ \frac{\vdash \Gamma}{\Gamma \vdash 0 \sim_{\mathbb{N}} 0 \Rightarrow \top : \Omega} \qquad \textsc{Eq-Suc}\ \frac{\Gamma \vdash n : \mathbb{N} : \mathscr{U}_0 \qquad \Gamma \vdash m : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathsf{S}\, m \sim_{\mathbb{N}} \mathsf{S}\, n \Rightarrow m \sim_{\mathbb{N}} n : \Omega}$$

$$\textsc{Eq-zero-suc}\ \frac{\Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash 0 \sim_{\mathbb{N}} \mathsf{S}\, n \Rightarrow \bot : \Omega} \qquad \textsc{Eq-suc-zero}\ \frac{\Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathsf{S}\, n \sim_{\mathbb{N}} 0 \Rightarrow \bot : \Omega}$$

3.2.8 Universes

The Universe of Propositions The universe Ω is a proof-relevant type of sort $\mathscr{U}_0$. Note that it is natural for this universe to be a proof-relevant type, since its inhabitants are the propositions, which are not convertible to one another despite being proof-irrelevant.

$$\textsc{Univ-Prop}\ \frac{\vdash \Gamma}{\Gamma \vdash \Omega : \mathscr{U}_0}$$

The observational equality between two inhabitants of Ω reduces to their logical equivalence, which bakes the principle of propositional extensionality into $\mathrm{CC}^{\mathrm{obs}}$.

Recall that the implication is a non-dependent proof-irrelevant Π-type. Likewise, we define the conjunction as the non-dependent case of the dependent conjunction.

$$\textsc{Eq-Ω}\ \frac{\Gamma \vdash A : \Omega \qquad \Gamma \vdash B : \Omega}{\Gamma \vdash A \sim_{\Omega} B \Rightarrow (A \to B) \wedge (B \to A) : \Omega}$$

The Universe Hierarchy of Proof-Relevant Types $\mathrm{CC}^{\mathrm{obs}}$ has a predicative hierarchy of proof-relevant universes.

$$\textsc{Univ-Rel}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}$$

The observational equality on inhabitants of $\mathcal{U}$ computes by recursively checking that the head constructors of the two types match. For instance, two proof-relevant function types $A \to B$ and $A' \to B'$ are observationally equal exactly when $A \sim A'$ and $B \sim B'$. In case the head constructors do not match, the observational equality is false, as enforced by the following rule.

$$\textsc{Eq-Univ-}\neq\ \frac{\vdash \Gamma \qquad \mathrm{hd}\, A \neq \mathrm{hd}\, B}{\Gamma \vdash A \sim_{\mathcal{U}} B \Rightarrow \bot : \Omega}$$

Rule Eq-Univ-$\neq$ is not strictly necessary. We can drop it and get a system that works just as well, with slightly less control over the universe.

The auxiliary function $\mathrm{hd}\, A$ returns the head constructor of A, including sort annotations. In particular, this means that the observational equality between a Π-type with a relevant domain and a Π-type with an irrelevant domain reduces to $\bot$.

In the case the head constructors *do* match, however, the behavior of the observational equality needs to be specified with a rule for each constructor. Here are the rules that handle the proof-relevant types that we described so far:

$$\textsc{Eq-Univ}\ \frac{\vdash \Gamma \qquad A \in \{\mathbb{N}, \Omega, \mathcal{U}_i\}}{\Gamma \vdash A \sim_{\mathcal{U}} A \Rightarrow \top : \Omega}$$

$$\textsc{Eq-}\Pi\ \frac{\Gamma \vdash A, A' : \mathfrak{s} \qquad \Gamma, x : A \vdash B : \mathcal{U}_i \qquad \Gamma, x : A' \vdash B' : \mathcal{U}_i \qquad a := \mathsf{cast}(A', A, e, a')}{\Gamma \vdash \begin{array}{l}\Pi(x : A).\, B \sim_{\mathcal{U}} \Pi(x : A').\, B' \Rightarrow \\ (e : A' \sim_{\mathfrak{s}} A)\ \&\ \Pi(a' : A').\, B[x := a] \sim_{\mathcal{U}} B'[x := a']\end{array} : \Omega}$$

$$\textsc{Eq-}\Sigma\ \frac{\Gamma \vdash A, A' : \mathfrak{s} \qquad \Gamma, x : A \vdash B : \mathcal{U}_i \qquad \Gamma, x : A' \vdash B' : \mathcal{U}_i \qquad a' := \mathsf{cast}(A, A', e, a)}{\Gamma \vdash \begin{array}{l}\Sigma(x : A).\, B \sim_{\mathcal{U}} \Sigma(x : A').\, B' \Rightarrow \\ (e : A \sim_{\mathfrak{s}} A')\ \&\ \Pi(a : A).\, B[x := a] \sim_{\mathcal{U}} B'[x := a']\end{array} : \Omega}$$

Remark that the resulting types for Π and Σ are slightly different (but equivalent) propositions. In each case, we picked the version that will result in a simpler reduction rule for cast in the next section.

The first rule Eq-Univ says that the diagonal equalities on the base type constructors are true. The other two (Eq-Π and Eq-Σ) say that two dependent function (resp. sum) types are equal when their domain are equal, and their codomain are pointwise equal (as type families) up to the equality on their domain. Naturally, there are also rules for quotients, box types and the inductive equality, that we will describe after having introduced them.

Reduction Rules for Cast As we explained in Subsection 3.1.3, the cast operator is an eliminator of the universe, which computes with a specific rule for each type. First, casting from $\mathbb{N}$ to $\mathbb{N}$ is defined by recursion.

We could instead define type-casting of natural numbers to be the identity, but we prefer this definition because it can be generalized to more complex inductive types, such as lists.

$$\textsc{Cast-Zero}\quad \frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathscr{U}_0} \mathbb{N}}{\Gamma \vdash \mathsf{cast}(\mathbb{N}, \mathbb{N}, e, 0) \Rightarrow 0 : \mathbb{N} : \mathscr{U}_0}$$

$$\textsc{Cast-Suc}\quad \frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathscr{U}_0} \mathbb{N} \qquad \Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathsf{cast}(\mathbb{N}, \mathbb{N}, e, \mathsf{S}\ n) \Rightarrow \mathsf{S}\ \mathsf{cast}(\mathbb{N}, \mathbb{N}, e, n) : \mathbb{N} : \mathscr{U}_0}$$

Casting a type from a universe to the same universe is the identity.

$$\textsc{Cast-Univ}\quad \frac{\Gamma \vdash e : \mathfrak{s} \sim_{\mathfrak{s}^+} \mathfrak{s} \qquad \Gamma \vdash A : \mathfrak{s}}{\Gamma \vdash \mathsf{cast}(\mathfrak{s}, \mathfrak{s}, e, A) \Rightarrow A : \mathfrak{s}}$$

Casting between two dependent products is slightly more involved: it produces a new function by casting back and forth the argument and return value of the original function.

$$\textsc{Cast-}\Pi\quad \frac{\Gamma \vdash e : \Pi(x : A).\, B \sim_{\mathscr{U}} \Pi(x : A').\, B' \qquad \Gamma \vdash f : \Pi(x : A).\, B : \mathscr{U} \qquad a := \mathsf{cast}(A', A, \mathsf{fst}(e), a')}{\Gamma \vdash \begin{array}{l}\mathsf{cast}(\Pi(x : A).\, B, \Pi(x : A').\, B', e, f) \Rightarrow \\ \lambda(a' : A').\, \mathsf{cast}(B[x := a], B'[x := a'], \mathsf{snd}(e)\ a', f\ a)\end{array} : \Pi(x : A').\, B' : \mathscr{U}_i}$$

Finally, casting between two dependent sums produces a new dependent pair by applying cast to the projections of the original one.

$$\textsc{Cast-}\Sigma\quad \frac{\Gamma \vdash e : \Sigma(x : A).\, B \sim_{\mathscr{U}} \Sigma(x : A').\, B' \qquad \Gamma \vdash t : \Sigma(x : A).\, B : \mathscr{U} \qquad a' := \mathsf{cast}(A, A', \mathsf{fst}(e), \mathsf{proj}_1(t))}{\Gamma \vdash \begin{array}{l}\mathsf{cast}(\Sigma(x : A).\, B, \Sigma(x : A').\, B', e, t) \Rightarrow \\ \langle a' \,;\, \mathsf{cast}(B[x := \mathsf{proj}_1(t)], B'[x := a'], \mathsf{snd}(e), \mathsf{proj}_2(t))\rangle\end{array} : \Sigma(x : A').\, B' : \mathscr{U}_i}$$

We only define reduction rules when the two types have the same head, and in other cases the cast will simply be a stuck term. Alternatively, we could reduce a cast between any two incompatible types to $\bot$–elim applied to the equality proof, which has type $\bot$ according to rule Eq-Univ-$\neq$. This amounts to replacing a stuck term with another, so we do not bother.

3.2.9 Quotients

Quotients are a ubiquitous construction in mathematics, and one that is famously difficult to handle smoothly in MLTT. The usual way to handle quotients is *via* setoids, but since this structure is not built in MLTT, all the functions between setoids, all the predicates, etc... have to be supplemented with bureaucratic equality preservation lemmas.

Recall that a setoid is a type A equipped with an equivalence relation, that plays the role of the equality of A.

In $\mathrm{CC}^{\mathrm{obs}}$ however, every type is naturally a setoid (with the observational equality) and every term preserves the setoid equality by construction. This is a very comfortable setting for quotients, that can thus be added to the theory—provided the relation that induces the quotient is proof-irrelevant. This can be seen as a limitation, as noticed by Sterling et al. [39], as it is generally impossible to extract proof-relevant information from equality in the quotient type. This is in contrast with the development of higher inductive types in the cubical setting [40]. On the other hand, the positive consequence of this limitation is that the elimination principle of the quotient types is fairly easy to manipulate in $\mathrm{CC}^{\mathrm{obs}}$.

[39]: Sterling et al. (2019), "Cubical Syntax for Reflection-Free Extensional Equality"

[40]: Coquand et al. (2018), "On Higher Inductive Types in Cubical Type Theory"

Quotient types are defined on a type A equipped with an equivalence relation on A.

Quotient-Form

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash R : A \to A \to \Omega : \mathcal{U}_i \\ \Gamma \vdash R_r : \Pi(x : A).\, R\, x\, x : \Omega \qquad \Gamma \vdash R_s : \Pi(x, y : A).\, R\, x\, y \to R\, y\, x : \Omega \\ \Gamma \vdash R_t : \Pi(x, y, z : A).\, R\, x\, y \to R\, y\, z \to R\, x\, z : \Omega\end{array}}{\Gamma \vdash A/(R, R_r, R_s, R_t) : \mathcal{U}_i}$$

Since the proofs of reflexivity, symmetry and transitivity appear everywhere but are proof-irrelevant, we will abreviate them as $\mathsf{isrel}(R)$ in the assumptions of the rules, and we write A/R instead of $A/(R, R_r, R_s, R_t)$.

The only constructor of quotient types is the canonical projection: from an element t of A, one obtains an element $\pi(t)$ of A/R. Equality between two canonical projections reduces to R.

Quotient-Proj

$$\frac{\Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash R : A \to A \to \Omega : \mathcal{U}_i \qquad \mathsf{isrel}(R)}{\Gamma \vdash \pi(t) : A/R : \mathcal{U}_i}$$

Quotient-Proj-Eq

$$\frac{\begin{array}{c}\Gamma \vdash t : A : \mathcal{U}_i \\ \Gamma \vdash u : A : \mathcal{U}_i \qquad \Gamma \vdash R : A \to A \to \Omega : \mathcal{U}_i \qquad \mathsf{isrel}(R)\end{array}}{\Gamma \vdash \pi(t) \sim_{A/R} \pi(u) \Rightarrow R\, t\, u : \Omega}$$

Observational equality between two quotient types reduces to equality of the (proof-relevant part of the) telescopes that define each quotient:

Quotient-Eq

$$\frac{\begin{array}{c}\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash R : A \to A \to \Omega \\ \mathsf{isrel}(R) \qquad \Gamma \vdash A' : \mathcal{U}_i \qquad \Gamma \vdash R' : A' \to A' \to \Omega \\ \mathsf{isrel}(R') \qquad x' := \mathsf{cast}(A, A', e, x) \qquad y' := \mathsf{cast}(A, A', e, y)\end{array}}{\Gamma \vdash \begin{array}{l} A/R \sim_{\mathcal{U}} A'/R' \Rightarrow \\ (e : A \sim_{\mathcal{U}} A') \;\&\; \Pi(x\, y : A).\, R\, x\, y \sim_{\Omega} R\, x'\, y' \end{array} : \Omega}$$

This rule is perhaps too fine compared to what we might hope for: plenty of isomorphic quotients are not identified. But remember that the observational equality between types is by no means univalent, and only identifies types that have been constructed in the same way.

Then, given a proof of equality between quotients, the cast operator reduces on canonical projections.

Quotient-Proj-Cast

$$\frac{\Gamma \vdash e : A/R \sim_{\mathcal{U}_i} A'/R' : \Omega \qquad \Gamma \vdash t : A : \mathcal{U}_i}{\Gamma \vdash \mathsf{cast}(A/R, A'/R', e, \pi(t)) \Rightarrow \pi(\mathsf{cast}(A, A', \mathsf{fst}(e), t)) : A/R' : \mathcal{U}_i}$$

The eliminator for quotient types encodes the universal property of

quotients: to construct a function out of a quotient A/R, it suffices to give a function t_π out of A such that if $R\ x\ y$, then their images under t_π are equal.

Quotient-Elim

$$\frac{\Gamma \vdash B : A/R \to \mathfrak{s} \qquad \Gamma \vdash t_\pi : \Pi(x : A).\, B\ \pi(x) \qquad \Gamma \vdash t_\sim : \Pi(x, y : A).\, \Pi(e : R\ x\ y).\, (t_\pi\ x) \sim_{B\ \pi(x)} \mathsf{cast}(B\ \pi(y), B\ \pi(x), B\ e^{-1}, t_\pi\ y) \qquad \Gamma \vdash u : A/R}{\Gamma \vdash \mathsf{Q{-}elim}(B, t_\pi, t_\sim, u) : B\ u : \mathfrak{s}}$$

The eliminator for quotient types has the obvious computation rule

Quotient-Proj-Elim

$$\frac{\Gamma \vdash B : A/R \to \mathscr{U}_i \qquad \Gamma \vdash t_\pi : \Pi(x : A).\, B\ \pi(x) \qquad \Gamma \vdash t_\sim : \Pi(x, y : A).\, \Pi(e : R\ x\ y).\, (t_\pi\ x) \sim_{B\ \pi(x)} \mathsf{cast}(B\ \pi(y), B\ \pi(x), B\ e^{-1}, t_\pi\ y) \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Q{-}elim}(B, t_\pi, t_\sim, \pi(u)) \Rightarrow t_\pi\ u : B\ (\pi(u)) : \mathscr{U}_i}$$

On top of this, quotients also come with the appropriate substitution and congruence rules.

3.2.10 Squash and Box Types

Squash Types The *squash*, or propositional truncation type former embeds the proof-relevant types into the proof-irrelevant world. Given a proof-relevant type A, its truncated version $\|A\|$ is a proposition that forgets all the computational information of inhabitants of A and only retains their mere existence. The squash type constructor is defined *via* an impredicative encoding.

$$\|A\| \quad := \quad \Pi(X : \Omega).\, (A \to X) \to X$$

Given an inhabitant t of A, we can give a proof of $\|A\|$ with a simple lambda-term:

$$|t| \quad := \quad \lambda\, X\, H\, .\, H\, t \quad : \quad \|A\|$$

The propositional truncation of $\mathrm{CC}^{\mathrm{obs}}$ is reminiscent of the propositional truncation used in Homotopy Type Theory, but the eliminator is much less flexible. Indeed, in HoTT the propositional truncation may be eliminated into any type A, provided we can show that all the inhabitants of A are propositionally equal. But in $\mathrm{CC}^{\mathrm{obs}}$, propositions are *strict*, meaning that being a proposition is about the definitional equality and not the propositional equality. Thus the eliminator would ideally apply to any type whose inhabitants are all convertible—but since we cannot reason internally about the definitional equality, we restrict elimination to types of sort Ω.

Box Types Conversely, Box types embed the propositions into the proof-relevant world. They are useful for a number of constructions: for instance, from a type $A : \mathscr{U}_i$ and a proof-irrelevant predicate $P : A \to \Omega$, one can build a *subset type* $\Sigma(x : A).\, \Box(P\ x) : \mathscr{U}_i$ whose inhabitants come

with a proof of P, but retain the computational behavior of inhabitants of A.

Another typical use of $\Box$ is to define a singleton type on which it is possible to reason about equality. Indeed, although $\top$ has only one inhabitant up-to conversion, it is not possible to state this internally as equality between proofs of propositions is not defined. However, one can state contractibility of the type $\Box\top$ and prove it, as it lives in $\mathscr{U}$.

Box types can be defined with the following rules:

$$\textsc{Box-Form}\ \frac{\Gamma \vdash A : \Omega}{\Gamma \vdash \Box A : \mathscr{U}_0} \qquad \textsc{Box-Proof}\ \frac{\Gamma \vdash t : A : \Omega}{\Gamma \vdash \diamond t : \Box A : \mathscr{U}_0} \qquad \textsc{Box-Proof-Eq}\ \frac{\Gamma \vdash t, u : \Box A : \mathscr{U}_0}{\Gamma \vdash t \sim_{\Box A} u \Rightarrow \top : \Omega}$$

$$\textsc{Unbox}\ \frac{\Gamma \vdash A : \Omega \qquad \Gamma \vdash t : \Box A}{\Gamma \vdash \Box\text{-elim}(t) : A} \qquad \textsc{Box-Eq}\ \frac{\Gamma \vdash A, B : \Omega}{\Gamma \vdash \Box A \sim_{\mathscr{U}} \Box B \Rightarrow A \sim_{\Omega} B : \Omega}$$

$$\textsc{Box-Proof-Cast}\ \frac{\Gamma \vdash A, B : \Omega \qquad \Gamma \vdash t : A \qquad \Gamma \vdash e : \Box A \sim_{\mathscr{U}} \Box B}{\Gamma \vdash \mathsf{cast}(\Box A, \Box B, e, \diamond t) \Rightarrow \diamond \mathsf{cast}(A, B, e, t) : \Box B}$$

3.2.11 The Inductive Equality

The J Eliminator for the Observational Equality The reader may wonder if $\mathrm{CC}^{\mathrm{obs}}$ extends Martin-Löf type theory— that is, whether a judgment of MLTT is also a judgment in the proof-relevant fragment of $\mathrm{CC}^{\mathrm{obs}}$. Indeed, our rules handle the universe hierarchy, dependent products, dependent sums and the natural numbers in the exact same way. But there are some difficulties with the Martin-Löf identity type.

The first idea that might come to the reader's mind is to use the observational equality of $\mathrm{CC}^{\mathrm{obs}}$ to interpret the inductive equality of MLTT. Surely, proof irrelevance will provide us with more definitional equalities than what we need! For this plan to work, we need to derive the J eliminator for the observational equality. It is not too difficult to come up with a term that satisfies the same typing rule:

$$\frac{\begin{array}{ccc}\Gamma \vdash A : \mathscr{U}_i & \Gamma \vdash t : A & \Gamma \vdash B : \Pi(x : A).\, t \sim_A x \to \mathbb{s} \\ \Gamma \vdash b : B\ t\ \mathsf{refl}(t) & \Gamma \vdash t' : A & \Gamma \vdash e : t \sim_A t'\end{array}}{\Gamma \vdash \mathsf{cast}(B\ t\ \mathsf{refl}(t), B\ t'\ e, \mathsf{eq_J}(A, t, B, t', e), b) : B\ t'\ e}$$

where the auxiliary operator $\mathsf{eq_J}$ is defined by

$$\begin{array}{lcl}\mathsf{eq_J}(A, t, B, t', e) & : & B\ t\ \mathsf{refl}(t) \sim B\ t'\ e \\ \mathsf{eq_J}(A, t, B, t', e) & := & \mathsf{transp}(A, t, \lambda\, x\, .\, \Pi(e' : t \sim x).\, B\ t\ \mathsf{refl}(t) \sim B\ x\ e', \\ & & \qquad \lambda\, e'.\, \mathsf{refl}, t', e)\ e.\end{array}$$

Remark that definitional proof irrelevance plays a role in the construction of $\mathsf{eq_J}$: the fourth argument of the transp operator is only well-typed when refl is convertible to any other proof of equality between t and itself.

There is however a discrepancy with the computation rule of the J eliminator. In MLTT, applying J to a proof of reflexivity is definitionally equal to the fourth argument:

$$\mathsf{J}(A, t, P, u, t, \mathsf{refl}(t)) \equiv u$$

but in $\mathrm{CC}^{\mathrm{obs}}$ with our replacement for J, this definitional equality is weakened to a propositional equality in general—it might hold by definition when P is a closed term, but it certainly will not if P is a variable. This is not a huge deal, as a theory with weakened computation rules still proves the exact same theorems in presence of UIP and funext [41], but it is an obstacle to a direct translation from MLTT to $\mathrm{CC}^{\mathrm{obs}}$.

[41]: Boulier et al. (2019), "Weak Type Theory is Rather Strong"

The Inductive Equality in $\mathrm{CC}^{\mathrm{obs}}$ A very similar problem was encountered by Cohen et al. [22] in Cubical Type Theory when trying to interpret the Martin-Löf identity type as the cubical path type, and was solved by Swan [42]. Following his ideas, we extend $\mathrm{CC}^{\mathrm{obs}}$ with Id-types:

[22]: Cohen et al. (2015), "Cubical Type Theory: a constructive interpretation of the univalence axiom"

[42]: Swan (2016), "An algebraic weak factorisation system on 01-substitution sets: a constructive proof"

$$\textsc{Id-Form}\quad \frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Id}(A,t,u) : \mathscr{U}_i} \qquad \textsc{Idrefl}\quad \frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{Idrefl}(t) : \mathsf{Id}(A,t,t)}$$

$$\textsc{Id-Proof-Eq}\quad \frac{\Gamma \vdash e, e' : \mathsf{Id}(A,t,u)}{\Gamma \vdash e \sim_{\mathsf{Id}(A,t,u)} e' \Rightarrow \top : \Omega}$$

$$\mathrm{J}\quad \frac{\begin{array}{c}\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash B : \Pi(x : A).\, \mathsf{Id}(A,t,x) \to \mathbb{s} \\ \Gamma \vdash u : B\ t\ \mathsf{Idrefl}(t) \qquad \Gamma \vdash t' : A \qquad \Gamma \vdash e : \mathsf{Id}(A,t,t')\end{array}}{\Gamma \vdash \mathsf{J}(A,t,B,u,t',e) : B\ t'\ e}$$

$$\textsc{J-Idrefl}\quad \frac{\begin{array}{c}\Gamma \vdash A : \mathscr{U}_i \\ \Gamma \vdash t : A \qquad \Gamma \vdash B : \Pi(x : A).\, \mathsf{Id}(A,t,x) \to \mathscr{U}_j \qquad \Gamma \vdash u : B\ t\ \mathsf{Idrefl}(t)\end{array}}{\Gamma \vdash \mathsf{J}(A,t,B,u,t,\mathsf{Idrefl}(t)) \Rightarrow u : B\ t\ \mathsf{Idrefl}(t)}$$

These rules mimic the behavior of Martin-Löf identity types, quotiented so they contain only one inhabitant up to observational equality. Let's see how we could extend the operations of $\mathrm{CC}^{\mathrm{obs}}$ to this new type former.

Firstly, we define the observational equality between two identity types as the equality of the telescopes of arguments, like we did for the quotient type:

$$\textsc{Id-Eq}\quad \frac{\begin{array}{c}\Gamma \vdash A : \mathscr{U}_i \\ \Gamma \vdash t : A \qquad \Gamma \vdash u : A \qquad \Gamma \vdash A' : \mathscr{U}_i \qquad \Gamma \vdash t' : A' \qquad \Gamma \vdash u' : A'\end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{Id}(A,t,u) \sim_{\mathscr{U}} \mathsf{Id}(A',t',u') \Rightarrow \\ (e : A \sim_{\mathscr{U}} A')\ \&\ \mathsf{cast}(A,A',e,t) \sim_{A'} t' \wedge \mathsf{cast}(A,A',e,u) \sim_{A'} u'\end{array} : \Omega}$$

Next, we may hope to define computation rules for cast by simply reducing the equality proof to a weak head normal form, and then commuting cast with the head constructor like we did for the type of natural numbers. However, we quickly run into a problem:

$$\frac{\begin{array}{c}\Gamma \vdash A, A' : \mathscr{U}_i \\ \Gamma \vdash t : A \qquad \Gamma \vdash t, u : A' \qquad \Gamma \vdash e : \mathsf{Id}(A,t,t) \sim \mathsf{Id}(A',t',u')\end{array}}{\Gamma \vdash \mathsf{cast}(\mathsf{Id}(A,t,t), \mathsf{Id}(A',t',u'), e, \mathsf{Idrefl}(t)) \Rightarrow\ ? : \mathsf{Id}(A',t',u')}$$

We cannot reduce this term to Idrefl, because t' and u' are not con-

vertible in general—e only provides us with a propositional equality $t' \sim_{A'} u'$. So in order to fix this, we add a new operator $\mathsf{Idpath}(e)$ that turns any inhabitant of $t \sim_A u$ into an inhabitant of $\mathsf{Id}(A,t,u)$.

IDPATH
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A \qquad \Gamma \vdash e : t \sim_A u}{\Gamma \vdash \mathsf{Idpath}(e) : \mathsf{Id}(A,t,u)}$$

Now we can define the computation rule for cast on $\mathsf{Idrefl}(t)$, using our new operator:

CAST-IDREFL
$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A, A' : \mathcal{U}_i \qquad \Gamma \vdash t', u' : A' \qquad \Gamma \vdash e : \mathsf{Id}(A,t,t) \sim \mathsf{Id}(A',t',u')}{\Gamma \vdash \begin{array}{l}\mathsf{cast}(\mathsf{Id}(A,t,t), \mathsf{Id}(A',t',u'), e, \mathsf{Idrefl}(t)) \Rightarrow \\ \mathsf{Idpath}(\mathsf{fst}(\mathsf{snd}(e))^{-1} \cdot \mathsf{snd}(\mathsf{snd}(e)))\end{array} : \mathsf{Id}(A',t',u')}$$

And naturally, we need to account for this additional constructor in the reduction rules for the observational equality and type-casting:

J-IDPATH
$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \Pi(x:A).\, \mathsf{Id}(A,t,x) \to \mathcal{U}_j \qquad \Gamma \vdash b : B\ t\ \mathsf{Idrefl}(t) \qquad \Gamma \vdash t' : A \qquad \Gamma \vdash e : t \sim_A t'}{\Gamma \vdash \begin{array}{l}J(A,t,B,b,t',\mathsf{Idpath}(e)) \Rightarrow \\ \mathsf{cast}(B\ t\ \mathsf{Idrefl}(t), B\ t'\ \mathsf{Idpath}(e), \mathsf{eq}_J(A,t,B,t',e), b)\end{array} : B\ t'\ \mathsf{Idpath}(e)}$$

CAST-IDPATH
$$\frac{\Gamma \vdash A, A' : \mathcal{U}_i \qquad \Gamma \vdash t, u : A \qquad \Gamma \vdash e : t \sim_A u \qquad \Gamma \vdash t', u' : A' \qquad \Gamma \vdash e' : \mathsf{Id}(A,t,t) \sim \mathsf{Id}(A',t',u')}{\Gamma \vdash \begin{array}{l}\mathsf{cast}(\mathsf{Id}(A,t,u), \mathsf{Id}(A',t',u'), e', \mathsf{Idpath}(e)) \Rightarrow \\ \mathsf{Idpath}(\mathsf{fst}(\mathsf{snd}(e'))^{-1} \cdot \mathsf{ap}\ (\mathsf{cast}(A,A',\mathsf{fst}(e'),-))\ e \cdot \mathsf{snd}(\mathsf{snd}(e')))\end{array} : \mathsf{Id}(A',t',u')}$$

With this handful of additional rules, our Id types satisfy all the computational rules of the Martin-Löf identity types. Using these, we can complete our embedding of MLTT into the proof-relevant layer of $\mathrm{CC}^{\mathrm{obs}}$.

Relation with the Observational Equality The Id type is logically equivalent to the observational equality: in one direction, the constructor IdPath proves that $t \sim_A u$ implies $\mathsf{Id}(A,t,u)$, and in the other direction we can get a proof of observational equality from an inhabitant of $\mathsf{Id}(A,t,u)$ by using the J eliminator. In particular, this means that the principles of uniqueness of identity proofs, of function extensionality and of propositional extensionality are derivable for the inductive equality, in stark contrast with Martin-Löf type theory.

Thus, the only difference between the two equalities is a tradeoff between definitional and propositional equations:

- The observational equality satisfies UIP by definition, but the J eliminator applied to reflexivity only satisfies its computation rule up to propositional equality.
- Id has a J eliminator that computes on reflexivity, but it only satisfies UIP up to a propositional equality.

In chapter 5, we explain how to modify $\mathrm{CC}^{\mathrm{obs}}$ to get the best of both worlds: an observational equality that satisfies both UIP and the computation of J.

3.2.12 Weak-head Reduction

All the rules that we enunciated so far provide us with a complete description of the typing discipline and of the definitional equality of $\mathrm{CC}^{\mathrm{obs}}$. But our reduction rules are still incomplete.

Indeed, so far we have only defined reduction rules for redexes consisting of an eliminator applied to a constructor. To complete our reduction rules into a proper evaluation strategy, we need to define a class of *normal forms* and add the necessary rules to evaluate an arbitrary program down to a normal form. To this end, we introduce the notions of constructors, destructors and scrutinees.

In the case of rule Subsection 3.2.5, the application is considered as an eliminator, and the lambda-abstraction as a constructor.

Constructors are all those tokens that are introduced by introduction rules, including 0 and S but also type formers (except for the observational equality) and lambda-abstractions.

On the other hand, eliminators are introduced by elimination rules, and they include tokens like $\mathbb{N}-\mathsf{elim}$ but also applications, cast or the observational equality type. An argument of an eliminator is said to be a *scrutinee* when it may trigger a computation rule if it contains a term with a constructor in head position. Note that an eliminator might have several scrutinees, for instance in $\mathsf{cast}(A, B, e, t)$ both A and B are scrutinees.

Now we can describe our evaluation strategy. We use a typed, deterministic *weak head* reduction:

- If a term has a constructor or a variable in head position, then it is a weak head normal form, and does not reduce further.
- If a term has an eliminator in head position, we reduce all the scrutinees of that eliminator down to normal form. Then, if all the scrutinees have a constructor in head position, we simplify the redex and restart the reduction process on the result. Otherwise, we have a weak head normal form.

In particular, we never reduce under a constructor, hence the name "weak head". The substitution rules that implement reduction of scrutinees to weak-head normal form are described in appendix A.6.

The normal forms for our weak head reduction strategy are the *weak head normal forms* (whnfs), defined in figure 3.2. A whnf is either a term with a constructor in head position, or a *neutral term*, *i.e.* a term that cannot be reduced to a normal form because it is stuck on a variable, a proof of $\bot$ or a cast along a non-diagonal equality. In $\mathrm{CC}^{\mathrm{obs}}$, inhabitants of a proof-irrelevant type are never considered as whnf, as there is no notion of reduction of proof-irrelevant terms.

$$
\begin{array}{llll}
\text{whnf} & w & ::= & N \\
& & | & \mathscr{U}_i \mid \Omega \mid \mathbb{N} \mid \bot \mid \Pi(x : A).\, B \mid \Sigma(x : A).\, B \mid A/R \mid \Box A \mid \mathsf{Id}(A, t, u) \\
& & | & 0 \mid \mathsf{S}\, t \mid \lambda(x : A).\, t \mid \langle t\,;\, u\rangle \mid \pi(t) \mid \diamond\, t \mid \mathsf{Idrefl}(t) \mid \mathsf{Idpath}(e) \\
\text{neutral} & N & ::= & x \\
& & | & N\, t \mid \mathsf{proj}_1(N) \mid \mathsf{proj}_2(N) \mid \bot\text{–}\mathsf{elim}(A, e) \\
& & | & \mathbb{N}\text{–}\mathsf{elim}(P, t, u, N) \mid \mathsf{Q}\text{–}\mathsf{elim}(P, t, e, N) \mid \mathsf{J}(A, t, P, u, N, e) \\
& & | & t \sim_N u \\
& & | & N \sim_{\mathbb{N}} m \mid 0 \sim_{\mathbb{N}} N \mid \mathsf{S}\, m \sim_{\mathbb{N}} N \\
& & | & N \sim_{A/R} t \mid \pi(t) \sim_{A/R} N \\
& & | & N \sim_{\mathscr{U}_i} A \mid w \sim_{\mathscr{U}_i} N \\
& & | & \mathsf{cast}(N, A, e, t) \mid \mathsf{cast}(w, N, e, t) \\
& & | & \mathsf{cast}(\mathbb{N}, \mathbb{N}, e, N) \mid \mathsf{cast}(A/R, A'/R', e, N) \\
& & | & \mathsf{cast}(w, w', e, t) \qquad (\text{where } \mathrm{hd}\, w \neq \mathrm{hd}\, w')
\end{array}
$$

Figure 3.2: Weak-head normal and neutral forms

3.3 Overview of the Meta-Theory

3.3.1 Properties of $\mathrm{CC}^{\mathrm{obs}}$

In order for a type theory to be a reasonable candidate for implementation in a proof assistant, it is desirable to have some amount of control over its behavior and its semantics. There is a wide variety of properties we can ask of a type theory, but in this section we will focus our attention on four important properties: consistency, normalization, canonicity and decidability. All the theorems of this section will be proved in chapter 4.

Theorem 3.3.1 (Consistency) *In the empty context, there is no proof of* $\bot$.

Our system is consistent in the sense that it is impossible to use it to derive a contradiction. It goes without saying that this property is of utmost importance if one wants to build interesting mathematics. When proving a theory consistent, one should pay some attention to the meta-theory where reasoning takes place, as a consistency result is really a reduction of the consistency of the theory to the consistency of the meta-theory.

Theorem 3.3.2 (Normalization) *If the judgment* $\Gamma \vdash t : A : \mathscr{U}_i$ *is derivable, then the weak-head reduction procedure applied to* t *eventually reaches a weak-head normal form.*

By recursively applying the weak-head normalization to the subterms of t*, we eventually reach a deep normal form.*

The adjective *normal* deserves a bit of discussion. Ideally, the normal form of a term should be a distinguished representative of its equivalence class under convertibility—and thus any two convertible terms should have the same normal form, so that checking convertibility is reduced to comparing the normal forms. However our deep reduction strategy does not η-expand/reduce terms, meaning that normal forms are only unique up to η-equivalence. This is not too much of an issue, since η-equivalence of β-normal well-typed terms is easy to decide—we just put them in η-long form.

Theorem 3.3.3 (Canonicity of $\mathbb{N}$) *If a term t has type $\mathbb{N}$ in the empty context, then there exists an external integer n such that t is convertible with $S^n\ 0$.*

A term belonging to a type is called canonical when it can be explicitly built up using the constructors of that type. For instance, an inhabitant of the type $\mathbb{N}$ of natural numbers is canonical if it is an explicit numeral. If all terms of type $\mathbb{N}$ in an empty context normalize to a canonical form, then the theory is said to enjoy canonicity for natural numbers.

Together with normalization, canonicity gives a computational justification of constructiveness: provided a closed witness of the type $\Sigma(n : \mathbb{N}).\, P\ n$, we can take its first projection and normalize it to recover a concrete integer n that satisfies P. Thus, proving a theorem truly does amount to giving an explicit construction of a witness. Note however that this is only true in the proof-relevant layer of $\mathsf{CC}^{\mathsf{obs}}$, as there is no computational content to extract from a proof of $(n : \mathbb{N})\ \&\ P\ n$.

Theorem 3.3.4 (Decidability) *There is an algorithm that decides whether any two terms t and u that have type A in context Γ are convertible or not.*

There is an algorithm that, given a context Γ, a term t, a type A and a sort $\mathbb{s}$, decides whether the judgment $\Gamma \vdash t : A : \mathbb{s}$ is derivable.

Finally, in order to implement a type theory in a proof assistant, it is desirable to have an algorithm that, given a context Γ and terms t and A, decides whether t is an inhabitant of A in context Γ. This ensures users that when they find a proof for a statement, the proof assistant can automatically check that their proof is correct.

3.3.2 Comparing $\mathsf{CC}^{\mathsf{obs}}$ with Martin-Löf Type Theory

The proof-relevant layer of $\mathsf{CC}^{\mathsf{obs}}$ supports all the basic types of Martin-Löf type theory: a universe hierarchy, dependent products, dependent sums, the natural numbers, and the inductive equality; furthermore all the computation rules that these types satisfy in MLTT still hold true in $\mathsf{CC}^{\mathsf{obs}}$. Thus, we can embed this minimalistic version of MLTT into our system.

Theorem 3.3.5 (Embedding) *If the judgment $\Gamma \vdash t : A$ is derivable in* MLTT*, then we can find an integer i such that $\Gamma \vdash t : A : \mathcal{U}_i$ is derivable in* $\mathsf{CC}^{\mathsf{obs}}$.

Of course, $\mathsf{CC}^{\mathsf{obs}}$ is not conservative over MLTT, as it is possible to prove function extensionality and UIP for the inductive equality. Still, there is a precise sense in which $\mathsf{CC}^{\mathsf{obs}}$ does not offer more computational power than MLTT.

An extension $\mathcal{T}'$ of a theory $\mathcal{T}$ is said to be conservative when it proves no new theorems that can be stated in $\mathcal{T}$

Theorem 3.3.6 (Computational power) *Any function that can be expressed as a term of type $\mathbb{N} \to \mathbb{N}$ in $\mathsf{CC}^{\mathsf{obs}}$ can be expressed as a term of type $\mathbb{N} \to \mathbb{N}$ in* MLTT.

This may come as a surprise, because $\mathrm{CC}^{\mathrm{obs}}$ is impredicative, and thus should be much more powerful than a predicative theory such as MLTT. Well, in a way, it is! The impredicativity allows us to prove the existence of functions that grow much faster than what is definable in MLTT. But there is a catch: we can only prove their existence in the proof-irrelevant layer, and we have no way to extract them to actual functions of type $\mathbb{N} \to \mathbb{N}$. We will discuss this in more detail in chapter 5.

3.3.3 Semantics of $\mathrm{CC}^{\mathrm{obs}}$

Because of the observational equality and the impredicative universe of propositions, types in $\mathrm{CC}^{\mathrm{obs}}$ feel a lot more "set-like" than types in Martin-Löf type theory: functions are extensional, equality proofs are unique, we can build quotients and subtypes, *etc*. Thus, it does not come as a surprise that we can interpret proofs in $\mathrm{CC}^{\mathrm{obs}}$ as talking about classical sets.

Theorem 3.3.7 $\mathrm{CC}^{\mathrm{obs}}$ *has a model in classical set theory, in which the universe hierarchy contains codes for all ZF sets.*

But $\mathrm{CC}^{\mathrm{obs}}$ is also constructive, *i.e.* the principle of excluded middle does not hold. This incites us to investigate more general classes of models, that do not collapse the propositions to a two-valued set. Now, there happens to be an important class of categories that share most properties of the category of sets but not excluded middle, called *Grothendieck toposes*. And indeed, Gratzer showed that $\mathrm{CC}^{\mathrm{obs}}$ has models in all Grothendieck toposes [43].

[43]: Gratzer (2022), *An inductive-recursive universe generic for small families*

This makes $\mathrm{CC}^{\mathrm{obs}}$ a reasonably good language to reason internally to a Grothendieck topos. However, $\mathrm{CC}^{\mathrm{obs}}$ does not validate all the reasoning principles that are available in the logic of toposes. In particular, the most glaring omission is the principle of unique choice, which states that given any relation $R : A \to B \to \Omega$ we can build a choice function if we know that R is the graph of a functional relation.

The unique existence quantifier $\exists!(x : A).\, P\, x$ is defined as the following proposition:
$\exists(x : A).\, (P\, x \wedge \Pi\, y\, .\, P\, y \to x \sim y)$.

$$\Pi(x : A).\, \exists!(y : B).\, R\, x\, y \quad \to \quad \Sigma(f : A \to B).\, \Pi(x : A).\, R\, x\, (f\, x).$$

With this, we conclude our overview of the Observational Calculus of Constructions and we turn to the next chapter, in which we will develop the tools that are needed to prove the various theorems that we enunciated.

Meta-theory of $\mathrm{CC}^{\mathrm{obs}}$ 4

In this chapter, we develop the meta-theory of $\mathrm{CC}^{\mathrm{obs}}$ to prove the theorems that are presented in section 3.3.

Firstly, in section 4.1 we build a normalization model for $\mathrm{CC}^{\mathrm{obs}}$ in Agda, based on the model for MLTT presented by Abel *et al.* [30]. By interpreting types as logical predicates on terms, we prove that every well-typed term of $\mathrm{CC}^{\mathrm{obs}}$ has a normal form that can be computed by reduction.

Then in section 4.2, we study the consequences of the normalization model. We explain how to deduce canonicity of $\mathrm{CC}^{\mathrm{obs}}$ from a proof of consistency, we show that conversion is decidable, and we give the outline of an algorithm for type-checking. Except for the decidability of typing, all the proofs of this chapter have been formally verified in Agda.

In section 4.3, we explain how to replay the construction of the normalization model in Martin-Löf Type Theory with W-types, a theory that is weaker than the theory of Agda. From there, we deduce that $\mathrm{CC}^{\mathrm{obs}}$ cannot express more integer functions than MLTT. This section is also supported by an Agda development.

Finally, in section 4.4 we construct a model of $\mathrm{CC}^{\mathrm{obs}}$ in ZF set theory with Grothendieck universes, from which we obtain a proof of consistency (and therefore canonicity) for $\mathrm{CC}^{\mathrm{obs}}$. Furthermore, we make sure that our model provides reasonable set-theoretic semantics for our system, so that statements of $\mathrm{CC}^{\mathrm{obs}}$ may be interpreted as natural statements of ZF set theory.

4.1 The Normalization Model

4.1.1 Overview of the Proof

Since the construction of the normalization model is rather intricate, it might be easy to get lost in the technical details and to miss the big idea. Therefore, in hope that the reader will find it helpful, we start by giving a lengthy bird's eye view of the proof with references and comparisons to the existing literature before diving head first into the actual proof in Subsection 4.1.2.

Proving Normalization In dependent type theory, proving that every well-typed term is normalizing is a complex matter. If we try to prove it by naive induction on the typing derivations, we quickly get stuck on the case of the application rule, because we cannot obtain that $t\ u$ is normalizing from the hypotheses that t and u are normalizing. Still, reasoning by induction on the typing derivations is pretty much our only option, as type theories are fundamentally inductive objects, presented by the inference rules. Thus what we really need is a better induction

hypothesis, one that implies normalization of well-typed terms but is also preserved by all the inference rules of the type theory.

When we are dealing with a type theory that is as sophisticated as MLTT or $\mathrm{CC}^{\mathrm{obs}}$, the typical induction hypotheses get very complex, with cases specifically tailored to handle each basic type and a lot of interdependence. At this point, it becomes easier to present the proof as the construction of a *model* for the type theory: we interpret every type A as a pair of a predicate $[A]$ that expresses the induction hypothesis for terms of type A, and a binary relation $[A]_=$ that gives the induction hypothesis for convertible elements of A. Next, we show by induction that all typing derivations are valid in the model, meaning that if $t : A$ is derivable then $t \in [A]$ and that if $t \equiv u : A$ is derivable then $[A]_= t\, u$. If we designed our predicates so that they imply normalization, we have our proof. Such models are traditionally built using the *reducibility* technique pioneered by Tait [44], or its reinterpretation in categorical language that goes by the name *gluing* [45].

The two predicates $[A]$ and $[A]_=$ could equivalently be presented as a partial equivalence relations on terms, which is the more traditional way to phrase it.

In [30], Abel *et al.* present a normalization model for a subset of MLTT that includes one predicative universe, dependent products and natural numbers with large elimination. They build their model in intensional type theory with induction-recursion, a meta-theory which is remarkably close to the theory being studied. Furthermore, they entirely formalized their proofs in Agda, down to the proof of decidability of conversion. In this chapter, we will extend their model to show normalization and decidability of conversion for the full system $\mathrm{CC}^{\mathrm{obs}}$. We formalized our proofs in Agda for a significant subset of $\mathrm{CC}^{\mathrm{obs}}$ that includes two predicative universes, the impredicative universe of propositions, dependent products, natural numbers and the observational equality. Links to our code will be scattered thoughout the chapter, but the main file can be consulted at *[Everything.agda]*.

[44]: Tait (1967), "Intensional Interpretations of Functionals of Finite Type I"

[45]: Altenkirch et al. (1997), "Reduction-free normalisation for system F"

[30]: Abel et al. (2018), "Decidability of Conversion for Type Theory in Type Theory"

Informal summary of the model of Abel *et al.* Abel *et al.* start by defining the *reducibility predicates*. Reducibility predicates are three proof-relevant predicates on untyped terms that are associated to some types of the theory:

- the unary predicate $\Gamma \Vdash_\ell \_ : A$ defines the set of *reducible terms* of type A in context Γ,
- the binary relation $\Gamma \Vdash_\ell \_ \equiv \_ : A$, defines *reducible equality* between two reducible terms of type A in context Γ,
- and the unary predicate $\Gamma \Vdash_\ell A \equiv \_$ defines the set of types *reducibly equal* to A in context Γ.

Whenever a type A is equipped with these three reducibility predicates in a context Γ, we write $\Gamma \Vdash_\ell A$ and we call A a *reducible type*.

A typical reducibility predicate characterizes the behaviour that "well-behaved" terms of type A should exhibit: for instance, a term t is deemed to be a reducible inhabitant of type $B \to C$ if it reduces to a term t' in weak-head normal form, such that t' applied to any reducible term of type B produces a reducible term of type C (up to some technical details).

The reducibility predicates are also indexed by a level ℓ, which reflects the predicative nature of the universe hierarchy of MLTT: reducibility

is first defined at level 0 to characterize types and terms that inhabit the smallest universe $\mathcal{U}_0$. Then Abel *et al.* use reducibility at level 0 to define reducibility at level 1, that applies to types and terms that live in $\mathcal{U}_1$—in the case of the type $\mathcal{U}_0 : \mathcal{U}_1$, they define

$$\begin{aligned} \Gamma \Vdash_1 A : \mathcal{U}_0 &:= \Gamma \Vdash_0 A \\ \Gamma \Vdash_1 A \equiv B : \mathcal{U}_0 &:= \Gamma \Vdash_0 A \equiv B. \end{aligned}$$

And this definition is iterated to handle any universe level.

However, reducibility is not quite strong enough to build a model of MLTT, so Abel *et al.* go on to define *validity* which is the closure of reducibility under substitution. Only after this step can they prove the *fundamental lemma* which shows that the validity model supports all the inference rules of their fragment of MLTT, and thus that every well-typed term can be interpreted in the validity model. Since validity implies normalization, this concludes the proof.

If we want to construct a similar model for $\mathrm{CC}^{\mathrm{obs}}$, we will need to add support for impredicative proof-irrelevant propositions and the observational equality.

Adding Proof-Irrelevant Types Gilbert *et al.* [6] extended the model of Abel *et al.* to add a universe of definitionally proof-irrelevant types. Contrary to the universe Ω of $\mathrm{CC}^{\mathrm{obs}}$, their universe is predicative, and only contains Π-types and the false proposition.

[6]: Gilbert et al. (2019), "Definitional Proof-Irrelevance without K"

In their proof, Gilbert *et al.* extend the reduction rules to the inhabitants of proof-irrelevant types, and they go on to prove normalization for both relevant and irrelevant terms. From normalization, they also deduce a proof of consistency for MLTT extended with predicative proof-irrelevant types: any proof of the false proposition $\bot$ in an an empty context will reduce to a weak head normal form, and the only weak head normal forms that can inhabit $\bot$ are neutral terms. But since there are no variables in the empty context, neutral terms cannot exist, so $\bot$ has no inhabitant in the empty context.

However, this strategy is not applicable in our setting. Contrary to the theory studied by Gilbert *et al.*, $\mathrm{CC}^{\mathrm{obs}}$ relies crucially on proof-irrelevant axioms such as castrefl, which do not have any clear reduction rule or normal forms. Therefore we will drop the idea of having reduction rules for inhabitants of propositions, and we will only prove normalization for proof-relevant terms. We argue that this is more faithful to the philosophy of computational irrelevance, and results in a proof that is completely agnostic about the proof-irrelevant content of the theory—we could postulate any consistent proof-irrelevant axiom (even excluded middle!) and the normalization proof would carry through just as well.

Unfortunately, this means that consistency and canonicity do not follow from normalization anymore. The reason is simple: since we gave up any kind of control on the proof-irrelevant terms, we might have proofs of $\bot$ in the empty context for all we can tell. But this also weakens our control on neutral terms in the proof-*relevant* layer, as $\bot$-elim can build an inhabitant of any proof-relevant type from a proof of $\bot$. Therefore, if we want to show canonicity, we need to show that there is no proof

of $\bot$ in the empty context—in other words, that the theory is consistent. This idea of deriving canonicity from consistency already appeared in the work of Altenkirch *et al.* [13].

[13]: Altenkirch et al. (2007), "Observational equality, now!"

We will get around this issue by constructing two models: the normalization model that provides us with normalization proofs, and a set-theoretic model from which we will derive a consistency proof for $\mathrm{CC}^{\mathrm{obs}}$. This second model is presented in section 4.4.

Adding Support for Impredicativity Impredicativity is famously difficult to model in reducibility proofs, because it breaks the well-foundedness of the type hierarchy: impredicative dependent products may have an arbitrarily large domain, while still being at the bottom of the universe hierarchy. The standard way to build a normalization model for impredicative theories that avoids self-referential issues is to use *reducibility candidates*, the device introduced by Girard [46] to prove normalization for System F. Fortunately, in $\mathrm{CC}^{\mathrm{obs}}$ the impredicative dependent products are propositions, and thus proof-irrelevant. As we explained above, we do not prove normalization for computationally irrelevant terms, so we will not need Girard's reducibility candidates.

[46]: Girard (1972), "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur"

However, we still need to define reducibility for Ω, which is a proof-relevant type. And since its sort is $\mathscr{U}_0$, the reducibility predicates for Ω should be defined at level 0, without any knowledge of the model for higher universes. So, when should an impredicative dependent product be reducible? In the model of Abel *et al.*, dependent products are deemed reducible when their domain is reducible, and their codomain is reducible for any reducible element of the domain. But an impredicative dependent product may have an arbitrarily large domain, so this definition will not work. We will break out of the circularity by defining a weaker notion of reducibility for impredicative types that does not provide any control on the domain and codomain of dependent products.

Interestingly, this weakened reducibility means that we will not be able to prove the fundamental lemma directly on the economic rules we presented in chapter 3, which omit plenty of well-formedness hypotheses for the sake of readability (for instance, the rule Fun-Rel does not ask for well-formedness of the involved types). In their work, Abel *et al.* were able to define the model directly on the economic rules by collecting the well-formedness hypotheses in the reducibility predicates—but we are not, so we will construct our model using more verbose inference rules. Once the fundamental lemma is proved, we will be able to show that the economic rules of chapter 3 do in fact contain sufficient information to recover these additional hypotheses (see section 4.2).

This concludes our bird's eye view of the proof. It only remains to roll up our sleeves and unfold our strategy.

We will present a somewhat informal version of the proof where some technical details are left implicit, and encourage the reader who is interested in complete formal proofs to have a look at the Agda development.

4.1.2 Defining Reducibility

Following Abel *et al.* [30], our constructions starts with the definition of the four reducibility predicates, which are also annotated with a

sort in our version—see figure 4.1.

$\Gamma \Vdash_\ell A : \mathfrak{s}$	A is a reducible type of sort $\mathfrak{s}$ at level ℓ in context Γ
$\Gamma \Vdash_\ell A \equiv B : \mathfrak{s}$	A and B are reducibly equal types of sort $\mathfrak{s}$ at level ℓ in context Γ
$\Gamma \Vdash_\ell t : A : \mathfrak{s}$	t is a reducible term at level ℓ of type A in context Γ
$\Gamma \Vdash_\ell t \equiv u : A : \mathfrak{s}$	t and u are reducibly equal terms at level ℓ of type A in context Γ

Figure 4.1: The four reducibility predicates

These reducibility predicates are defined by induction-recursion, as illustrated in figure 4.2. The reducible types are defined inductively with eleven constructors, one corresponding to each basic type former of $\mathrm{CC}^{\mathrm{obs}}$ and one for neutral types. Each constructor takes as argument a proof of an auxiliary predicate of the form $\Gamma \Vdash_X t$ that packs information specific to the type X, that we will define below. The logical relations on terms and equalities are simultaneously defined by recursion on proofs of $\Gamma \Vdash_\ell A : \mathfrak{s}$, using more auxiliary predicates.

Remark that some types that we defined in chapter 3 do not appear in the definition of reducibility. First, we do not need to account for $\top$, existential types or propositional truncation, because they are defined in terms of $\bot$ and dependent products. What's more, the observational equality does not appear in the definition either. This is because it does not play the role of a type constructor, but rather that of a destructor that computes by pattern-matching on types. Therefore, all types with the shape $t \sim_A u$ will reduce to simpler types, unless they are neutral, in which case they are handled by the generic rule for neutral terms.

4.1.3 Reducibility for the Proof-Relevant Layer

We now give the definition of all these auxiliary predicates, starting with the predicates for neutral types.

Proof-relevant Neutral Types

$$\frac{\Gamma \vdash A \Rightarrow^* N : \mathscr{U}_i \qquad \mathrm{neutral}\, N}{\Gamma \Vdash_{\mathsf{ne}} A : \mathscr{U}_i}$$

Terms which reduce to neutral types are reducible types. In this rule, the premise $\Gamma \vdash A \Rightarrow^* N : \mathscr{U}_i$ means that A reduces to a term N in a finite number of steps (possibly zero), and that both A and N are of sort $\mathscr{U}_i$ in context Γ. When A is reducible to a neutral type, we also define:

Our reduction strategy is the weak-head reduction that we defined in Subsection 3.2.12.

- $\Gamma \Vdash_{\mathsf{ne}} A \equiv B : \mathscr{U}_i$ if there is a neutral term M such that $\Gamma \vdash B \Rightarrow^* M : \mathscr{U}_i$ and $\Gamma \vdash N \equiv M : \mathscr{U}_i$.
- $\Gamma \Vdash_{\mathsf{ne}} t : A : \mathscr{U}_i$ if there is a neutral term n such that $\Gamma \vdash t \Rightarrow^* n : N$.
- $\Gamma \Vdash_{\mathsf{ne}} t \equiv u : A : \mathscr{U}_i$ if there are neutral terms n, m such that $\Gamma \vdash t \Rightarrow^* n : N$ and $\Gamma \vdash u \Rightarrow^* m : N$, and $\Gamma \vdash n \equiv m : N$.

In other words, being a reducible inhabitant of a neutral type is simply reducing to a neutral term.

$$
\begin{array}{lcl}
\_ \Vdash_\ell \_ : \_ & : & (\Gamma : \mathsf{Context}) \to (t : \mathsf{Term}) \to (\mathfrak{s} : \mathsf{Sort}) \to \mathsf{Set} \\
\_ \Vdash_\ell \_ : \_ & ::= & \Vdash_{\mathsf{ne}} : \forall\ \Gamma\ t\ \mathfrak{s} \to (\Gamma \Vdash_{\mathsf{ne}} t : \mathfrak{s}) \to \Gamma \Vdash_\ell t : \mathfrak{s} \\
& | & \Vdash_{\Pi\mathsf{i}} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\Pi\mathsf{i}} t) \to \Gamma \Vdash_\ell t : \Omega \\
& | & \Vdash_{\bot} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\bot} t) \to \Gamma \Vdash_\ell t : \Omega \\
& | & \Vdash_{\mathsf{U}} : \forall\ \Gamma\ t\ i \to (\Gamma \Vdash_{\mathsf{U}} t : \mathscr{U}_i) \to \Gamma \Vdash_\ell t : \mathscr{U}_i \\
& | & \Vdash_{\Omega} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\Omega} t) \to \Gamma \Vdash_\ell t : \mathscr{U}_0 \\
& | & \Vdash_{\mathbb{N}} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\mathbb{N}} t) \to \Gamma \Vdash_\ell t : \mathscr{U}_0 \\
& | & \Vdash_{\Pi} : \forall\ \Gamma\ t\ i \to (\Gamma \Vdash_{\Pi} t : \mathscr{U}_i) \to \Gamma \Vdash_\ell t : \mathscr{U}_i \\
& | & \Vdash_{\Sigma} : \forall\ \Gamma\ t\ i \to (\Gamma \Vdash_{\Sigma} t : \mathscr{U}_i) \to \Gamma \Vdash_\ell t : \mathscr{U}_i \\
& | & \Vdash_{\Box} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\Box} t) \to \Gamma \Vdash_\ell t : \mathscr{U}_0 \\
& | & \Vdash_{\mathsf{Q}} : \forall\ \Gamma\ t\ i \to (\Gamma \Vdash_{\mathsf{Q}} t : \mathscr{U}_i) \to \Gamma \Vdash_\ell t : \mathscr{U}_i \\
& | & \Vdash_{\mathsf{Id}} : \forall\ \Gamma\ t\ i \to (\Gamma \Vdash_{\mathsf{Id}} t : \mathscr{U}_i) \to \Gamma \Vdash_\ell t : \mathscr{U}_i
\end{array}
$$

$$
\begin{array}{lcl}
\_ \Vdash_\ell \_ \equiv \_ : \_ & : & (\Gamma : \mathsf{Context}) \to (A : \mathsf{Term}) \to (B : \mathsf{Term}) \to (s : \mathsf{Sort}) \to \{\Gamma \Vdash_\ell A : s\} \to \mathsf{Set} \\
\Gamma \Vdash_\ell A \equiv B : s\ \{\Vdash_X\} & = & \Gamma \Vdash_X A \equiv B : s \qquad \text{for all } X \in \{\mathsf{ne}, \Pi\mathsf{i}, \bot, \mathsf{U}, \Omega, \mathbb{N}, \Pi, \Sigma, \Box, \mathsf{Q}, \mathsf{Id}\}
\end{array}
$$

$$
\begin{array}{lcl}
\_ \Vdash_\ell \_ : \_ : \_ & : & (\Gamma : \mathsf{Context}) \to (t : \mathsf{Term}) \to (A : \mathsf{Term}) \to (s : \mathsf{Sort}) \to \{\Gamma \Vdash_\ell A : s\} \to \mathsf{Set} \\
\Gamma \Vdash_\ell t : A : s\ \{\Vdash_X\} & = & \Gamma \Vdash_X t : A : s \qquad \text{for all } X \in \{\mathsf{ne}, \Pi\mathsf{i}, \bot, \mathsf{U}, \Omega, \mathbb{N}, \Pi, \Sigma, \Box, \mathsf{Q}, \mathsf{Id}\}
\end{array}
$$

$$
\begin{array}{lcl}
\_ \Vdash_\ell \_ \equiv \_ : \_ : \_ & : & (\Gamma : \mathsf{Context}) \to (t\ u : \mathsf{Term}) \to (A : \mathsf{Term}) \to (s : \mathsf{Sort}) \to \{\Gamma \Vdash_\ell A : s\} \to \mathsf{Set} \\
\Gamma \Vdash_\ell t \equiv u : A : s\ \{\Vdash_X\} & = & \Gamma \Vdash_X t \equiv u : A : s \qquad \text{for all } X \in \{\mathsf{ne}, \Pi\mathsf{i}, \bot, \mathsf{U}, \Omega, \mathbb{N}, \Pi, \Sigma, \Box, \mathsf{Q}, \mathsf{Id}\}
\end{array}
$$

Figure 4.2: Inductive-recursive presentation of reducibility

Natural Numbers

$$\frac{\Gamma \vdash A \Rightarrow^* \mathbb{N} : \mathscr{U}_0}{\Gamma \Vdash_{\mathbb{N}} A}$$

Terms which reduce to $\mathbb{N}$ are reducible types. In case A is reducible to $\mathbb{N}$, we also define:

- $\Gamma \Vdash_{\mathbb{N}} A \equiv B$ if $\Gamma \vdash B \Rightarrow^* \mathbb{N} : \mathscr{U}_0$.
- $\Gamma \Vdash_{\mathbb{N}} t : A$ if there is a normal form t' such that $\Gamma \vdash t \Rightarrow^* t' : \mathbb{N}$ and $\Gamma \Vdash_{\mathbb{N}\mathsf{t}} t'$, which is inductively defined by

$$\frac{}{\Gamma \Vdash_{\mathbb{N}\mathsf{t}} 0} \qquad \frac{\Gamma \Vdash_{\mathbb{N}\mathsf{t}} t}{\Gamma \Vdash_{\mathbb{N}\mathsf{t}} \mathsf{S}\ t} \qquad \frac{\Gamma \vdash n : \mathbb{N} \qquad n \text{ is neutral}}{\Gamma \Vdash_{\mathbb{N}\mathsf{t}} n}$$

- $\Gamma \Vdash_{\mathbb{N}} t \equiv u : A$ if there are normal forms t', u' such that $\Gamma \vdash t \Rightarrow^* t' : \mathbb{N}$ and $\Gamma \vdash u \Rightarrow^* u' : \mathbb{N}$, and $\Gamma \Vdash_{\mathbb{N}\mathsf{t}=} t' \equiv u'$, which is inductively defined by

$$\frac{}{\Gamma \Vdash_{\mathbb{N}\mathsf{t}=} 0 \equiv 0} \qquad \frac{\Gamma \Vdash_{\mathbb{N}\mathsf{t}=} t \equiv u}{\Gamma \Vdash_{\mathbb{N}\mathsf{t}=} \mathsf{S}\ t \equiv \mathsf{S}\ u}$$

$$\frac{\Gamma \vdash n \equiv m : \mathbb{N} \qquad n, m \text{ are neutral}}{\Gamma \Vdash_{\mathbb{N}\mathsf{t}=} n \equiv m}$$

These definitions mean that a term is a reducible natural number when it reduces to either zero, a neutral term or a successor of a reducible natural number.

Predicative Universes

$$\frac{\Gamma \vdash A \Rightarrow^* \mathscr{U}_i : \mathscr{U}_{i+1}}{\Gamma \Vdash_{\cup} A : \mathscr{U}_{i+1}}\, i<\ell$$

Terms which reduce to a predicative universe are reducible types. In case A is reducible to a predicative universe, we also define:

- $\Gamma \Vdash_{\cup} A \equiv B : \mathscr{U}_{i+1}$ if $\Gamma \vdash B \Rightarrow^* \mathscr{U}_i : \mathscr{U}_{i+1}$.
- $\Gamma \Vdash_{\cup} t : A : \mathscr{U}_{i+1}$ if there is a normal form t' such that $\Gamma \vdash t \Rightarrow^* t' : \mathscr{U}_i$, and $\Gamma \Vdash_i t : \mathscr{U}_i$
  (which is already defined by induction hypothesis, since $i < \ell$).
- $\Gamma \Vdash_{\cup} t \equiv u : A : \mathscr{U}_{i+1}$ if there are normal forms t', u' such that $\Gamma \vdash t \Rightarrow^* t' : \mathscr{U}_i$ and $\Gamma \vdash u \Rightarrow^* u' : \mathscr{U}_i$, and $\Gamma \Vdash_i t : \mathscr{U}_i$, $\Gamma \Vdash_i u : \mathscr{U}_i$, and $\Gamma \Vdash_i t \equiv u : \mathscr{U}_i$.

According to these definitions, being a reducible inhabitant of a predicative universe is the same as being a proof-relevant reducible type: either a neutral, or a dependent product, etc. This is the only case that recursively calls the definition of reducibility at a lower level, forcing us to define the whole affair by induction on ℓ.

Proof-relevant Dependent Products

$$\frac{\begin{array}{c}\Gamma \vdash A \Rightarrow^* \Pi^{\mathfrak{s},\mathscr{U}_i}(x : F).\, G : \mathscr{U}_{\max(\mathfrak{s},i)} \\ \Gamma \vdash F : \mathfrak{s} \qquad \Gamma, x : F \vdash G : \mathscr{U}_i \qquad \forall (\rho : \Delta \sqsubseteq \Gamma),\ \Delta \Vdash_\ell F[\rho] : \mathfrak{s} \\ \forall (\rho : \Delta \sqsubseteq \Gamma),\ \Delta \Vdash_\ell a : F[\rho] : \mathfrak{s} \ \to\ \Delta \Vdash_\ell G[\rho, a] : \mathscr{U}_i \\ \forall (\rho : \Delta \sqsubseteq \Gamma), \Delta \Vdash_\ell a : F[\rho] : \mathfrak{s} \to \Delta \Vdash_\ell b : F[\rho] : \mathfrak{s} \to \Delta \Vdash_\ell a \equiv b : F[\rho] : \mathfrak{s} \to \Delta \Vdash_\ell G[\rho, a] \equiv G[\rho, b] : \mathscr{U}_i\end{array}}{\Gamma \Vdash_\Pi A : \mathscr{U}_{\max(\mathfrak{s},i)}}$$

This rule describes the condition for A to be reducible to a proof-relevant dependent product. We introduced quite a bit of notation here, so we go over the premises one by one. The first three premises state that A reduces to a dependent product in a finite number of steps, and that the domain and codomain of the dependent product are well-formed types. The fourth premise asks for the domain F to be a reducible type under any weakening: the notation $(\rho : \Delta \sqsubseteq \Gamma)$ means that ρ is a weakening from a context Δ to the context Γ, and the notation $F[\rho]$ represents the result of weakening the free variables of F. The fifth premise says that given any weakening ρ and a term a which is a reducible inhabitant of $F[\rho]$, the term we obtain by applying the weakening ρ to all free variables of G except for x, and then substituting a for x is reducible. This term is noted $G[\rho, a]$. Finally, the last premise states that under any weakening, applying G to reducibly equal inhabitants of F results in two reducibly equal types.

Despite the similarity with an inclusion symbol, the notation $\Delta \sqsubseteq \Gamma$ means that all types of Γ appear in Δ.

The reader might be wondering why we generalize the last three premises under any weakening. The simple explanation is that we want reducibility to be stable under weakening, and we will not be able to prove it by induction on the definition because of the negative occurences—so we generalize any premise that mentions reducibility on the left of an arrow.

Thus, reducibility has the structure of a *presheaf* over the category of weakenings. These generalized hypotheses are really an embodiment of the presheaf exponential.

When A is reducible to a dependent product, we define:

- $\Gamma \Vdash_\Pi A \equiv B : \mathscr{U}_{\max(\mathfrak{s},i)}$ if there are terms F' and G' such that

- $\Gamma \vdash B \Rightarrow^* \Pi(x : F').\, G' : \mathscr{U}_{\max(\mathfrak{s},i)}$
- $\Gamma \vdash \Pi(x : F).\, G \equiv \Pi(x : F').\, G' : \mathscr{U}_{\max(\mathfrak{s},i)}$
- $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell F[\rho] \equiv F'[\rho] : \mathfrak{s}$
- $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell a : F[\rho] : \mathfrak{s}$
  $\to \Delta \Vdash_\ell G[\rho, a] \equiv G'[\rho, a] : \mathscr{U}_i.$

▶ $\Gamma \Vdash_\Pi t : A : \mathscr{U}_{\max(\mathfrak{s},i)}$ if there is a normal form t' such that

- $\Gamma \vdash t \Rightarrow^* t' : \Pi(x : F).\, G$
- $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell a : F[\rho] : \mathfrak{s}$
  $\to \Delta \Vdash_\ell t'[\rho]\ a : G[\rho, a] : \mathscr{U}_i$
- $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell a : F[\rho] : \mathfrak{s}$
  $\to \Delta \Vdash_\ell b : F[\rho] : \mathfrak{s} \to \Delta \Vdash_\ell a \equiv b : F[\rho] : \mathfrak{s}$
  $\to \Delta \Vdash_\ell t'[\rho]\ a \equiv t'[\rho]\ b : G[\rho, a] : \mathscr{U}_i.$

▶ $\Gamma \Vdash_\Pi t \equiv u : A : \mathscr{U}_{\max(\mathfrak{s},i)}$ if there are normal forms t', u' such that

- $\Gamma \vdash t \Rightarrow^* t' : \Pi(x : F).\, G$ and $\Gamma \vdash u \Rightarrow^* u' : \Pi(x : F).\, G$
- $\Gamma \vdash t' \equiv u' : \Pi(x : F).\, G$
- $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell a : F[\rho] : \mathfrak{s}$
  $\to \Delta \Vdash_\ell t'[\rho]\ a \equiv u'[\rho]\ a : G[\rho, a] : \mathscr{U}_i.$

These definitions deem a term to be a reducible dependent function when it reduces to either a neutral term or a lambda-abstraction, and it sends reducible inhabitant of the domain to reducible inhabitants of the codomain. Note that in the three definitions above, we talk about reducibility of inhabitants of the domain and the codomain. We know that these reducibility predicates are well-defined because we assumed that the domain and the codomain are reducible types when we supposed that A is reducible to a dependent product.

Dependent Sums

$$\frac{\begin{array}{c}\Gamma \vdash A \Rightarrow^* \Sigma^{\mathscr{U}_i,\mathscr{U}_j}(x : F).\, G : \mathscr{U}_{\max(i,j)} \\ \Gamma \vdash F : \mathscr{U}_i \qquad \Gamma, x : F \vdash G : \mathscr{U}_j \qquad \forall (\rho : \Delta \sqsubseteq \Gamma),\ \Delta \Vdash_\ell F[\rho] : \mathscr{U}_i \\ \forall (\rho : \Delta \sqsubseteq \Gamma),\ \Delta \Vdash_\ell a : F[\rho] : \mathscr{U}_i \ \to\ \Delta \Vdash_\ell G[\rho, a] : \mathscr{U}_j \\ \forall (\rho : \Delta \sqsubseteq \Gamma), \Delta \Vdash_\ell a : F[\rho] : \mathscr{U}_i \to \Delta \Vdash_\ell b : F[\rho] : \mathscr{U}_i \to \Delta \Vdash_\ell a \equiv b : F[\rho] : \mathscr{U}_i \to \Delta \Vdash_\ell G[\rho, a] \equiv G[\rho, b] : \mathscr{U}_j\end{array}}{\Gamma \Vdash_\Sigma A : \mathscr{U}_{\max(i,j)}}$$

When A reduces to a dependent sum $\Sigma(x : F).\, G$ such that F and G verify the same conditions as for the dependent product, then A is a reducible type. In this case, we define:

▶ $\Gamma \Vdash_\Sigma A \equiv B : \mathscr{U}_{\max(i,j)}$ if there are terms F' and G' such that

- $\Gamma \vdash B \Rightarrow^* \Sigma(x : F').\, G' : \mathscr{U}_{\max(i,j)}$
- $\Gamma \vdash \Sigma(x : F).\, G \equiv \Sigma(x : F').\, G' : \mathscr{U}_{\max(i,j)}$
- $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell F[\rho] \equiv F'[\rho] : \mathscr{U}_i$
- $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell a : F[\rho] : \mathscr{U}_i$
  $\to \Delta \Vdash_\ell G[\rho, a] \equiv G'[\rho, a] : \mathscr{U}_j.$

▶ $\Gamma \Vdash_\Sigma t : A : \mathscr{U}_{\max(i,j)}$ if there is a normal form t' such that

- $\Gamma \vdash t \Rightarrow^* t' : \Sigma(x : F).\, G$
- $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell \mathsf{proj}_1(t'[\rho]) : F[\rho] : \mathscr{U}_i$
- $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell \mathsf{proj}_2(t'[\rho]) : G[\rho, \mathsf{proj}_1(t'[\rho])] : \mathscr{U}_j.$

- $\Gamma \Vdash_\Sigma t \equiv u : A : \mathscr{U}_{\max(i,j)}$ if there are normal forms t', u' such that
  - $\Gamma \vdash t \Rightarrow^* t' : \Sigma(x : F).\, G$ and $\Gamma \vdash u \Rightarrow^* u' : \Sigma(x : F).\, G$
  - $\Gamma \vdash t' \equiv u' : \Sigma(x : F).\, G$
  - $\forall(\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell \mathsf{proj}_1(t'[\rho]) \equiv \mathsf{proj}_1(u'[\rho]) : F[\rho] : \mathscr{U}_i$
  - $\forall(\rho : \Delta \sqsubseteq \Gamma).$
    $\Delta \Vdash_\ell \mathsf{proj}_2(t'[\rho]) \equiv \mathsf{proj}_2(u'[\rho]) : G[\rho, \mathsf{proj}_1(t'[\rho])] : \mathscr{U}_j.$

Thus, a term is a reducible inhabitant of A when it reduces to either a neutral term or a dependent pair, and both of its projections are reducible.

Box types

$$\frac{\Gamma \vdash A \Rightarrow^* \Box A' : \mathscr{U}_0 \qquad \Gamma \vdash A' : \Omega \qquad \Gamma \Vdash_\ell A' : \Omega}{\Gamma \Vdash_\Box A}$$

If A reduces to a boxed reducible proposition, then A is a reducible type. In which case, we define:

- $\Gamma \Vdash_\Box A \equiv B : \mathscr{U}_0$ if there is a term B' such that
  - $\Gamma \vdash B \Rightarrow^* \Box B' : \mathscr{U}_0$
  - $\Gamma \Vdash_\ell A' \equiv B' : \Omega$
- $\Gamma \Vdash_\Box t : A : \mathscr{U}_0$ if there is a normal form t' such that $\Gamma \vdash t \Rightarrow^* t' : \Box A'$ and $\Gamma \Vdash_{\Box \mathsf{t}}\ t'$, which is defined by

$$\frac{\Gamma \vdash t : A'}{\Gamma \Vdash_{\Box \mathsf{t}}\ \diamond t} \qquad \frac{\Gamma \vdash n : \Box A' \qquad n \text{ is neutral}}{\Gamma \Vdash_{\Box \mathsf{t}}\ n}$$

- $\Gamma \Vdash_\Box t \equiv u : A : \mathscr{U}_i$ if there are normal forms t', u' such that $\Gamma \vdash t \Rightarrow^* t' : \Box A'$, $\Gamma \vdash u \Rightarrow^* u' : \Box A'$, and $\Gamma \Vdash_{\Box \mathsf{t}=}\ t' \equiv u'$, which is defined by

$$\frac{\Gamma \vdash t : A' \qquad \Gamma \vdash u : A'}{\Gamma \Vdash_{\Box \mathsf{t}=}\ \diamond t \equiv \diamond u} \qquad \frac{\Gamma \vdash n \equiv m : \Box A' \qquad n, m \text{ are neutral}}{\Gamma \Vdash_{\Box \mathsf{t}=}\ n \equiv m}$$

In other words, an inhabitant of a box type is reducible when it reduces to either a boxed proof, or a neutral term.

Quotients

$$\frac{\begin{array}{c}\Gamma \vdash A \Rightarrow^* A'/(R, R_r, R_s, R_t) : \mathscr{U}_i \qquad \Gamma \vdash A' : \mathscr{U}_i \\ \forall(\rho : \Delta \sqsubseteq \Gamma).\ \Gamma \Vdash_\ell A'[\rho] : \mathscr{U}_i \qquad \Gamma \Vdash_\ell R : A' \to A' \to \Omega : \mathscr{U}_i \\ \Gamma \vdash R_r : \Pi(x : A').\, R\, x\, x \qquad \Gamma \vdash R_s : \Pi(x, y : A').\, R\, x\, y \to R\, y\, x \\ \Gamma \vdash R_t : \Pi(x, y, z : A').\, R\, x\, y \to R\, y\, z \to R\, x\, z\end{array}}{\Gamma \Vdash_{\mathsf{Q}} A : \mathscr{U}_i}$$

This rule stipulates that A is a reducible type if it reduces to a quotient type A'/R, its underlying type A' is reducible under any substitution, and R is a reducible relation on A'. Note that to enunciate this last condition, we need to make sure that the type of relations on A' is reducible, which is deduced from the reducibility of A' and of Ω.

If A is reducible to a quotient type, we define:

- $\Gamma \Vdash_{\mathrm{Q}} A \equiv B : \mathcal{U}_i$ if there are terms B', Q, Q_r, Q_s, Q_t such that
  - $\Gamma \vdash B \Rightarrow^* B'/(Q, Q_r, Q_s, Q_t) : \mathcal{U}_i$
  - $\forall (\rho : \Delta \sqsubseteq \Gamma).\ \Delta \Vdash_\ell A'[\rho] \equiv B'[\rho] : \mathcal{U}_i$
  - $\Gamma \Vdash_\ell R \equiv Q : A' \to A' \to \Omega : \mathcal{U}_i$.
- $\Gamma \Vdash_{\mathrm{Q}} t : A : \mathcal{U}_i$ if there is a normal form t' such that $\Gamma \vdash t \Rightarrow^* t' : A'/R$ and $\Gamma \Vdash_{\mathrm{Qt}} t'$, which is defined by

$$\frac{\Gamma \Vdash_\ell t : A' : \mathcal{U}_i}{\Gamma \Vdash_{\mathrm{Qt}} \pi(t)} \qquad \frac{\Gamma \vdash n : A'/R \qquad n \text{ is neutral}}{\Gamma \Vdash_{\mathrm{Qt}} n}$$

- $\Gamma \Vdash_{\mathrm{Q}} t \equiv u : A : \mathcal{U}_0$ if there are normal forms t', u' such that $\Gamma \vdash t \Rightarrow^* t' : A'/R$, $\Gamma \vdash u \Rightarrow^* u' : A'/R$, and $\Gamma \Vdash_{\mathrm{Qt}=} t' \equiv u'$, which is defined by

$$\frac{\Gamma \Vdash_\ell t \equiv u : A' : \mathcal{U}_i}{\Gamma \Vdash_{\mathrm{Qt}=} \pi(t) \equiv \pi(u)} \qquad \frac{\Gamma \vdash n \equiv m : A'/R \qquad n, m \text{ are neutral}}{\Gamma \Vdash_{\mathrm{Qt}=} n \equiv m}$$

In other words, an inhabitant of a quotient type is reducible when it reduces to either a projection of a reducible term, or a neutral term.

Inductive Identity Types

$$\frac{\Gamma \vdash A \Rightarrow^* \mathsf{Id}(A', t, u) : \mathcal{U}_i \qquad \Gamma \Vdash_\ell A' : \mathcal{U}_i \qquad \Gamma \Vdash_\ell t : A' : \mathcal{U}_i \qquad \Gamma \Vdash_\ell u : A' : \mathcal{U}_i}{\Gamma \Vdash_{\mathsf{Id}} A : \mathcal{U}_i}$$

When A reduces to an identity type between two reducible inhabitants of a reducible type, then A is reducible. In this case, we define:

- $\Gamma \Vdash_{\mathsf{Id}} A \equiv B : \mathcal{U}_i$ if there are terms B', t', u' such that
  - $\Gamma \vdash B \Rightarrow^* \mathsf{Id}(B', t', u') : \mathcal{U}_i$
  - $\Gamma \Vdash_\ell A' \equiv B' : \mathcal{U}_i$
  - $\Gamma \Vdash_\ell t \equiv t' : A' : \mathcal{U}_i$
  - $\Gamma \Vdash_\ell u \equiv u' : A' : \mathcal{U}_i$.
- $\Gamma \Vdash_{\mathsf{Id}} e : A : \mathcal{U}_i$ if there is a normal form e' such that $\Gamma \vdash e \Rightarrow^* e' : \mathsf{Id}(A', t, u)$ and $\Gamma \Vdash_{\mathsf{Idt}} e'$, which is defined by

$$\frac{}{\Gamma \Vdash_{\mathsf{Idt}} \mathsf{Idrefl}(t)} \qquad \frac{\Gamma \vdash e : t \sim_{A'} u}{\Gamma \Vdash_{\mathsf{Idt}} \mathsf{Idpath}(e)}$$

$$\frac{\Gamma \vdash n : \mathsf{Id}(A', t, u) \qquad n \text{ is neutral}}{\Gamma \Vdash_{\mathsf{Idt}} n}$$

- $\Gamma \Vdash_{\mathsf{Id}} e \equiv f : A : \mathcal{U}_i$ if there are normal forms e', f' such that $\Gamma \vdash e \Rightarrow^* e' : \mathsf{Id}(A', t, u)$ and $\Gamma \vdash f \Rightarrow^* f' : \mathsf{Id}(A', t, u)$, and $\Gamma \Vdash_{\mathsf{Idt}=} e' \equiv f'$, which is inductively defined by

$$\frac{}{\Gamma \Vdash_{\mathsf{Idt}=} \mathsf{Idrefl}(t) \equiv \mathsf{Idrefl}(t)} \qquad \frac{\Gamma \vdash e, f : t \sim_{A'} u}{\Gamma \Vdash_{\mathsf{Idt}=} \mathsf{Idpath}(e) \equiv \mathsf{Idpath}(f)}$$

$$\frac{\Gamma \vdash n \equiv m : \mathsf{Id}(A', t, u) \qquad n, m \text{ are neutral}}{\Gamma \Vdash_{\mathsf{Idt}=} n \equiv m}$$

Which states that a term is a reducible witness of the inductive equality when it reduces to either reflexivity, Idpath, or a neutral term.

4.1.4 Reducibility for the Proof-Irrelevant Layer

We now turn to the definition of reducibility for the proof-irrelevant fragment, which must be defined independently from the rest of the model to ensure a well-founded definition.

Since there is no computation whatsoever in proof-irrelevant types, their inhabitants do not interact with other terms. This allows us to give a generic definition for reducibility of terms and term equality, that will work for any $X \in \{\mathrm{ne}, \bot, \Pi\mathrm{i}\}$:

- $\Gamma \Vdash_X t : A : \Omega$ when $\Gamma \vdash t : A$.
- $\Gamma \Vdash_X t \equiv u : A : \Omega$ when $\Gamma \vdash t : A$ and $\Gamma \vdash u : A$.

which means that if A is in Ω, then the model does not need to collect any information on inhabitants of A, save for the fact that they are well-typed. And this applies to reducible equality too, for any two inhabitants of A are always convertible. It only remains to define the reducibility of types and type equality for neutrals, $\bot$ and the impredicative dependent products.

Proof-irrelevant Neutral Types

$$\frac{\Gamma \vdash A \Rightarrow^* N : \Omega \qquad \mathrm{neutral}\, N}{\Gamma \Vdash_{\mathrm{ne}} A : \Omega}$$

When A is reducible to a neutral proposition, we define reducible equality to A just as in the case of neutral proof-relevant types: $\Gamma \Vdash_{\mathrm{ne}} A \equiv B : \Omega$ if there is a neutral term M such that $\Gamma \vdash B \Rightarrow^* M : \Omega$ and $\Gamma \vdash N \equiv M : \Omega$.

Empty type The case of the empty type is easy, as it does not recursively call the logical relation.

$$\frac{\Gamma \vdash A \Rightarrow^* \bot : \Omega}{\Gamma \Vdash_{\bot} A}$$

When A is reducible to the empty type, we define $\Gamma \Vdash_{\bot} A \equiv B$ as $\Gamma \vdash B \Rightarrow^* \bot : \Omega$.

Impredicative Dependent Function Types For the impredicative function types, the situation is more complex, as we cannot reproduce the definition of their predicative counterpart: it involves a recursive call to the logical relation for the domain and codomain types, which might live in a higher universe than the function type. Consequently, we go for minimalism and only collect the fact that the domain and the codomain are well-typed.

$$\frac{\Gamma \vdash A \Rightarrow^* \Pi^{\mathfrak{s},\Omega}(x : F).\, G : \Omega \qquad \Gamma \vdash F : \mathscr{U} \qquad \Gamma, x : F \vdash G : \Omega}{\Gamma \Vdash_{\Pi \mathrm{i}} A}$$

Similarly, when A is reducible to an impredicative function type, we define reducible equality of A and B as the fact that B reduces to a convertible impredicative function type:

$$\frac{\Gamma \vdash B \Rightarrow^* \Pi(x : F').\, G' : \Omega \qquad \Gamma \vdash \Pi(x : F).\, G \equiv \Pi(x : F').\, G' : \Omega}{\Gamma \Vdash_{\Pi \mathrm{i}} A \equiv B}$$

These definitions do not recursively call the logical relation, but as a result the collected invariants are much weaker than those for relevant dependent function types. We will discuss this in the proof of the fundamental lemma in Subsection 4.1.7.

The Impredicative Universe

$$\frac{\Gamma \vdash A \Rightarrow^* \Omega : \mathscr{U}_0}{\Gamma \Vdash_{\Omega} A}$$

When A is reducible to the impredicative universe, we define:

- $\Gamma \Vdash_{\Omega} A \equiv B : \mathscr{U}_0$ if $\Gamma \vdash B \Rightarrow^* \Omega : \mathscr{U}_0$.
- $\Gamma \Vdash_{\Omega} t : A : \mathscr{U}_0$ if there is a normal form t' such that $\Gamma \vdash t \Rightarrow^* t' : \Omega$, and $\Gamma \Vdash_X t : \Omega$ for some $X \in \{\mathrm{ne}, \bot, \Pi\mathrm{i}\}$.
- $\Gamma \Vdash_{\Omega} t \equiv u : A : \mathscr{U}_0$ if there are normal forms t', u', as well as $X \in \{\mathrm{ne}, \bot, \Pi\mathrm{i}\}$ such that
  - $\Gamma \vdash t \Rightarrow^* t' : \Omega$ and $\Gamma \vdash u \Rightarrow^* u' : \Omega$
  - $\Gamma \Vdash_X t : \Omega$ and $\Gamma \Vdash_X u : \Omega$
  - $\Gamma \Vdash_X t \equiv u : \Omega$.

Being a reducible proposition amounts to reducing to either a neutral term, the false proposition, or an impredicative dependent product.

4.1.5 Auxiliary Lemmas on Reducibility

Now that we have completed our definition of reducibility, we prove some auxiliary lemmas:

Lemma 4.1.1 (Escape lemma)

1. *If* $\Gamma \Vdash_{\ell} t : A : \mathfrak{s}$ *then* $\Gamma \vdash t : A : \mathfrak{s}$.
2. *If* $\Gamma \Vdash_{\ell} t \equiv u : A : \mathfrak{s}$ *then* $\Gamma \vdash t \equiv u : A : \mathfrak{s}$.

Lemma 4.1.2

1. *Given a context* Γ*, the reducible equality* $\Gamma \Vdash_{\ell} A \equiv B : \mathfrak{s}$ *is reflexive, symmetric and transitive.*
2. *Given a context* Γ *and a reducible type* $\Gamma \Vdash_{\ell} A : \mathfrak{s}$*, the reducible equality* $\Gamma \Vdash_{\ell} t \equiv u : A : \mathfrak{s}$ *is also reflextive, symmetric and transitive.*

Lemma 4.1.3 (Conversion) *Given* $\Gamma \Vdash_\ell A \equiv B : \mathbb{s}$,

1. $\Gamma \Vdash_\ell t : A : \mathbb{s}$ *if and only if* $\Gamma \Vdash_\ell t : B : \mathbb{s}$.
2. $\Gamma \Vdash_\ell t \equiv u : A : \mathbb{s}$ *if and only if* $\Gamma \Vdash_\ell t \equiv u : B : \mathbb{s}$.

Lemma 4.1.4 (Neutrals are reducible) *Let A be a reducible type and n, m be neutral terms.*

1. *If* $\Gamma \vdash n : A$, *then* $\Gamma \Vdash_\ell n : A : \mathbb{s}$.
2. *If* $\Gamma \vdash n \equiv m : A$, *then* $\Gamma \Vdash_\ell n \equiv m : A : \mathbb{s}$.

Lemma 4.1.5 (Weak head expansion)

1. *If* $\Gamma \Vdash_\ell B : \mathbb{s}$ *and* $\Gamma \vdash A \Rightarrow B : \mathbb{s}$, *then* $\Gamma \Vdash_\ell A : \mathbb{s}$ *and* $\Gamma \Vdash_\ell A \equiv B : \mathbb{s}$.
2. *If* $\Gamma \Vdash_\ell u : A : \mathbb{s}$ *and* $\Gamma \vdash t \Rightarrow u : A$, *then* $\Gamma \Vdash_\ell t : A : \mathbb{s}$ *and* $\Gamma \Vdash_\ell t \equiv u : A : \mathbb{s}$.

All these lemmas are proved by a straightforward induction. The interested reader can find formal proofs in *[Properties.agda]*.

4.1.6 Building a Model From Reducibility

While reducibility is the main ingredient to our normalization proof, it is not strong enough to define a model of $\mathrm{CC}^{\mathrm{obs}}$. For instance, reducibility is not preserved by the lambda-abstraction rule Fun-Rel:

$$\textsc{Fun-Rel}\quad \frac{\Gamma, x : A : \mathbb{s} \vdash t : B : \mathcal{U}_i}{\Gamma \vdash \lambda(x : A).\, t : \Pi^{\mathbb{s},\mathcal{U}_i}(x : A).\, B : \mathcal{U}_{\max(i,\mathbb{s})}}$$

For the lambda-abstraction to be a reducible element of the dependent product, we need to prove that for any term a which is a reducible inhabitant of A, the term $t[x := a]$ is a reducible inhabitant of $B[x := a]$. But our induction hypothesis only provides that t is reducible, and we have no obvious way to show that reducibility is stable under substitution by reducible terms.

We use the weak-head expansion lemma to reduce the proof of reducibility of $(\lambda(x : A).\, t)\, a$ to a proof of reducibility of $t[x := a]$

Thus, following Abel et al. [30], we define the notion of *validity*, which is the closure of reducibility under substitution *[Substitution.agda]*. Validity consists of seven predicates.

[30]: Abel et al. (2018), "Decidability of Conversion for Type Theory in Type Theory"

- The unary predicate on substitutions $\Delta \Vdash^s \_ : \Gamma$ defines *valid substitutions* from Δ to Γ as telescopes of reducible terms.
- The binary predicate on substitutions $\Delta \Vdash^s \_ \equiv \_ : \Gamma$ defines *valid equality* of substitutions from Δ to Γ as telescopes of reducible equalities.
- The unary predicate on terms $\Gamma \Vdash^v_\ell \_ : \mathbb{s}$ defines *valid types* of sort $\mathbb{s}$ in a context Γ as types that are reducible under any valid substitution, and that preserve valid equality of substitutions.
- The binary relation on terms $\Gamma \Vdash^v_\ell \_ \equiv \_ : \mathbb{s}$, defines *valid equality* between two valid types in context Γ as reducible equality under any valid substitution.

We write $\Delta \to \Gamma$ for the type of substitutions from Δ to Γ. Given a substitution σ and a term t, we can substitute the free variables of t to produce the term $t[\sigma]$.

$$
\begin{array}{lcl}
\Vdash^{v} \_ & : & (\Gamma : \mathsf{Context}) \to \mathsf{Set} \\
\Vdash^{v} \_ & ::= & \varepsilon \ : \ \Vdash^{v} \bullet \\
& \mid & \_,\_ \ : \ ([\Gamma] : \Vdash^{v} \Gamma) \to \Gamma \Vdash^{v}_{\ell} A : \mathbb{s} \to \Vdash^{v} \Gamma, x : A : \mathbb{s}
\end{array}
$$

$$
\begin{array}{lcl}
\_ \Vdash^{v}_{\ell} \_ : \_ & : & (\Gamma : \mathsf{Context}) \to (A : \mathsf{Term}) \to (\mathbb{s} : \mathsf{Sort}) \to \{\Vdash^{v} \Gamma\} \to \mathsf{Set} \\
\Gamma \Vdash^{v}_{\ell} A : \mathbb{s} & = & \forall\ \sigma\ \tau \to \Delta \Vdash^{s} \sigma : \Gamma \ \to \ \Delta \Vdash_{\ell} A[\sigma] : \mathbb{s} \\
& & \times \quad \Delta \Vdash^{s} \sigma \equiv \tau : \Gamma \ \to \ \Delta \Vdash_{\ell} A[\sigma] \equiv A[\tau] : \mathbb{s}
\end{array}
$$

$$
\begin{array}{lcl}
\_ \Vdash^{s} \_ : \_ & : & (\Gamma\ \Delta : \mathsf{Context}) \to (\sigma : \Delta \to \Gamma) \to \{\Vdash^{v} \Gamma\} \to \mathsf{Set} \\
\Delta \Vdash^{s} \sigma : \bullet & = & \top \\
\Delta \Vdash^{s} t, \sigma : (\Gamma, x : A : \mathbb{s}) & = & \Delta \Vdash^{s} \sigma : \Gamma \ \times \ \Delta \Vdash_{\ell} t : A[\sigma] : \mathbb{s}
\end{array}
$$

$$
\begin{array}{lcl}
\_ \Vdash^{s} \_ \equiv \_ : \_ & : & (\Gamma\ \Delta : \mathsf{Context}) \to (\sigma\ \tau : \Delta \to \Gamma) \to \{\Vdash^{v} \Gamma\} \to \mathsf{Set} \\
\Delta \Vdash^{s} \sigma \equiv \tau : \bullet & = & \top \\
\Delta \Vdash^{s} t, \sigma \equiv u, \tau : (\Gamma, x : A : \mathbb{s}) & = & \Delta \Vdash^{s} \sigma \equiv \tau : \Gamma \ \times \ \Delta \Vdash_{\ell} t \equiv u : A[\sigma] : \mathbb{s}
\end{array}
$$

Figure 4.3: Inductive recursive definition of validity of contexts, types and substitutions

- The unary predicate on terms $\Gamma \Vdash^{v}_{\ell} \_ : A : \mathbb{s}$ defines *valid inhabitants* of a valid type A in context Γ as terms that are reducible inhabitants of A under any valid substitution, and that preserve valid equality of substitutions.
- The binary relation on terms $\Gamma \Vdash^{v}_{\ell} \_ \equiv \_ : A : \mathbb{s}$ defines *valid equality* between two valid inhabitants of a valid type A in context Γ as reducible equality under any valid substitution.
- The unary predicate on contexts $\Vdash^{v} \_$, defines *valid contexts*, as lists of valid types.

In figure 4.3, we use induction recursion to define a valid context inductively as a telescope of valid types; while we simultaneously define valid types, valid substitutions and valid equality of substitutions by recursion on a valid context. The valid type equality, valid terms and valid term equality do not need to be part of the inductive-recursive block and can be defined *a posteriori* in a straightforward manner—see *[Substitution.agda]*.

4.1.7 The Fundamental Lemma

The fundamental lemma is the proof that validity is preserved by all inference rules of $\mathrm{CC}^{\mathrm{obs}}$, or equivalently, that validity defines a model of $\mathrm{CC}^{\mathrm{obs}}$.

Lemma 4.1.6 (Fundamental lemma for terms *[Fundamental.agda]*) *If $\Gamma \vdash t : A : \mathbb{s}$, then there is a level ℓ such that $\Vdash^{v} \Gamma$, $\Gamma \Vdash^{v}_{\ell} A : \mathscr{U}$ and $\Gamma \Vdash^{v}_{\ell} t : A : \mathscr{U}$.*

The lemma is proved by a mutual induction on derivations for well-formed contexts, types, type equality, terms and term equality. Our various extensions of the model require numerous changes to the proof of Abel *et al.*, but most of these do not pose any major difficulty—although the proof is by no means trivial, as the arguments have to be spelled out in excruciating detail.

Therefore, we focus on the three features of CC$^{\text{obs}}$ that are the most disruptive: the observational equality, the cast operator, and the impredicative dependent products (Rules Eq-Form, Cast, and Π-Irr-Form).

Lemma 4.1.7 (Reducibility of cast) *Given any level ℓ, in a well-formed context Γ, if:*

1. $\Gamma \Vdash_\ell A : s$ *and* $\Gamma \Vdash_\ell A' : s$ *and* $\Gamma \Vdash_\ell A \equiv A' : s$
2. $\Gamma \Vdash_\ell B : s$ *and* $\Gamma \Vdash_\ell B' : s$ *and* $\Gamma \Vdash_\ell B \equiv B' : s$
3. $\Gamma \vdash e : A \sim_s B$ *and* $\Gamma \vdash e' : A' \sim_s B'$
4. $\Gamma \Vdash_\ell t : A : s$ *and* $\Gamma \Vdash_\ell t' : A' : s$ *and* $\Gamma \Vdash_\ell t \equiv t' : A : s$

then $\Gamma \Vdash_\ell \mathsf{cast}(A, B, e, t) : B : s$ *and*
$\Gamma \Vdash_\ell \mathsf{cast}(A, B, e, t) \equiv \mathsf{cast}(A', B', e', t') : B : s$.

Note that we are able to prove this lemma on reducibility, without needing validity. It is easy to derive the same lemma for validity once it is established for reducibility.

Proof. The proof is by case analysis on the reducibility proofs of A, A', B and B'. From the proofs of reducible equality, one obtains that the reducibility proofs of A and A' (resp. B and B') are introduced by the same rule. Then, if one of the normal forms is neutral, or if the normal forms of A and B have different head constructors, then $\mathsf{cast}(A, B, e, t)$ and $\mathsf{cast}(A', B', e', t')$ are neutral and easily seen to be reducible. Therefore, it suffices to prove the lemma when all the reducibility proofs are introduced by the same rule. We only cover the case of dependent products, as it is the most interesting.

We know that A reduces to a term of the form $\Pi(x : F_A).\, G_A$, as do A', B, B'. Therefore, we know that $\mathsf{cast}(A, B, e, t)$ reduces to $\lambda(a' : F_B).\, \mathsf{cast}(G_A[x := a], G_B[x := a'], \mathsf{snd}(e)\ a', t\ a)$ where a is a shorthand for $\mathsf{cast}(F_B, F_A, \mathsf{fst}(e)^{-1}, a')$, and $\mathsf{cast}(A', B', e', t')$ reduces similarly. By the weak head expansion lemma, it suffices to prove that these two normal forms are reducible, and reducibly equal at type $\Pi(x : F_B).\, G_B$—that is, we need to prove that

- applying either function to a reducible term results in a reducible term,
- applying the two functions to the same reducible term results in two reducibly equal terms,
- and that applying either function to two reducibly equal terms produces two reducibly equal terms.

To prove the first obligation, we first recursively apply the lemma to $\mathsf{cast}(F_B, F_A, \mathsf{fst}(e)^{-1}, a')$ so that we obtain reducibility of a, and then recursively apply it to $\mathsf{cast}(G_A[x := a], G_B[x := a'], \mathsf{snd}(e)\ a', t\ a)$. The other two obligations are proved in the exact same manner. □

Lemma 4.1.8 (Reducibility of the observational equality in the universe) *Given any level ℓ, in a well formed context Γ, if:*

1. $\Gamma \Vdash_\ell A : s$ *and* $\Gamma \Vdash_\ell A' : s$ *and* $\Gamma \Vdash_\ell A \equiv A' : s$
2. $\Gamma \Vdash_\ell B : s$ *and* $\Gamma \Vdash_\ell B' : s$ *and* $\Gamma \Vdash_\ell B \equiv B' : s$

then $\Gamma \Vdash_\ell A \sim_s B : \Omega$ *and* $\Gamma \Vdash_\ell A \sim_s B \equiv A' \sim_s B' : \Omega$.

Proof. The proof is by case analysis on the reducibility proofs of A, A', B and B'. As in the proof of lemma 4.1.7, we reduce the proof to introduction rules that correspond to the same kind of normal form. Most cases are straightforward, the most interesting case being again the dependent products.

We know that A reduces to a term of the form $\Pi(x : F_A).\, G_A$, and so for A', B and B'. Thus, we know that $A \sim_s B$ reduces to $(e : F_A \sim_s F_B)\ \&\ \Pi(a' : F_B).\, G_A[x := a] \sim_s G_B[x := a']$ where a is a shorthand for $\mathsf{cast}(F_B, F_A, \mathsf{fst}(e)^{-1}, a')$, and $A' \sim_s B'$ reduces similarly. By the weak head expansion lemma, it suffices to prove that the first normal form is reducible, and reducibly equal to the second one. We do this by applying reducibility of cast to get reducibility of a, and then doing recursive calls on $F_A \sim_s F_B$ and $G_A[x := a] \sim_s G_B[x := a']$. □

Lemma 4.1.9 (Reducibility of the observational equality) *Given any level ℓ, in a well formed context Γ, if:*

1. $\Gamma \Vdash_\ell A : \mathcal{U}_i$ *and* $\Gamma \Vdash_\ell A' : \mathcal{U}_i$ *and* $\Gamma \Vdash_\ell A \equiv A' : \mathcal{U}_i$
2. $\Gamma \Vdash_\ell t : A : \mathcal{U}_i$ *and* $\Gamma \Vdash_\ell t' : A' : \mathcal{U}_i$ *and* $\Gamma \Vdash_\ell t \equiv t' : A : \mathcal{U}_i$
3. $\Gamma \Vdash_\ell u : A : \mathcal{U}_i$ *and* $\Gamma \Vdash_\ell u' : A' : \mathcal{U}_i$ *and* $\Gamma \Vdash_\ell u \equiv u' : A : \mathcal{U}_i$

then $\Gamma \Vdash_\ell t \sim_A u : \Omega$ *and* $\Gamma \Vdash_\ell t \sim_A u \equiv t' \sim_A u' : \Omega$.

Proof. By case analysis on A and A'. The most difficult case is the universe, and is handled by lemma 4.1.8. The case of dependent products requires doing recursive calls on the domain and the codomain, as in the previous lemmas. □

The lemmas for cast and the observational equality constitute the bulk of the proof: they occupy approximately 5,000 lines of Agda code and most of the time required to check the whole proof is spent on them.

Lemma 4.1.10 *The rules* Π-Irr-Form, Fun-Irr *and* App-Irr *preserve validity.*

Proof. The case of the formation rule Π-Irr-Form is easy, as proving reducibility of a proof-irrelevant Π-type only requires proving that the domain and codomain are well-typed, which we obtain from the validity hypotheses.

The case of the lambda-abstraction rule Fun-Irr is also a one-liner, because proving reducibility of an inhabitant of a proof-irrelevant type is only a matter of showing that it is well-typed.

The case of the elimination rule App-Irr is more interesting: the conclusion needs us to prove that $B[x := u]$ is a valid proposition for any valid inhabitant $u : A$, and then that $t\ u$ is well-typed. Unfortunately, it seems that the premises give us very little information on B, because our definition of reducibility for impredicative dependent products is very weak (definition 4.1.4). We circumvent this problem by replacing rule App-Irr with the following typing rule, which adds well-formedness hypotheses for the domain and codomain of the Π-type.

Even though this change of rules gives the impression that the economic rules we presented in chapter 3 omit essential premises, it is in fact not the case. In Subsection 4.2.4, we will show that the economic rules contain sufficient information to recover all the necessary premises.

$$\textsc{App-Irr}'\quad \frac{\Gamma, x : A : \mathfrak{s} \vdash B : \Omega \qquad \Gamma \vdash t : \Pi(x : A).\, B : \Omega \qquad \Gamma \vdash u : A : \mathfrak{s}}{\Gamma \vdash A : \mathfrak{s}}}{\Gamma \vdash t\ u : B[x := u] : \Omega}$$

Now, our hypotheses also provide that B is valid as a type family over A. Since validity is defined as reducibility under any valid substitution, we can get the reducibility of $B[x := u]$. □

This concludes our overview of the proof of the fundamental lemma. As a direct consequence, we obtain that any well-typed term is reducible, and thus normalizing:

Corollary 4.1.11 (Normalization) *Any well-typed term has a weak-head normal form.*

4.2 Consequences of Normalization

4.2.1 Canonicity

A direct consequence of the fundamental lemma is that any closed term of type $\mathbb{N}$ reduces to a whnf of type $\mathbb{N}$. Thus, to derive canonicity for the natural numbers, we just need to know that there are no neutral terms of type $\mathbb{N}$ in an empty context.

In MLTT, neutral terms necessarily contain a variable, and therefore cannot exist in the empty context. But unfortunately, the situation is more delicate for $\mathrm{CC}^{\mathrm{obs}}$, as the eliminator $\bot - \mathsf{elim}$ can produce a neutral term from any proof of $\bot$, over which reducibility does not provide any control. Moreover, the cast operator can produce a neutral term from any proof of observational equality between two types with incompatible head constructors, which we do not control either.

Therefore, in order to prove canonicity for natural numbers (or for any other type), we need to show that there exists no proof of $\bot$ in the empty context, and no proof of equality between incompatible types. We will prove this by constructing a set-theoretic model of $\mathrm{CC}^{\mathrm{obs}}$ in section 4.4. As a consequence, we are able to derive our canonicity theorem:

Theorem 4.2.1 (Computational canonicity of $\mathbb{N}$) *If a term t has type $\mathbb{N}$ in the empty context, then there exists an external integer n such that t reduces to $\mathsf{S}^n\ 0$.*

This strategy of deriving canonicity from consistency and normalization was already envisioned by Altenkirch *et al.* [31].

[31]: Altenkirch et al. (2007), "Observational Equality, Now!"

4.2.2 Decidability of Conversion

In order to prove that convertibity is decidable for $\mathrm{CC}^{\mathrm{obs}}$, we need to devise a conversion checking algorithm, and then prove that it is both correct and complete.

To do so, we follow Abel et al. [30] and define *algorithmic conversion* as a relation $\Gamma \vdash t \cong u : A : \mathfrak{s}$ between two well-typed terms t and u of type A in context Γ. This algorithmic conversion keeps track of the relevance information, and uses it to either give an immediate answer in case t and u are irrelevant, or to compute their weak-head normal forms using the normalization theorem, then compare their head constructor, and possibly apply the algorithm recursively in case t and u are relevant. Correctness of this algorithmic conversion is direct as the rules used are subsumed by the general conversion judgement *[Soundness.agda]*.

[30]: Abel et al. (2018), "Decidability of Conversion for Type Theory in Type Theory"

Showing that the algorithmic conversion is also complete is more involved. The main ingredient is more or less a re-enactment of our proof of the fundamental lemma, but with a definition of reducibility that uses algorithmic conversion instead of the definitional equality of $\mathrm{CC}^{\mathrm{obs}}$. *[Completeness.agda]*. In our formal proof, we follow Abel et al. [30] in factoring the two instances of the fundamental lemma by defininig a generic interface for both algorithmic conversion and typed conversion, and using this interface in the definition of the logical relation *[EqualityRelation.agda]*.

[30]: Abel et al. (2018), "Decidability of Conversion for Type Theory in Type Theory"

Theorem 4.2.2 (Equivalence of conversion and algorithmic conversion) *Given two terms t and u such that $\Gamma \vdash t : A : \mathfrak{s}$ and $\Gamma \vdash u : A : \mathfrak{s}$, we have that*

$$\Gamma \vdash t \equiv u : A : \mathfrak{s} \iff \Gamma \vdash t \cong u : A : \mathscr{U}.$$

It still remains to provide a decision procedure for the algorithmic conversion *[Decidable.agda]*. Given two terms t and u well-typed at $A : \mathfrak{s}$ in context Γ, we can apply the fundamental lemma plus the reflexivity of the logical relation to get the fact that $\Gamma \Vdash_\ell t \equiv t : A : \mathfrak{s}$ and similarly for u. Because reducibly equal terms are also algorithmically convertible by the escape lemma, we have $\Gamma \vdash t \cong t : A : \mathfrak{s}$ (and similarly for u). Then the decision procedure is done by double induction on the proofs that t and u are reflexive for the algorithmic conversion. The idea is that t and u are convertible if and only if the two reflexive proofs are the same.

Note that the proof that t is algorithmically equal to itself contains the fact that t strongly normalizes, because t can be recursively put in whnf.

4.2.3 Decidability of Typing

Now that we have an algorithm that decides conversion, we can rely on the work of Lennon-Bertrand [47] on bidirectional type-checking, who explains how to obtain an algorithm for type-checking provided that the theory enjoys subject reduction and decidability of conversion.

[47]: Lennon-Bertrand (2022), "Bidirectional Typing in the Calculus of Inductive Constructions"

4.2.4 Inferring Premises from Economic Typing Rules

Recall that in our construction of the normalization model, impredicativity forced us to give a definition of reducibility for proof-irrelevant Π-types that is considerably weaker than the definition for proof-relevant

Π-types. As a consequence, we were not able to prove the fundamental lemma directly on the economic inference rules of chapter 3, and we circumvented this issue by adding well-formedness hypotheses to several rules. For instance, the rule for application of a proof-irrelevant function

$$\textsc{App-Irr}\quad \frac{\Gamma \vdash t : \Pi(x : A).\, B : \Omega \qquad \Gamma \vdash u : A : \mathfrak{s}}{\Gamma \vdash t\, u : B[x := u] : \Omega}$$

was modified to include hypotheses on the domain and the codomain of the function.

$$\textsc{App-Irr}'\quad \frac{\Gamma, x : A : \mathfrak{s} \vdash B : \Omega \qquad \Gamma \vdash t : \Pi(x : A).\, B : \Omega \qquad \Gamma \vdash u : A : \mathfrak{s}}{\Gamma \vdash t\, u : B[x := u] : \Omega}$$

Therefore, the reader may wonder if the economic rules are only approximations which need to be extended with the omitted well-formedness premises for types, contexts, etc. In this section, we show that it is not the case, and that the economic rules are in fact equivalent to the full rules.

Until the next section, we distinguish between the *economic* version of the system which uses the rules defined in chapter 3, that we denote $\vdash_e$, and the *paranoid* version which adds well-formedness hypotheses to all types that appear in rules, noted $\vdash_p$.

The terminology is in reference to Bauer *et al.* [48].

Obviously, when a term is well-typed for the paranoid variant, it is also well-typed in the economic system. But after proving the fundamental lemma on the paranoid variant, it becomes possible to show that the two typing systems are equivalent, leveraging the inversion lemmas provided by reducibility. Thus, the additional premises of the paranoid variant help us prove the fundamental lemma in presence of impredicativity, but end up being redundant once a sufficient amount of metatheory is established.

It must be noted that since the proof of equivalence of the two systems uses the fundamental lemma, it is by no means computationally trivial. Therefore, in an actual implementation of a typechecker for $\mathrm{CC}^{\mathrm{obs}}$, it might be more sensible to work with the paranoid system if we need the supplementary information.

First, we establish the following result on the paranoid typing, by combining the fundamental lemma with the fact that reducible types are well-formed and reducible terms are well-typed:

Corollary 4.2.3 (Typing Validity *[Syntactic.agda]*)

1. *If* $\Gamma \vdash_p t : A : \mathfrak{s}$ *then* $\Gamma \vdash_p A : \mathfrak{s}$
2. *If* $\Gamma \vdash_p t \equiv u : A : \mathfrak{s}$ *then* $\Gamma \vdash_p A : \mathfrak{s}$, $\Gamma \vdash_p t : A : \mathfrak{s}$ *and* $\Gamma \vdash_p u : A : \mathfrak{s}$.

And now that we are equipped with this corollary, as well as some inversion lemmas that we obtained from the reducibility, we know enough to build a paranoid typing derivation from a standard derivation:

Theorem 4.2.4 (*[NonParanoidTyping.agda]*) *If* $\Gamma \vdash_e t : A : \mathfrak{s}$ *then* $\Gamma \vdash_p t : A : \mathfrak{s}$.

Proof. The proof is by induction on the typing derivation. The correspondence between the two systems is one-to-one except for the additional paranoid assumptions. In all cases, we can get them from corollary 4.2.3 and from inversion lemmas, such as the fact that A and B are well-typed whenever $\Pi(x : A).\, B$ is. □

Thus, we also have the fundamental lemma for the economic type system:

Corollary 4.2.5 (Fundamental lemma on the economic type system) *If* $\Gamma \vdash_e t : A : \mathbb{s}$*, then there is a level* ℓ *such that* $\Vdash^v \Gamma$*,* $\Gamma \Vdash^v_\ell A : \mathbb{s}$ *and* $\Gamma \Vdash^v_\ell t : A : \mathbb{s}$.

4.3 Analysis of the Normalization Proof

Both the informal normalization proof presented in section 4.1 and its formalized version in Agda take place in a somewhat sophisticated metatheory: Martin-Löf Type Theory extended with induction-recursion. In this section, we explain how to re-enact the proof in a weaker meta-theory in order to control the computational complexity of the proof terms more finely.

4.3.1 The Computational Expressivity of CC$^{\text{obs}}$

Proof-relevant terms in CC$^{\text{obs}}$ are *programs*, at least in the sense of being terms in a lambda-calculus with additional operators. As such, they may be extracted and evaluated just like regular programs— and in fact, evaluating open terms is pretty much what our conversion checking algorithm from Subsection 4.2.3 does. Therefore, it makes sense to investigate the computational complexity of these proof terms: which integer functions can be expressed as proof terms?

We restrict our attention to functions from $\mathbb{N}$ to $\mathbb{N}$, because higher-order computability is a complex topic with many non-equivalent definitions.

As an aside, we must mention that these considerations are not a useful measure of the feasability of type-checking. Indeed, most proof assistants based on type theory have ridiculously high worst-case complexity, but they are still very efficient in practice! However, investigating the expressive power of proof terms is still a worthwile endeavor, as it provides us with information on the kinds of mathematical statements that can be constructively proved in the relevant fragment of CC$^{\text{obs}}$: for instance, is it possible to define a normalization function for System F terms?

To derive a lower bound for the computational expressivity, we simply remark that Martin-Löf type theory is a subset of CC$^{\text{obs}}$, provided we use a definition of MLTT that does not support more inductive types than our definition of CC$^{\text{obs}}$. Therefore, any function that can be defined in MLTT is definable in CC$^{\text{obs}}$.

In order to obtain an upper bound, we need to analyze the normalization proof: since the argument is constructive, it provides us with a concrete algorithm to evaluate terms of CC$^{\text{obs}}$. Therefore, we seek to replay our normalization proof in a meta-theory that is as weak as

possible, to get a tighter upper bound. Of course, we cannot hope to go lower than MLTT, given that it is a subset $\mathsf{CC}^{\mathsf{obs}}$. But we can get pretty close:

Theorem 4.3.1 MLTT *with $n + 4$ universes and a general scheme for inductive types can prove normalization of* $\mathsf{CC}^{\mathsf{obs}}$ *with n universes.*

Proof. The proof of this theorem is given in Subsection 4.3.2. □

In the rest of this section, we will write $\mathsf{MLTT}^{\mathsf{ind}}_n$ for Martin-Löf type theory with n universes and a general scheme of inductive types, and we will write $\mathsf{CC}^{\mathsf{obs}}_n$ for the observational calculus of constructions with n universes.

When we let n go to infinity, or in other words when we consider theories with an unbounded hierarchy of universes, we get that $\mathsf{MLTT}^{\mathsf{ind}}$ is sufficient to prove normalization for $\mathsf{CC}^{\mathsf{obs}}$.

Corollary 4.3.2 MLTT *with a general scheme of inductive types can prove normalization of* $\mathsf{CC}^{\mathsf{obs}}$.

Note that we cannot hope for a similar result in terms of consistency or canonicity: while consistency and canonicity for MLTT follow from normalization, proving that $\mathsf{CC}^{\mathsf{obs}}$ is consistent requires an impredicative theory, which is necessarily much stronger than $\mathsf{MLTT}^{\mathsf{ind}}$.

Theorem 4.3.1 provides us with an upper bound on the computational power of $\mathsf{CC}^{\mathsf{obs}}$:

Corollary 4.3.3 *Any integer function that can be defined as a closed term of type $\mathbb{N} \to \mathbb{N}$ in* $\mathsf{CC}^{\mathsf{obs}}$ *can also be defined in* MLTT *with a general scheme for inductive types.*

Proof. We start by noting that any closed term f of type $\mathbb{N} \to \mathbb{N}$ in $\mathsf{CC}^{\mathsf{obs}}$ only mentions a finite number of universes N. Thus, theorem 4.3.1 provides us with a normalization function for $\mathsf{CC}^{\mathsf{obs}}_N$ in $\mathsf{MLTT}^{\mathsf{ind}}_{N+4}$, that computes the normal form of any well-typed term. From there, we can define a function $f' : \mathbb{N} \to \mathbb{N}$ in MLTT that represents the same integer function as f. Given an integer $n : \mathbb{N}$, f' first computes a $\mathsf{CC}^{\mathsf{obs}}$ typing derivation for $f\,n$ in the empty context, then applies the normalization function to obtain a weak-head normal form that is necessarily canonical, and converts this normal form back to an integer. □

Corollary 4.3.3 might come off as a surprise, since $\mathsf{CC}^{\mathsf{obs}}$ is equipped with an impredicative universe, and impredicativity generally adds a great deal of proof-theoretic strength! And $\mathsf{CC}^{\mathsf{obs}}$ does possess this logical power, in fact. Indeed, it can represent many more functions as Ω-valued functional relations than $\mathsf{MLTT}^{\mathsf{ind}}$. But this corollary shows that there is no way to extract them in the proof-relevant fragment, even though we have access to elimination principles for false propositions and the observational equality. Thus, the logical power of impredicativity is locked inside of Ω.

This is in stark contrast with the Calculus of Inductive Constructions, in which it is possible to define closed terms of type $\mathbb{N} \to \mathbb{N}$ that leverage the power of impredicativity, using a principle called large elimination of singleton inductive types. We discuss this principle with more detail in chapter 5.

The standard technique is to use the accessibility predicate as defined in https://coq.inria.fr/library/Coq.Init.Wf.html.

4.3.2 Fitting the Normalization Proof in $\mathsf{MLTT}^{\mathrm{ind}}$

In this section, we give a proof of theorem 4.3.1, by encoding the logical relation sketched in section 4.1 without induction-recursion. This argument has been formalized in Agda. For the remainder of this section, we work in $\mathsf{MLTT}^{\mathrm{ind}}_{n+4}$ with a deep embedding of $\mathrm{CC}^{\mathrm{obs}}_n$. To avoid confusion between the meta-theory and the object theory, we will use Agda-style notations for the meta-theory.

The formalization is available at https://github.com/loic-p/logrel-mltt/tree/without-IR. It is done for MLTT and not full $\mathrm{CC}^{\mathrm{obs}}$, but it does not make a difference as the crux of the proof lies in the definition of the normalization model without induction-recursion.

From a bird's eye perspective, the normalization proof builds a model of $\mathrm{CC}^{\mathrm{obs}}$ in which well-formed types are interpreted as proof-relevant predicates on untyped terms, and induction-recursion is used to construct a universe in the model: we define the inductive predicate of reducible types simultaneously with recursive functions that associates reducibility predicates to a reducible type. On closer inspection, we realize that the proof still works if the reducibility predicate of a universe lives one universe higher than the reducibility predicates of the types it contains. This allows us to use *small* induction-recursion, which can be replaced by functional relations that only require simple indexed inductive types [49].

[49]: Hancock et al. (2013), "Small Induction Recursion"

Figure 4.4 showcases what the relation-based definition looks like. For all $\ell \leq n$, we define an inductive relation R^ℓ between a context, an untyped term t (the reducible type), its sort, a predicate $P_=$ of types that are convertible to t, a predicate P_t which is the reducibility predicate associated to t, and a binary relation $P_{t=}$ that encodes convertibility of terms in P_t. The reducibility for types $\Vdash_\ell$ is then defined in terms of R^ℓ. The inductive relation features twelve constructors that are all described in section 4.1, except for the W-types (which do not pose any additional difficulty).

Note that the logical relation is built in stages:

- We first define an inductive relation R^0 that has cases for dependent products and all base types, except for proof-relevant universes: it only accounts for inhabitants of $\mathscr{U}_0$.
- From R^0, we define a reducibility predicate $P_{\mathscr{U}_0}$ for $\mathscr{U}_0$: a term t is in $P_{\mathscr{U}_0}$ if there exist $P_=, P_t, P_{t=}$ such that $R^0(t, P_=, P_t, P_{t=})$. Now that we have a predicate for $\mathscr{U}_0$, we can define a relation R^1 that accounts for inhabitants of $\mathscr{U}_1$. Of course, since $P_{\mathscr{U}_0}$ lives in Set_1, R^1 will land in Set_2.
- We repeat this process n times to obtain a relation R^n that handles our n universes.

Each step of this process requires an additional meta-theoretical universe level. This is not too surprising, since we are proving normalization for a theory that subsumes MLTT_n, a property from which we can deduce the consistency of MLTT_n. Therefore, if this proof scheme can be extended to deal with general inductive types in the object theory,

$$
\begin{array}{lcl}
\mathrm{R}^{\ell} & : & \mathrm{Context} \to \mathrm{Term} \to (\mathrm{Term} \to \mathrm{Set}_{\ell}) \to (\mathrm{Term} \to \mathrm{Set}_{\ell}) \to (\mathrm{Term} \to \mathrm{Term} \to \mathrm{Set}_{\ell}) \to \mathrm{Set}_{\ell+1} \\
\mathrm{R}^{\ell} & ::= & \mathrm{R}_{\mathrm{ne}} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\mathrm{ne}} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\mathrm{ne}} t \equiv \_)\ (\Gamma \Vdash_{\mathrm{ne}} \_ : t)\ (\Gamma \Vdash_{\mathrm{ne}} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\Pi\mathrm{i}} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\Pi\mathrm{i}} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\Pi\mathrm{i}} t \equiv \_)\ (\Gamma \Vdash_{\Pi\mathrm{i}} \_ : t)\ (\Gamma \Vdash_{\Pi\mathrm{i}} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\bot} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\bot} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\bot} t \equiv \_)\ (\Gamma \Vdash_{\bot} \_ : t)\ (\Gamma \Vdash_{\bot} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\mathrm{U}} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\mathrm{U}} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\mathrm{U}} t \equiv \_)\ (\Gamma \Vdash_{\mathrm{U}} \_ : t)\ (\Gamma \Vdash_{\mathrm{U}} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\Omega} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\Omega} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\Omega} t \equiv \_)\ (\Gamma \Vdash_{\Omega} \_ : t)\ (\Gamma \Vdash_{\Omega} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\mathbb{N}} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\mathbb{N}} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\mathbb{N}} t \equiv \_)\ (\Gamma \Vdash_{\mathbb{N}} \_ : t)\ (\Gamma \Vdash_{\mathbb{N}} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\Pi} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\Sigma} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\Pi} t \equiv \_)\ (\Gamma \Vdash_{\Pi} \_ : t)\ (\Gamma \Vdash_{\Pi} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\Sigma} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\Sigma} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\Sigma} t \equiv \_)\ (\Gamma \Vdash_{\Sigma} \_ : t)\ (\Gamma \Vdash_{\Sigma} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\Box} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\Box} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\Box} t \equiv \_)\ (\Gamma \Vdash_{\Box} \_ : t)\ (\Gamma \Vdash_{\Box} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\mathrm{Q}} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\mathrm{Q}} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\mathrm{Q}} t \equiv \_)\ (\Gamma \Vdash_{\mathrm{Q}} \_ : t)\ (\Gamma \Vdash_{\mathrm{Q}} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\mathrm{Id}} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\mathrm{Id}} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\mathrm{Id}} t \equiv \_)\ (\Gamma \Vdash_{\mathrm{Id}} \_ : t)\ (\Gamma \Vdash_{\mathrm{Id}} \_ \equiv \_ : t) \\
& | & \mathrm{R}_{\mathrm{W}} : \forall\ \Gamma\ t \to (\Gamma \Vdash_{\mathrm{Id}} t) \to \mathrm{R}^{\ell}\ \Gamma\ t\ (\Gamma \Vdash_{\mathrm{W}} t \equiv \_)\ (\Gamma \Vdash_{\mathrm{W}} \_ : t)\ (\Gamma \Vdash_{\mathrm{W}} \_ \equiv \_ : t)
\end{array}
$$

All the auxiliary predicates whose name mentions $\Vdash_{\mathrm{ne}}$, $\Vdash_{\Pi\mathrm{i}}$, $\Vdash_{\bot}$, $\Vdash_{\mathrm{U}}$ ... are defined exactly as in section 4.1.

$$
\begin{array}{lcl}
\_ \Vdash_{\ell} \_ : \_ & : & (\Gamma : \mathrm{Context}) \to (t : \mathrm{Term}) \to (A : \mathrm{Term}) \to (s : \mathrm{Sort}) \to \mathrm{Set}_{\ell+1} \\
\Gamma \Vdash_{\ell} t : s & = & (P_{=} : \mathrm{Term} \to \mathrm{Set}_{\ell}) \times (P_{t} : \mathrm{Term} \to \mathrm{Set}_{\ell}) \\
& & \times\ (P_{t=} : \mathrm{Term} \to \mathrm{Term} \to \mathrm{Set}_{\ell}) \times (\mathrm{R}^{\ell}\ \Gamma\ t\ s\ P_{=}\ P_{t}\ P_{t=})
\end{array}
$$

$$
\begin{array}{lcl}
\_ \Vdash_{\ell} \_ \equiv \_ : \_ & : & (\Gamma : \mathrm{Context}) \to (A : \mathrm{Term}) \to (B : \mathrm{Term}) \to (s : \mathrm{Sort}) \to \{\Gamma \Vdash_{\ell} A : s\} \to \mathrm{Set}_{\ell} \\
\Gamma \Vdash_{\ell} A \equiv B : s & = & P_{=}\ B
\end{array}
$$

$$
\begin{array}{lcl}
\_ \Vdash_{\ell} \_ : \_ : \_ & : & (\Gamma : \mathrm{Context}) \to (t : \mathrm{Term}) \to (A : \mathrm{Term}) \to (s : \mathrm{Sort}) \to \{\Gamma \Vdash_{\ell} A : s\} \to \mathrm{Set}_{\ell} \\
\Gamma \Vdash_{\ell} t : A : s & = & P_{t}\ t
\end{array}
$$

$$
\begin{array}{lcl}
\_ \Vdash_{\ell} \_ \equiv \_ : \_ : \_ & : & (\Gamma : \mathrm{Context}) \to (t\ u : \mathrm{Term}) \to (A : \mathrm{Term}) \to (s : \mathrm{Sort}) \to \{\Gamma \Vdash_{\ell} A : s\} \to \mathrm{Set}_{\ell} \\
\Gamma \Vdash_{\ell} t \equiv u : A : s & = & P_{t=}\ t\ u
\end{array}
$$

Figure 4.4: Inductive encoding of reducibility

then Gödel's incompleteness theorem applies and guarantees that we need more universes in the meta-theory than in the object theory.

Constructing this universe of reducible types is only the first half of the normalization proof. To complete the construction, we also need to encode validity without induction-recursion in $\mathrm{MLTT}^{\mathrm{ind}}_{n+4}$ and prove the fundamental lemma. We do not develop this here as the construction is a similar encoding of small induction-recursion with functional relations. The interested reader can consult the formalization at `https://github.com/loic-p/logrel-mltt/tree/without-IR`.

4.4 Semantics of $\mathrm{CC}^{\mathrm{obs}}$

In this section, we turn to the construction of the "standard" set-theoretic model of $\mathrm{CC}^{\mathrm{obs}}$. This model serves two purposes: to show consistency of our theory on the one hand, and to ensure that $\mathrm{CC}^{\mathrm{obs}}$ is a good internal language for set theory on the other hand.

4.4.1 Deriving Consistency from a Model

As we already mentioned, the normalization model that we built in section 4.1 is not strong enough to obtain consistency and canonicity for $\mathrm{CC}^{\mathrm{obs}}$. Indeed, this model interprets the proof-irrelevant proposition $\bot$ as the set of syntactic terms with type $\bot$, which does not grant any kind of control over its proofs in the empty context.

This was already the case in our previous work on $\mathsf{TT}^{\mathrm{obs}}$ [35], so we supplemented our normalization proof with a model in the category of sets (presented as setoids), using induction-recursion to interpret universes as sets of codes *à la Tarski* defined mutually with eliminators and coercion functions. In that model, $\bot$ is actually interpreted as the empty set, which proves consistency of $\mathsf{TT}^{\mathrm{obs}}$: we can use it to show that there is no term of type $\bot$ in the empty context.

[35]: Pujet et al. (2022), "Observational Equality: Now For Good"

4.4.2 $\mathrm{CC}^{\mathrm{obs}}$ as an Internal Language for Sets

While the construction from our earlier work [35] does show consistency of $\mathsf{TT}^{\mathrm{obs}}$, it arguably falls short of presenting $\mathsf{TT}^{\mathrm{obs}}$ as an internal language for classical mathematics. Indeed, the inductive-recursive construction of the universe of codes is designed to only account for the codes that come from the syntax of $\mathsf{TT}^{\mathrm{obs}}$. In other words, the following statement

$$\Pi(A : \mathscr{U})(x\ y\ z : A).\ x \sim_A y \to y \sim_A z \to x \sim_A z$$

only states transitivity of equality for sets that are built from the type formers of $\mathsf{TT}^{\mathrm{obs}}$, when interpreted in their model.

This state of affairs is obviously not ideal, as readers who are not willing to accept $\mathsf{TT}^{\mathrm{obs}}$/ $\mathrm{CC}^{\mathrm{obs}}$ as a new foundation of mathematics would probably be more inclined to use it if they knew that any theorem they proved in type theory is also true for classical set theory. Thus, instead of replicating this construction in an impredicative meta-theory (to be able to interpret Ω), we seize the opportunity to solve two problems at once, and present a model in classical set theory that gives a much more sensible meaning to statements in $\mathrm{CC}^{\mathrm{obs}}$ while still being able to show consistency.

4.4.3 Constructing the Set-theoretic Model

We work in ZF set theory with a countable hierarchy of Grothendieck universes $\mathbf{V}_0, \mathbf{V}_1, \ldots$ We call $\mathbb{B}$ the lattice of truth values (or in other words, the subobjects of the singleton set $\{*\}$), with $\emptyset$ as its minimal value. Even though we stay in the world of set theory for the duration of this section, we remain type theorists, and as such we use dependent products, dependent sums and inductive types in this section. They should be understood as their standard interpretation in the set-theoretic model of type theory. In an attempt to minimize confusion, we change our notations a little bit: we use $(a \in A) \to (B\ a)$ for the set-theoretic dependent product, $(a \in A) \times (B\ a)$ for the the set-theoretic dependent sum and N for the set-theoretic integers.

The central ingredient of our standard model is the interpretation of the proof-relevant universes of $\mathrm{CC}^{\mathrm{obs}}$. It is tempting to define them directly as Grothendieck universes, but unfortunately this does not work: in $\mathrm{CC}^{\mathrm{obs}}$, type constructors are injective with respect to the observational equality (for instance, $(A \to B) \sim (A' \to B')$ implies that $A \sim A'$ and $B \sim B'$) while set-theoretic function spaces collapse too much information for this to be possible ($\bot \to \mathsf{Bool}$ and $\bot \to \mathbb{N}$ are identical function sets).

This is not too difficult to fix, however: we can define a hierarchy of sets $\mathbf{U}_0, \mathbf{U}_1 \ldots$ that have the same elements as the Grothendieck universes, but decorated with labels that keep track of how they were constructed. For instance, the label of a set that has been constructed as a function space will contain the labels of its domain and codomain, so that there is no way to confuse two different function spaces that happen to have the same elements.

The most natural way to do this from a type theorist's perspective is to build an inductive predicate $\mathbf{U}_i^\varepsilon$ over $\mathbf{V}_i$ and then define our alternative universes as $\mathbf{U}_i := (X \in \mathbf{V}_i) \times (\mathbf{U}_i^\varepsilon\ X)$, as described in figure 4.5. Note how the constructor $\mathsf{c}_{\mathsf{emb}}$ builds a proof of $\mathbf{U}_i^\varepsilon\ X$ for any (small) set X, so that $\mathrm{CC}^{\mathrm{obs}}$ statements that quantify over the universe apply to all appropriately-sized sets of the model. This also implies that there might be several codes for the same set: for instance $(\mathbb{N} \to \mathbb{N}\ ;\ \mathsf{c}_{\mathsf{emb}})$ and $(\mathbb{N} \to \mathbb{N}\ ;\ \mathsf{c}_{\Pi\mathsf{UU}}\ \ldots)$ are both codes for the set $\mathbb{N} \to \mathbb{N}$, but only the second one remembers that it has been built as a function space.

Remark that the natural equality between elements of $\mathbf{U}_i$ corresponds closely the equality of $\mathrm{CC}^{\mathrm{obs}}$: two elements can only be equal if they pack the same witness of $\mathbf{U}_i^\varepsilon$, which means that they have been built in the same way. Moreover, if we compare two codes of the form $(X \to Y;\ \mathsf{c}_{\Pi\mathsf{UU}}\ \ldots)$, the equality of the second member means that the domains and the codomains of the function spaces are equal—we recover the injectivity of the type formers.

4.4.4 Interpreting the Syntax of $\mathrm{CC}^{\mathrm{obs}}$

Now that we know how to deal with the universes, it only remains to spell out the interpretation of the full syntax of $\mathrm{CC}^{\mathrm{obs}}$. In the standard fashion [10], we define interpretation functions that send

- a context Γ to a set $[\![\Gamma]\!]$, and
- a pair of a term t and a context Γ to a set $[\![t]\!]_\Gamma$ indexed over $[\![\Gamma]\!]$.

[10]: Hofmann (1995), "Extensional concepts in intensional type theory"

Since these functions are defined on raw syntax without any guarantee of well-typedness, we cannot expect every term to have a sensible interpretation so we need to define them as partial functions. We will be able to prove that all well-typed terms admit an interpretation *a posteriori* by induction on the derivations.

As usual, contexts are interpreted as telescopes: $[\![\bullet]\!] = \{*\}$ and $[\![\Gamma, x : A]\!] = (\rho : [\![\Gamma]\!]) \times ([\![A]\!]_\Gamma\ \rho)$ when $[\![A]\!]_\Gamma$ is defined. Given $\rho \in [\![\Gamma]\!]$ and a variable x that appears in Γ, we write $\rho(x)$ for the corresponding projection.

The interpretation of the terms is defined in figure 4.6.

$$
\begin{array}{lcl}
\mathbf{U}_i^\varepsilon & : & \mathbf{V}_i \to \mathbf{V}_{i+1} \\
\mathbf{U}_i^\varepsilon & ::= & \mathsf{c}_{\mathsf{emb}} : (X \in \mathbf{V}_i) \to \mathbf{U}_i^\varepsilon\ X \\
& | & \mathsf{c}_{\Pi UU} : \{j,k \in \mathsf{N} \mid \max(j,k) \le i\} \to (A \in \mathbf{V}_j) \to (A_\varepsilon \in \mathbf{U}_i^\varepsilon\ A) \\
& & \quad \to (B \in (A \to \mathbf{V}_k)) \to (B_\varepsilon \in ((a \in A) \to \mathbf{U}_i^\varepsilon\ (B\ a))) \\
& & \quad \to \mathbf{U}_i^\varepsilon\ ((a \in A) \to B\ a) \\
& | & \mathsf{c}_{\Pi \Omega U} : (A \in \mathbb{B}) \\
& & \quad \to (B \in (A \to \mathbf{V}_i)) \to (B_\varepsilon \in ((a \in A) \to \mathbf{U}_i^\varepsilon\ (B\ a))) \\
& & \quad \to \mathbf{U}_i^\varepsilon\ ((a \in A) \to B\ a) \\
& | & \mathsf{c}_{\Sigma} : \{j,k \in \mathsf{N} \mid \max(j,k) \le i\} \to (A \in \mathbf{V}_j) \to (A_\varepsilon \in \mathbf{U}_i^\varepsilon\ A) \\
& & \quad \to (B \in (A \to \mathbf{V}_k)) \to (B_\varepsilon \in ((a \in A) \to \mathbf{U}_i^\varepsilon\ (B\ a))) \\
& & \quad \to \mathbf{U}_i^\varepsilon\ ((a \in A) \times B\ a) \\
& | & \mathsf{c}_{\square} : (A \in \mathbb{B}) \to \mathbf{U}_i^\varepsilon\ A \\
& | & \mathsf{c}_{\mathsf{Q}} : \{j \in \mathsf{N} \mid j < i\} \to (A \in \mathbf{V}_j) \to (A_\varepsilon \in \mathbf{U}_i^\varepsilon\ A) \\
& & \quad \to \{R \in A \to A \to \mathbb{B} \mid R \text{ is an equivalence relation}\} \\
& & \quad \to \mathbf{U}_i^\varepsilon\ (A/R) \\
& | & \mathsf{c}_{\mathsf{Id}} : \{j \in \mathsf{N} \mid j < i\} \to (A \in \mathbf{V}_j) \to (A_\varepsilon \in \mathbf{U}_i^\varepsilon\ A) \\
& & \quad \to (x \in A) \to (y \in A) \\
& & \quad \to \mathbf{U}_i^\varepsilon\ \{x \in \{*\} \mid t = u\} \\
& | & \mathsf{c}_{\mathsf{U}} : \{j \in \mathsf{N} \mid j < i\} \to \mathbf{U}_i^\varepsilon\ \mathbf{U}_j \\
& | & \mathsf{c}_{\Omega} : \mathbf{U}_i^\varepsilon\ \mathbb{B}
\end{array}
$$

$$
\mathbf{U}_i \quad := \quad (X \in \mathbf{V}_i) \times (\mathbf{U}_i^\varepsilon\ X)
$$

$$
\begin{array}{l}
\mathsf{el} \quad : \quad \mathbf{U}_i \to \mathbf{V}_i \\
\mathsf{el}\ (X\ ;\ X_\varepsilon) \ := \ X
\end{array}
$$

Figure 4.5: The universe of codes, defined with set-theoretic inductive types

- In the proof-relevant fragment, we interpret types as elements of our decorated hierarchy $\mathbf{U}_1, \mathbf{U}_2 \ldots$ and terms as sets. The intention is that if t is an inhabitant of the proof relevant term A in context Γ, then its interpretation $[\![t]\!]_\Gamma\ \rho$ should be an element of $\mathsf{el}\ ([\![A]\!]_\Gamma\ \rho)$ for all $\rho \in [\![\Gamma]\!]$.
- In order to validate proposition extensionality, proof-irrelevant propositions are directly interpreted as *truth values*, or in other words, subobjects of the singleton set. Computationally irrelevant proofs such as refl of $\square$–elim are all interpreted as $\rho \mapsto *$, which guarantees that the corresponding proposition is true for all $\rho \in [\![\Gamma]\!]$. We can interpret cast as the identity, since the two types are interpreted as the same set, and $\bot$–elim(A,t) is not even interpreted since it can only be formed in inconsistent contexts, which are empty in the model.

In order to prove the soundness of our interpretation, we need to extend it to weakenings and substitutions between contexts. Assume Γ and Δ are a syntactic contexts, and A and t are syntactic terms. In case $[\![\Gamma, x : A, \Delta]\!]$ and $[\![\Gamma, \Delta]\!]$ are well-defined, let π_A be the projection:

$$
\begin{array}{c}
\pi_A : [\![\Gamma, x : A, \Delta]\!] \to [\![\Gamma, \Delta]\!] \\
(\vec{x_\Gamma}, x_A, \vec{x_\Delta}) \mapsto (\vec{x_\Gamma}, \vec{x_\Delta}).
\end{array}
$$

In case $[\![\Gamma, \Delta[x := t]]\!]$ and $[\![\Gamma, x : A, \Delta]\!]$ are well-defined, we define the function σ_t by:

$$
\begin{array}{c}
\sigma_t : [\![\Gamma, \Delta[x := t]]\!] \to [\![\Gamma, x : A, \Delta]\!] \\
(\vec{x_\Gamma}, \vec{x_\Delta}) \mapsto (\vec{x_\Gamma}, [\![t]\!]_\Gamma\ \vec{x_\Gamma}, \vec{x_\Delta}).
\end{array}
$$

Lemma 4.4.1 (Weakening) *π_A is the semantic counterpart to the weakening of A: for all terms u, when both sides are well defined, we have:*

$$[\![u]\!]_{\Gamma,x:A,\Delta} = [\![u]\!]_{\Gamma,\Delta} \circ \pi_A$$

Lemma 4.4.2 (Substitution) *σ_t is the semantic counterpart to the substitution by t: for all terms u, when both sides are well defined, we have:*

$$[\![u[x := t]]\!]_{\Gamma,\Delta[x:=t]} = [\![u]\!]_{\Gamma,x:A,\Delta} \circ \sigma_t$$

Theorem 4.4.3 (Soundness of the Standard Model)

1. *If $\vdash \Gamma$ then $[\![\Gamma]\!]$ is defined.*
2. *If $\Gamma \vdash A : \Omega$ then $[\![A]\!]_\Gamma\ \rho$ is a subset of $\{*\}$ for all $\rho \in [\![\Gamma]\!]$.*
3. *If $\Gamma \vdash A : \mathscr{U}_i$ then $[\![A]\!]_\Gamma\ \rho$ is in $\mathbf{U}_i$ for all $\rho \in [\![\Gamma]\!]$.*
4. *If $\Gamma \vdash t : A : \Omega$ then $[\![A]\!]_\Gamma\ \rho$ is equal to $\{*\}$ for all $\rho \in [\![\Gamma]\!]$.*
5. *If $\Gamma \vdash t : A : \mathscr{U}_i$ then $[\![t]\!]_\Gamma\ \rho$ is in $\mathsf{el}\ ([\![A]\!]_\Gamma\ \rho)$ for all $\rho \in [\![\Gamma]\!]$.*
6. *If $\Gamma \vdash t \equiv u : A : \mathfrak{s}$ then $[\![t]\!]_\Gamma = [\![u]\!]_\Gamma$.*

Proof. By induction on the typing derivations, using lemmas 4.4.1 and 4.4.2. □

4.4.5 Consequences of the Model

From the soundness theorem and the interpretation of $\bot$ being empty, the consistency of $\mathrm{CC}^{\mathrm{obs}}$ is immediate:

Theorem 4.4.4 (Consistency) *There are no proofs of $\bot$ in the empty context.*

Furthermore, the model allows us to show that there are no neutral terms in the empty context: if we inspect the normal forms provided by the normalization theorem, we realize that a cast between two incompatible types is the only way to build a neutral term in the empty context. But giving a type to such a cast requires having an equality proof between two incompatible types, which contradicts consistency.

Theorem 4.4.5 *There are no neutral terms in the empty context.*

From there, we deduce the canonicity theorem for the integers: all integers reduce to standard integers in the empty context. Thus, our modified model allows us to establish all of the meta-theoretical properties that we claimed in chapter 3.

On the other hand, it seems difficult to measure to what extent our set-theoretic model presents $\mathrm{CC}^{\mathrm{obs}}$ as a good internal language for sets. Compared to the universe construction of Pujet et al. [35], we gain the property that every set belongs to a universe—meaning that theorems which quantify over $\mathscr{U}_i$ will apply to all sufficiently small sets. Of course, this does not prevent us from proving some theorems that are

[35]: Pujet et al. (2022), "Observational Equality: Now For Good"

obviously false in classical mathematics, such as the injectivity of type formers. But we would argue that all such results are artifacts of the choice of encodings, and not meaningful mathematical statements.

$$
\begin{array}{rcl}
[\![x]\!]_\Gamma\ \rho & := & \rho(x) \\
[\![\mathscr{U}_i]\!]_\Gamma\ \rho & := & \langle \mathbf{U}_i\ ;\ \mathsf{c}_{\mathsf{U}}\ i \rangle \\
[\![\Omega]\!]_\Gamma\ \rho & := & \langle \mathbb{B}\ ;\ \mathsf{c}_{\Omega} \rangle \\
\\
[\![\Pi^{\mathscr{U}_j,\mathscr{U}_k}(y:F).\,G]\!]_\Gamma\ \rho & := & \big\langle (x \in \mathsf{el}\ [\![F]\!]_\Gamma\ \rho) \to (\mathsf{el}\ [\![G]\!]_{\Gamma,x:F}\ \langle \rho\ ;\ x\rangle) \\
 & & \quad ;\ \mathsf{c}_{\Pi\mathsf{U}\mathsf{U}}\ j\ k\ ([\![F]\!]_\Gamma\ \rho)\ (x \mapsto [\![G]\!]_{\Gamma,x:F}\ \langle \rho\ ;\ x\rangle)\big\rangle \\
[\![\Pi^{\Omega,\mathscr{U}_k}(y:F).\,G]\!]_\Gamma\ \rho & := & \big\langle (x \in [\![F]\!]_\Gamma\ \rho) \to (\mathsf{el}\ [\![G]\!]_{\Gamma,x:F}\ \langle \rho\ ;\ x\rangle) \\
 & & \quad ;\ \mathsf{c}_{\Pi\Omega\mathsf{U}}\ ([\![F]\!]_\Gamma\ \rho)\ (x \mapsto [\![G]\!]_{\Gamma,x:F}\ \langle \rho\ ;\ x\rangle)\big\rangle \\
[\![\Pi^{\mathscr{U}_j,\Omega}(y:F).\,G]\!]_\Gamma\ \rho & := & \forall x \in (\mathsf{el}\ [\![F]\!]_\Gamma\ \rho),\ [\![G]\!]_{\Gamma,x:F}\ \langle \rho\ ;\ x\rangle \\
[\![\Pi^{\Omega,\Omega}(y:F).\,G]\!]_\Gamma\ \rho & := & ([\![F]\!]_\Gamma\ \rho) \Rightarrow ([\![G]\!]_{\Gamma,x:F}\ \langle \rho\ ;\ *\rangle) \\
[\![\lambda(x:F).\,t]\!]_\Gamma\ \rho & := & x \mapsto ([\![t]\!]_{\Gamma,x:F}\ \langle \rho\ ;\ x\rangle) \\
[\![t\ u]\!]_\Gamma\ \rho & := & ([\![t]\!]_\Gamma\ \rho)\ ([\![u]\!]_\Gamma\ \rho) \\
\\
[\![\Sigma^{\mathscr{U}_j,\mathscr{U}_k}(y:F).\,G]\!]_\Gamma\ \rho & := & \big\langle (x \in \mathsf{el}\ [\![F]\!]_\Gamma\ \rho) \times (\mathsf{el}\ [\![G]\!]_{\Gamma,x:F}\ \langle \rho\ ;\ x\rangle) \\
 & & \quad ;\ \mathsf{c}_{\Sigma}\ j\ k\ ([\![F]\!]_\Gamma\ \rho)\ (x \mapsto [\![G]\!]_{\Gamma,x:F}\ \langle \rho\ ;\ x\rangle)\big\rangle \\
[\![\langle t\ ;\ u\rangle]\!]_\Gamma\ \rho & := & \langle [\![t]\!]_\Gamma\ \rho\ ;\ [\![u]\!]_\Gamma\ \rho\rangle \\
[\![\mathsf{proj}_1(t)]\!]_\Gamma\ \rho & := & \pi_1([\![t]\!]_\Gamma\ \rho) \\
[\![\mathsf{proj}_2(t)]\!]_\Gamma\ \rho & := & \pi_2([\![t]\!]_\Gamma\ \rho) \\
\\
[\![\mathbb{N}]\!]_\Gamma\ \rho & := & \langle \mathsf{N}\ ;\ \mathsf{c}_{\mathsf{emb}}\ \mathsf{N}\rangle \\
[\![0]\!]_\Gamma\ \rho & := & 0 \\
[\![\mathsf{S}\ t]\!]_\Gamma\ \rho & := & \mathsf{S}\ ([\![t]\!]_\Gamma\ \rho) \\
[\![\mathbb{N}\text{-}\mathsf{elim}(P,t_0,t_S,n)]\!]_\Gamma\ \rho & := & \mathsf{N}\text{-}\mathsf{elim}(\mathsf{el} \circ ([\![P]\!]_\Gamma\ \rho), [\![t_0]\!]_\Gamma\ \rho, [\![t_S]\!]_\Gamma\ \rho, [\![n]\!]_\Gamma\ \rho) \\
\\
[\![\Box A]\!]_\Gamma\ \rho & := & \langle [\![A]\!]_\Gamma\ \rho\ ;\ \mathsf{c}_{\Box}\ ([\![A]\!]_\Gamma\ \rho)\rangle \\
[\![\diamond t]\!]_\Gamma\ \rho & := & * \\
[\![\Box\text{-}\mathsf{elim}(t)]\!]_\Gamma\ \rho & := & * \\
\\
[\![A/R]\!]_\Gamma\ \rho & := & \langle ([\![A]\!]_\Gamma\ \rho)/([\![R]\!]_\Gamma\ \rho)\ ;\ \mathsf{c}_{\mathsf{Q}}\ ([\![A]\!]_\Gamma\ \rho)\ ([\![R]\!]_\Gamma\ \rho)\rangle \\
[\![\pi(t)]\!]_\Gamma\ \rho & := & \text{the equivalence class of } [\![t]\!]_\Gamma\ \rho \text{ in the quotient} \\
[\![\mathsf{Q}\text{-}\mathsf{elim}(A,t,u,v)]\!]_\Gamma\ \rho & := & ([\![t]\!]_\Gamma\ \rho)\ x \qquad \text{for any } x \in ([\![v]\!]_\Gamma\ \rho) \\
\\
[\![\mathsf{Id}(A,t,u)]\!]_\Gamma\ \rho & := & \langle \{x \in \{*\} \mid [\![t]\!]_\Gamma\ \rho = [\![u]\!]_\Gamma\ \rho\} \\
 & & \quad ;\ \mathsf{c}_{\mathsf{Id}}\ ([\![A]\!]_\Gamma\ \rho)\ ([\![t]\!]_\Gamma\ \rho)\ ([\![u]\!]_\Gamma\ \rho)\rangle \\
[\![\mathsf{Idrefl}(t)]\!]_\Gamma\ \rho & := & * \\
[\![\mathsf{Idpath}(t)]\!]_\Gamma\ \rho & := & * \\
[\![\mathsf{J}(A,t,B,u,t',e)]\!]_\Gamma\ \rho & := & [\![u]\!]_\Gamma\ \rho \\
\\
[\![\bot]\!]_\Gamma\ \rho & := & \emptyset \\
[\![\bot\text{-}\mathsf{elim}(A,t)]\!]_\Gamma\ \rho & := & \text{undefined} \\
\\
[\![t \sim_A u]\!]_\Gamma\ \rho & := & \{x \in \{*\} \mid [\![t]\!]_\Gamma\ \rho = [\![u]\!]_\Gamma\ \rho\} \\
[\![\mathsf{refl}(t)]\!]_\Gamma\ \rho & := & * \\
[\![\mathsf{transp}(F,t,G,u,t',e)]\!]_\Gamma\ \rho & := & * \\
[\![\mathsf{cast}(A,B,e,t)]\!]_\Gamma\ \rho & := & [\![t]\!]_\Gamma\ \rho \\
[\![\mathsf{castrefl}(A,t)]\!]_\Gamma\ \rho & := & *
\end{array}
$$

Figure 4.6: Interpretation of $\mathrm{CC}^{\mathrm{obs}}$ in the Standard Model

Optional Features and Extensions of $\mathrm{CC}^{\mathrm{obs}}$

5

In this chapter, we investigate several variants and extensions of the observational calculus of constructions.

In section 5.1, we add a rule to the cast operator so that it is reduces when applied to a reflexive identity proof. We discuss normalization of the resulting system, and we give a proof of decidability of conversion. Then in section 5.2, we go over two alternative presentations of the observational equality: a heterogeneous equality and an equality with no reduction rules. In section 5.3, we explain how to add non-recursive indexed inductive types to the system, and we discuss some difficulties with universe levels. Finally, in section 5.4 we add a universe $\mathbb{P}$ of proof-*relevant* impredicative types to $\mathrm{CC}^{\mathrm{obs}}$ to increase its computational power. We show that if we extend the observational equality and the cast operator to this universe, type-checking becomes undecidable.

5.1 Cast and Reflexivity

5.1.1 Observational versus Inductive

In $\mathrm{CC}^{\mathrm{obs}}$, we have a bit of an awkward trade-off between two different notions of propositional equality.

On the one hand, we have the observational equality $t \sim_A u$, a proof-irrelevant proposition. The observational equality has strong selling points: not only does it validate the principle of uniqueness of identity proofs by definition (thanks to computational irrelevance), but it is also equipped with reduction rules that bake in useful reasoning principles such as function extensionality and proposition extensionality.

But being computationally irrelevant also has its drawbacks: because the equality proofs carry no information, the eliminator for the observational equality has to compute on types constructors instead. This means that cast will only reduce fully when it is applied to closed types. To make up for this weakness, $\mathrm{CC}^{\mathrm{obs}}$ provides the axiom castrefl which is a proof of observational equality between a cast along reflexivity and the identity function.

$$\textsc{Cast-Refl}\quad \frac{\Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash e : A \sim_{\mathcal{U}_i} A : \Omega}{\Gamma \vdash \mathsf{castrefl}(A, t) : t \sim_A \mathsf{cast}(A, A, e, t) : \Omega}$$

Note that the rule for castrefl is phrased with an arbitrary proof of equality between A and itself. This is equivalent to the statement using refl because of proof irrelevance.

On the other hand, we have the inductive equality $\mathsf{Id}(A, t, u)$ which is a proof-relevant inductive type, analogue of the Martin-Löf identity type. As such, it comes equipped with the usual J eliminator and its

computation rule.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash B : \Pi(x : A).\, \mathsf{Id}(A, t, x) \to \mathscr{U}_j \qquad \Gamma \vdash u : B\ t\ \mathsf{Idrefl}(t)}{\Gamma \vdash \mathsf{J}(A, t, B, u, t, \mathsf{Idrefl}(t)) \Rightarrow u : B\ t\ \mathsf{Idrefl}(t)}$$

Id is also stronger than the Martin-Löf identity type since it validates the signature reasoning principles of CC^obs that are UIP, function extensionality and proposition extensionality. However, computing on reflexivity comes at a price: for the inductive equality, the principle of UIP only holds up to a propositional equality, and the same goes for function extensionality and proposition extensionality.

All in all, the difference is rather subtle. The two relations are logically equivalent, but they do not satisfy the exact same computation rules. As if the distinction between propositional and definitional equality wasn't a sufficient source of confusion for newcomers to type theory!

5.1.2 The System CC^obs+

In hope to reconciliate these two propositional equalities, we introduce a variation on the observational calculus of constructions, that we call CC^obs+. This new system supports all the rules of chapter 3, plus the rule Cast-Refl+ which promotes the castrefl axiom to a definitional equality.

$$\textsc{Cast-Refl+}\quad \frac{\Gamma \vdash t : A : \mathscr{U}_i \qquad \Gamma \vdash e : A \sim_{\mathscr{U}_i} A : \Omega}{\Gamma \vdash \mathsf{cast}(A, A, e, t) \equiv t : A : \mathscr{U}_i}$$

In CC^obs+, we can define a J eliminator for the observational equality that computes on reflexivity. To do so, we simply replicate the definition described in Subsection 3.2.11, and in presence of rule Cast-Refl+ it will actually satisfy the computation rule on reflexivity as a definitional equality. Therefore, the observational equality of CC^obs+ really has the best of both worlds: not only is it definitionally proof irrelevant, but its eliminator also supports the computation rule on reflexivity by definition.

The J eliminator for the observational equality is defined as $J(A, t, B, u, t', e) := \mathsf{cast}(B\ t\ \mathsf{refl}, B\ t'\ e, \mathsf{eq_J}(A, t, B, t', e), u)$

Naturally, extending the convertibility relation of CC^obs means that we need a new algorithm to decide conversion for CC^obs+ terms. We will describe two such algorithms below, but we were only able to supplement the second algorithm with a proof of correction. We leave the first one as a conjecture for further work.

5.1.3 Implementing Cast-Refl+ with reduction rules

The first idea that we might want to try is to add a new reduction rule for $\mathsf{cast}(A, B, e, t)$, so that it will actually reduce to t in case the types A

and B are convertible.

$$\textsc{Cast-Refl-Red}\quad \frac{\Gamma \vdash B : \mathscr{U}_i \qquad \Gamma \vdash t : A : \mathscr{U}_i \qquad \Gamma \vdash e : A \sim_{\mathscr{U}_i} B : \Omega \qquad \Gamma \vdash A \equiv B : \mathscr{U}_i}{\Gamma \vdash \mathsf{cast}(A, B, e, t) \Rightarrow t : A}$$

This is a bit careless however, as this rule breaks the determinstic property of our weak-head reduction strategy. For instance, in the case of a cast between two convertible Π-types, both rule Cast-Refl-Red and Cast-Π apply.

$$\mathsf{cast}(\Pi(x : A).\, B, \Pi(x : A).\, B, \mathsf{refl}, f)$$

$$f \qquad\qquad \lambda\, x\, .\, \mathsf{cast}(B[x := a], B[x := a], \mathsf{refl}, f\, \mathsf{cast}(A, A, \mathsf{refl}, x))$$

Note that we need to reduce under the λ-abstraction to unify this critical pair, but reducing under binders is not allowed in the weak-head reduction strategy. A non-deterministic *and* non-confluent reduction strategy is no good, so rule Cast-Refl-Red is not a reasonable addition to the weak-head reduction.

In fact, the reader can check that all critical pairs introduced by rule Cast-Refl-Red can be unified by reducing under binders. Therefore, this rule is more reasonable in the context of a deep reduction strategy.

To avoid such conflicts with the reduction rules on constructors, we can restrict our rule to apply only when A and B are neutral types. This way, we recover a deterministic reduction strategy, although we lose stability of weak-head reduction under substitution.

$$\textsc{Cast-Refl-Neutral}\quad \frac{\Gamma \vdash t : A : \mathscr{U}_i \qquad \Gamma \vdash B : \mathscr{U}_i \qquad \Gamma \vdash e : A \sim_{\mathscr{U}_i} B : \Omega \qquad \Gamma \vdash A \equiv B : \mathscr{U}_i \qquad A \text{ and } B \text{ neutral}}{\Gamma \vdash \mathsf{cast}(A, B, e, t) \Rightarrow t : A}$$

Still, rule Cast-Refl-Neutral is by no means innocuous, as it dramatically changes the behavior of the system. Unlike all the other reduction rules of CC^obs+, it is not triggered by a redex consisting of a destructor applied to a constructor, but rather by a convertibility premise. As a consequence, it deeply intertwines weak-head reduction with conversion checking: when computing the head normal form of a term, it might be necessary to check convertibility of two terms, which requires putting them in weak head normal form, etc.

Normalization Type casting operators that reduce on identical types have a long history in type theory. In the context of impredicative systems, they were notably studied by Girard for System F [46], and by Werner for pCIC [50]. In both cases, it unfortunately turns out that the resulting system loses normalization for open terms [5].

This incompatibility between impredicativity and type casting can be interpreted as a consequence of the breakage of parametricity. Indeed, the operator $\mathsf{cast} : (A\ B\ : \mathscr{U}_i) \to A \to B$ is fundamentally anti-parametric because in case $A \equiv B$ it reduces to t, which satisfies the parametricity

[46]: Girard (1972), "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur"

[50]: Werner (2008), "On the Strength of Proof-Irrelevant Type Theories"

[5]: Abel et al. (2020), "Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality"

predicate associated to A but not necessarily the predicate associated to B. But normalization models for impredicative type theories are usually built using reducibility candidates, which rely crucially on parametricity to work.

Fortunately, we expect that our system CC^{obs+} will avoid these issues, because cast only computes in the predicative fragment and proof-irrelevance once again comes to our rescue in the impredicative layer. Therefore, we feel justified in conjecturing that this system is in fact normalizing.

Conjecture 5.1.1 CC^{obs+} *with computation rules for* cast *on neutral types is normalizing.*

As further evidence for this conjecture, we remark that the situation is very similar to reduction rules for cast on type constructors, which are certainly not parametric either. In fact, in section 5.4 we will show that extending these rules to a proof-relevant impredicative universe breaks normalization for the exact same reason. Nonetheless, by restricting computation to the predicative layer, we were able to prove normalization for CC^{obs} in chapter 4.

5.1.4 Implementing Cast-Refl+ in conversion checking

In conclusion to the previous section, our attempt at implementing rule Cast-Refl+ with reduction is not a resounding success. The resulting reduction strategy cannot be defined independently from conversion checking, and it is not even stable under substitution. At that point, it is not clear that we gain much by performing said reductions.

Here, it is tempting to draw a parallel with the η-equality of functions. Just like rule Cast-Refl+, the η-equality rule does not fit the general pattern of reducing a redex, and it does not lend itself too well to implementation *via* reduction rules either—both the η-expansion and the η-reduction ideas are unsatisfying [51]. In practice, all major proof assistants chose to implement η during the conversion checking phase: when the algorithm needs to decide convertibility between a function and a neutral term, it performs an η expansion of the neutral term and calls itself recursively.

[51]: Lennon-Bertrand (2022), "À bas l'η – Coq's troublesome η-conversion"

This is also how our conversion checking algorithm from chapter 4 proceeds.

Therefore we suggest that it might be wise to imitate this strategy for Cast-Refl+, *i.e.* to give up on an implementation *via* reduction rules and to handle that equality during conversion checking instead, as described in the work of Allais *et al.* [52]. Concretely, we propose the following informal algorithm for conversion checking in CC^{obs+}:

[52]: Allais et al. (2013), "New Equations for Neutral Terms: A Sound and Complete Decision Procedure, Formalized"

Input: A context Γ, three terms A, t, u, and an integer i.
Output: 'yes' if $\Gamma \vdash t \equiv u : A : \mathcal{U}_i$, 'no' otherwise.
Begin
  $A \longleftarrow$ the weak head normal form of A
  $t \longleftarrow$ the weak head normal form of t
  $u \longleftarrow$ the weak head normal form of u
  if A starts with a type constructor **then**
    proceed as usual
  else
    if $t = \mathsf{cast}(B, C, e, t')$ **and** u does not start with cast **then**
      do a recursive call to decide whether $\Gamma \vdash B \equiv C : \mathcal{U}_i$
      if 'yes' **then**
        do a recursive call on t' and u
      if 'no' **then**
        return 'no'
    else if t does not start with cast **and** $u = \mathsf{cast}(B, C, e, u')$ **then**
      do a recursive call on B and C, *etc*, as above
    else
      proceed as usual

Figure 5.1: Conversion checking algorithm for CC^{obs+}

Theorem 5.1.2 *The algorithm sketched above decides conversion for* CC^{obs+}.

We have formalized a proof of decidability for CC^{obs+} with a similar algorithm in the Agda proof assistant. The proof is available on GitHub at `https://github.com/CoqHott/logrel-mltt`, on the branch `impredicativity-cast-compute-refl`. The source code can also be browsed at *[CCobs-plus.agda]*.

5.2 Variations on the Observational Equality

5.2.1 A Proof-Irrelevant Heterogeneous Equality

In CC^{obs}, we can form the observational equality of two terms only when they have the same type. In other words the observational equality is *homogeneous*, just like the definitional equality or the inductive equality. But in their seminal work on observational type theory, Altenkirch, McBride and Swierstra decided to use a *heterogenous* equality [13], arguing that it results in simpler rules. In this section, we explain how to modify the presentation of CC^{obs} so that it uses a proof-irrelevant heterogeneous equality instead of the homogeneous observational equality, and we discuss the consequences.

[13]: Altenkirch et al. (2007), "Observational equality, now!"

First, we can change the formation rule of the equality so that it applies between two terms with different types:

$$\textsc{Heterogenous-Eq-Form}\quad \frac{\Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash u : B : \mathcal{U}_i}{\Gamma \vdash t \;{}_A{\sim}_B\; u : \Omega}$$

Note that we are now using two indices in the type $t\ {}_A{\sim}_B\ u$, one on the left for the type of t and one on the right for the type of u. Remark also that we do not ask for a proof of equality between the type A and B in order to form the type $t\ {}_A{\sim}_B\ u$. Unlike the "path-over" equality of cubical type theory, our heterogeneous equality can be used to compare two terms with completely unrelated types.

We also replace rules Refl, Transport-Ω and Cast with the following rules, which should be self-explanatory:

Heterogeneous-Transport-Ω

$$\frac{\Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \Omega \qquad \Gamma \vdash u : B[x := t] : \Omega \qquad \Gamma \vdash t' : A : \mathcal{U}_i \qquad \Gamma \vdash e : t\ {}_A{\sim}_A\ t' : \Omega}{\Gamma \vdash \mathsf{transp}(A, t, B, u, t', e) : B[x := t'] : \Omega}$$

Heterogeneous-Cast

$$\frac{\Gamma \vdash e : A\ {}_{\mathfrak{s}}{\sim}_{\mathfrak{s}}\ B : \Omega \qquad \Gamma \vdash t : A : \mathfrak{s}}{\Gamma \vdash \mathsf{cast}(A, B, e, t) : B : \mathfrak{s}}$$

Heterogeneous-Refl

$$\frac{\Gamma \vdash t : A : \mathcal{U}_i}{\Gamma \vdash \mathsf{refl}(t) : t\ {}_A{\sim}_A\ t : \Omega}$$

Just like with the homogeneous equality, the heterogeneous equality plays the role of an eliminator of the universe. In other words, the term $t\ {}_A{\sim}_B\ u$ computes by comparing the head constructors of the types A and B. If the constructors happen to match, $t\ {}_A{\sim}_B\ u$ reduces to the definition of the observational equality for the corresponding base type, and if A and B have incompatible head constructors, then $t\ {}_A{\sim}_B\ u$ reduces to $\bot$.

Reducing to $\bot$ on non-matching types is optional. It is a reasonable axiom, but it also adds an unnecessary constraint on the models of the theory.

Naturally, we also adjust the rules that describe equality on the base types. For instance, the heterogeneous equality between two dependent functions now reads as follows:

Heterogeneous-Eq-Fun

$$\frac{\Gamma \vdash f : \Pi^{\mathcal{U}_i, \mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)} \qquad \Gamma \vdash g : \Pi^{\mathcal{U}_i, \mathcal{U}_j}(x : A').\, B' : \mathcal{U}_{\max(i,j)}}{\Gamma \vdash \begin{array}{l} f\ {}_{\Pi AB}{\sim}_{\Pi A'B'}\ g \Rightarrow \\ \Pi(t : A)\ (u : A')\ .\ (t\ {}_A{\sim}_{A'}\ u) \to (f\ t\ {}_{B[t]}{\sim}_{B'[u]}\ g\ u) \end{array} : \Omega}$$

Remark that contrary to the homogeneous rule Eq-Fun, we do not need to use the type casting operator in the reduction rule. This is exactly what makes the heterogeneous equality nicer than the homogeneous equality: it can be defined independently from cast.

In counterpart, the heterogeneous equality brings a couple of interesting quirks to the table. In the rule above, remark that if the domains A and A' are incompatible, then the equality type $f\ {}_{\Pi AB}{\sim}_{\Pi A'B'}\ g$ is equivalent to:

$$\Pi(t : A)\ (u : A')\ .\ \bot \to (f\ t\ {}_{B[t]}{\sim}_{B'[u]}\ g\ u)$$

Since this type is always inhabited, any two functions with incompatible domains are equal. This property becomes even more unsettling when we notice that it implies that the heterogeneous equality is not transitive: the boolean identity function is equal to the identity on $\mathbb{N}$, which is equal to the boolean negation—but the boolean identity is not equal to the negation.

The only way out is to consider that the heterogeneous equality mathematically makes sense only when it is combined with a proof of equality of the two types (in which case it becomes equivalent to the homogeneous equality with transport).

As far as the meta-theoretical properties are concerned, picking a heterogeneous equality does not change much. The proof of normalization goes through with a few minor tweaks here and there, and similarly for the proof of consistency. Thus, the choice between homogeneous and heterogeneous is mostly a matter of taste.

5.2.2 Equality Without Computation

In fact, the exact nature of the observational equality does not really matter. As a computationally irrelevant property, its sole purpose is to add a logical constraint to cast so that it becomes impossible to cast between two incompatible types in an empty context.

Two types are incompatible if they start with different head constructors. Since cast between incompatible types are stuck terms, we want to make sure they can't be formed in the empty context, lest we get non-canonical integers.

Therefore, any alternative definition is fine as long as it does not add equalities between incompatible types in an empty context, and as long as it allows us to give a type to all the reduction rules for cast. For instance, a cast between two dependent sums reduces to a pair of cast (rule Cast-Σ), which means that in particular, we need to derive a proof of $A \sim_{\mathcal{U}} A'$ from a proof of $\Sigma(x : A).\, B \sim_{\mathcal{U}} \Sigma(x : A').\, B'$.

But there is really nothing that requires these derivations to come from reduction rules. In fact, we can completely do away with the computation rules for the observational equality, and replace them with proof-irrelevant axioms – for instance the following two axioms for equality on dependent sums:

$$\begin{aligned}
&\mathsf{eq{-}fst} \quad : \quad \Sigma(x : A).\, B \sim_{\mathcal{U}} \Sigma(x : A').\, B' \;\to\; A \sim_{\mathcal{U}} A' \\
&\mathsf{eq{-}snd} \quad : \quad \Pi(e : \Sigma(x : A).\, B \sim_{\mathcal{U}} \Sigma(x : A').\, B') \;. \\
&\qquad\qquad \Pi(a : A) \;.\; B[x := a] \sim_{\mathcal{U}} B'[x := \mathsf{cast}(A, A', \mathsf{eq{-}fst}\ e, a)].
\end{aligned}$$

And the resulting system still enjoys pretty much all the properties of CC$^{\text{obs}}$, except that the proof of function extensionality is not given by reflexivity anymore. This variation on observationaly type theory has been explored and implemented by Atkey [53], who argues that this avoids some problems with explosion of the size of the types.

[53]: Atkey (2017), "Simplified Observational Type Theory"

5.3 Non-Recursive Indexed Inductive Types

In our presentation of the system, we described the type of natural numbers, the dependent sums and the inductive equality which are three examples of inductive types. Together, they cover the three fundamental features of inductive types: recursion, type dependency, and indices. But in practice, proof assistant users are understandably not too fond of encoding their inductive definitions with these three basic types—the standard approach is rather to have a general scheme for inductive definitions.

Technically speaking, the dependent sums of CC$^{\text{obs}}$ are *negative*, meaning they are defined with projections and an eta-equality rule, not as an inductive type with a single constructor. Close enough.

In this section, we discuss the extension CC^obs with a scheme for non-recursive indexed inductive types. Now, inductive schemes have this unfortunate tendency to unleash notation hell, with vectors of parameters, indices and constructors. We will not be able to escape this completely, but we still make some simplifying assumptions: no mutual definitions, only one index in the type signature and only one argument per constructor. Concretely, a generic definition might look like this:

$$\begin{array}{l}\mathsf{Inductive}\ I : A \to \mathscr{U}_j := \\ |\quad c_0 \ :\ \Pi(x : B_0).\, I\, t_0 \\ |\quad \ldots \\ |\quad c_n \ :\ \Pi(x : B_n).\, I\, t_n\end{array}$$

Note that it is possible to use dependent sums to pack multiple indices or parameters in this definition, and thus these restrictions are not too serious.

This definition is well-formed if

- A is a closed type of sort $\mathscr{U}_i$, and $i \leq j$
- for all k the term B_k is a closed type of sort $\mathbb{s}_k$ such that $\mathbb{s}_k$ is either Ω or a universe bounded by $\mathscr{U}_j$, and
- for all k the term t_k is a term of type A in context B_k.

The constraint $i \leq j$ is not required in proof assistats such as Coq or Agda. But in an observational type theory, it is essential to prevent inconsistencies—see Subsection 5.3.1.

The elimination rule generated by this definition is the following:

Ind-elim
$$\frac{\Gamma \vdash P : \Pi(x : A).\, I\, x \to \mathbb{s} \qquad \{\ \Gamma \vdash p_k : \Pi(x : B_k).\, P\, t_k\ (c_k\ x)\ \}_{1 \leq k \leq n} \qquad \Gamma \vdash a : A : \mathscr{U}_i \qquad \Gamma \vdash u : I\, a : \mathscr{U}_j}{\Gamma \vdash \mathsf{Ind{-}elim}(P, p_1, \ldots, p_n, a, u) : P\, a\, u : \mathbb{s}}$$

Ind-comp
$$\frac{[\ldots]}{\Gamma \vdash \mathsf{Ind{-}elim}(P, p_1, \ldots, p_n, a, c_k\ b) \Rightarrow p_k\ b : P\, t_k[x := b]\ (c_k\ b) : \mathbb{s}}$$

and the observational equality between two instances of I simply reduces to the equality between the indices.

Ind-eq
$$\frac{\Gamma \vdash a : A : \mathscr{U}_i \qquad \Gamma \vdash a' : A : \mathscr{U}_i}{\Gamma \vdash I\, a \sim_{\mathscr{U}_j} I\, a' \Rightarrow a \sim_A a' : \Omega}$$

Equality between two constructors is also straightforward: if the constructors match, it reduces to the equality of the arguments, and it reduces to $\bot$ otherwise.

Ind-Cons-eq
$$\frac{\Gamma \vdash b : B_k : \mathbb{s}_k \qquad \Gamma \vdash b' : B_k : \mathbb{s}_k}{\Gamma \vdash c_k\ b \sim_{I\ \_} c_k\ b' \Rightarrow b \sim_{B_k} b' : \Omega}$$

Ind-Cons-neq
$$\frac{\Gamma \vdash b : B_k : \mathbb{s}_k \qquad \Gamma \vdash b' : B_\ell : \mathbb{s}_\ell}{\Gamma \vdash c_k\ b \sim_{I\ \_} c_\ell\ b' \Rightarrow \bot : \Omega}\, k \neq \ell$$

As with the Id types from Subsection 3.2.11, applying cast to a canonical inhabitant of $I\ a$ might produce a non-canonical term. For instance, consider the following term in a context with $e : a \sim a'$.

$$\mathsf{cast}(I\, a, I\, a', e, c_0\ b)$$

We have no way to reduce it to an element of $I\ a'$ of the form $c_0\ b'$, because such a term would only be well-typed if $t_0[x := b']$ is convertible to a'. As we have no way to satisfy this constraint, we add n new canonical terms $c_0^{\sim}, ..., c_n^{\sim}$ that relax the convertibility constraint to an observational equality:

$$\frac{\Gamma \vdash a : A : \mathcal{U}_i \qquad \Gamma \vdash b : B_k : \mathbb{s}_k \qquad \Gamma \vdash e : t_k[x := b] \sim_A a : \Omega}{\Gamma \vdash c_k^{\sim}(b, e) : I\ a : \mathcal{U}_j}$$

The observational equality is defined on these new terms just like it is defined on the regular constructors:

$$\frac{\Gamma \vdash b, b' : B_k : \mathbb{s}_k \qquad \Gamma \vdash e : t_k[x := b'] \sim_A t_k[x := b] : \Omega}{\Gamma \vdash c_k\ b \sim_{I\ \_} c_k^{\sim}(b', e) \Rightarrow b \sim_{B_k} b' : \Omega}$$

This rule has also a symmetric counterpart

$$\frac{\begin{array}{c}\Gamma \vdash a : A : \mathcal{U}_i \qquad \Gamma \vdash b, b' : B_k : \mathbb{s}_k \\ \Gamma \vdash e : t_k[x := b] \sim_A a : \Omega \qquad \Gamma \vdash e : t_k[x := b'] \sim_A a : \Omega\end{array}}{\Gamma \vdash c_k^{\sim}(b, e) \sim_{I\ a} c_k^{\sim}(b', e') \Rightarrow b \sim_{B_k} b' : \Omega}$$

The eliminator of I computes on the new constructors *via* cast.

$$\frac{\begin{array}{c}\Gamma \vdash P : \Pi(x : A).\, I\ x \to \mathbb{s} \qquad \{\ \Gamma \vdash p_k : \Pi(x : B_k).\, P\ t_k\ (c_k\ x)\ \}_{1 \le k \le n} \\ \Gamma \vdash a : A : \mathcal{U}_i \qquad \Gamma \vdash e : t_k[x := b] \sim_A a \qquad e' := \mathsf{ap}\ P\ e\ b\end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{Ind{-}elim}(P, p_1, ..., p_n, a, c_k^{\sim}(b, e)) \Rightarrow \\ \mathsf{cast}(P\ t_k[x := b]\ (c_k\ b), P\ a\ (c_k^{\sim}(b, e)), e', p_k\ b)\end{array} : P\ a\ (c_k^{\sim}(b, e)) : \mathbb{s}}$$

And finally, the computation rules for cast are straightforward to define—they accumulate the equalities in the second argument of $c_k^{\sim}$:

$$\frac{\Gamma \vdash a : A : \mathcal{U}_i \qquad \Gamma \vdash b : B_k : \mathbb{s}_k \qquad \Gamma \vdash e : t_k[x := b] \sim_A a : \Omega}{\Gamma \vdash \mathsf{cast}(I\ t_k[x := b], I\ a, e, c_k\ b) \Rightarrow c_k^{\sim}(b, e) : I\ a : \mathcal{U}_j}$$

$$\frac{\begin{array}{c}\Gamma \vdash a, a' : A : \mathcal{U}_i \\ \Gamma \vdash e : a \sim_A a' : \Omega \qquad \Gamma \vdash b : B_k : \mathbb{s}_k \qquad \Gamma \vdash e' : t_k[x := b] \sim_A a : \Omega\end{array}}{\Gamma \vdash \mathsf{cast}(I\ a, I\ a', e, c_k^{\sim}(b, e')) \Rightarrow c_k^{\sim}(b, e \cdot e') : I\ a' : \mathcal{U}_j}$$

All in all, these reduction rules are a bit of a eyesore, but the underlying idea is just to use the same trick as in our definition of the inductive equality. I expect that adding recursion to our inductive scheme does not pose fundamental problems, but the rules become much more complex.

5.3.1 Indices and Universe Levels

As we see, devising sensible computation rules for indexed inductive types is not too difficult. However, we still need to pay close attention to the consistency of our system when adding indexed inductive types: consider the following type with one index and no constructors:

$$\mathsf{Inductive\ Emb} : (\mathcal{U}_0 \to \mathcal{U}_0) \to \mathcal{U}_0 := \emptyset$$

In proof assistants based on intensional type theory such as Coq or Agda, this definition is accepted even though the index type is larger than $\mathcal{U}_0$. Of course this is safe and sound, as evidenced by the set-theoretic model of pCIC [54], but the situation is very different in CC^{obs}.

[54]: Timany et al. (2017), "Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)"

Remember that we defined the following computation rule for the observational equality:

$$\textsc{Ind-eq}\quad \frac{\Gamma \vdash X : \mathcal{U}_0 \to \mathcal{U}_0 \qquad \Gamma \vdash Y : \mathcal{U}_0 \to \mathcal{U}_0}{\Gamma \vdash \mathrm{Emb}\ X \sim_{\mathcal{U}_0} \mathrm{Emb}\ Y \Rightarrow X \sim_{\mathcal{U}_0 \to \mathcal{U}_0} Y : \Omega}$$

This means that Emb is an injection of $\mathcal{U}_0 \to \mathcal{U}_0$ into $\mathcal{U}_0$, which is obviously anti-classical. In fact the impredicativity of Ω is enough to derive a contradiction, with an argument due to Hur [55]: we start by defining

[55]: Hur (2010), "Agda with the excluded middle is inconsistent"

$$\begin{array}{lcl} P & : & \mathcal{U}_0 \to \mathcal{U}_0 \\ P & := & \lambda\ (x : \mathcal{U}_0)\ .\ \Box\exists(a : \mathcal{U}_0 \to \mathcal{U}_0).\ ((\mathrm{Emb}\ a \sim x) \wedge (a\ x\ \to \bot)) \end{array}$$

And then we show the following three theorems, which encode Russell's antinomy.

$$\begin{array}{lcl} \mathit{negP} & : & P\ (\mathrm{Emb}\ P) \to \bot \\ \mathit{negP}\ H_P & := & \mathsf{transp}(\mathcal{U}_0 \to \mathcal{U}_0,\ \mathsf{fst}(\Box\text{–}\mathsf{elim}(H_P)), \\ & & \qquad \lambda\ X\ .\ X\ (\mathrm{Emb}\ P) \to \bot, \\ & & \qquad \mathsf{snd}(\mathsf{snd}(\Box\text{–}\mathsf{elim}(H_P))),\ P,\ \mathsf{fst}(\mathsf{snd}(\Box\text{–}\mathsf{elim}(H_P)))) \end{array}$$

$$\begin{array}{lcl} \mathit{posP} & : & (P\ (\mathrm{Emb}\ P) \to \bot) \to P\ (\mathrm{Emb}\ P) \\ \mathit{posP}\ H_P & := & \diamond\langle P, \langle \mathsf{refl}(P), H_P\rangle\rangle \end{array}$$

$$\begin{array}{lcl} \mathit{abs} & : & \bot \\ \mathit{abs} & := & \mathit{negP}\ (\mathit{posP}\ \mathit{negP}) \end{array}$$

Thus we see that it is necessary to have a constraint on the universe level of indices in inductive types, because of rule Ind-eq. Of course, the same argument applies to parameters as well.

5.4 Proof-Relevant Impredicativity

5.4.1 CC^{obs} versus pCIC

Our system is named the "Observational Calculus of Constructions" to highlight the similarities with the Predicative Calculus of Inductive Constructions, the type theory which serves as the base of the Coq proof assistant [54]. Most notably, both systems enforce a separation between an impredicative universe of propositions and a hierarchy of predicative universes, with the idea that the predicative layer cannot tell apart two proofs of a same proposition.

[54]: Timany et al. (2017), "Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)"

Mind the different notation: Ω is the proof-*irrelevant* universe of propositions from CC^{obs}, and Prop is the proof-*relevant* universe of propositions from pCIC.

But there is an important difference in the treatment of impredicativity: contrary to CC^{obs}, the impredicative universe Prop of pCIC is *computationally relevant*. By this, we mean that there is no analogue of the rule

Proof-Irrelevance. Therefore, in pCIC proofs of propositions are subject to computation rules, just like the terms that inhabit the predicative sorts; and the normalization theorem applies to both layers.

Furthermore, pCIC allows inductive definitions in the universe of propositions Prop. Just like regular inductive types, an inductive proposition may have parameters, indices, many constructors with arguments, *etc.* but its eliminator is restricted to propositional predicates in order to avoid paradoxes. For instance, here is what an inductive definition of the *propositional* integers might look like in pCIC:

$$\begin{aligned}&\mathsf{Inductive}\ \mathbb{N}_p : \mathsf{Prop} :=\\ &\mid\ 0 : \mathbb{N}_p\\ &\mid\ \mathsf{S} : \mathbb{N}_p \to \mathbb{N}_p\end{aligned}$$

They work just like the usual integers, except that they cannot be eliminated into a type that is not a proposition.

However, there is an exception to this elimination constraint: if the inductive proposition is a *subsingleton*, *i.e.* it has at most one constructor and all constructor arguments are propositions, then large elimination is allowed. Large elimination of subsingletons is very important, as it plays the role of a bridge that connects the two layers of pCIC, and allows some of the raw logical power of impredicativity to leak to the predicative layer. More concretely, we can use impredicativity and large elimination to define all the integer functions that are provably computable in higher-order Peano Arithmetic as terms of type $\mathbb{N} \to \mathbb{N}$ in the predicative layer (and then some more!).

An inductive type is said to support large elimination if its eliminator can be used with arbitrary predicates.

Compare this situation to CC^{obs}, where most propositions are defined using impredicative encodings. Since impredicative encodings do not provide large elimination principles, the only propositions that may be eliminated into the predicative layer are the false proposition (*via* $\bot$−elim) and the observational equality (*via* cast). This impoverished communication between the two layers confines the power of impredicativity to the proof-irrelevant layer: while it may be possible to define very complex functions *via* an impredicative encoding of their graph, there is no way to apply them to proof-relevant integers. As a result, CC^{obs} cannot define more functions in the proof-relevant layer than MLTT (see section 4.3), despite being an impredicative type theory.

The observational equality is technically not an inductive proposition, but its role is similar enough to the inductive equality that it makes sense to compare cast to a large elimination principle.

This whole story makes a rather compelling argument for the addition of inductive propositions with subsingleton elimination to CC^{obs}. Unfortunately, proof irrelevance is a serious obstacle: how is the eliminator supposed to compute when we destroyed all computational information from the proofs? In the case of the observational equality we can get by with computation on types, but there is no way this trick generalizes to all subsingleton propositions. A more in-depth discussion of the difficulties with large elimination of a proof-irrelevant accessibility predicate can be found in the work of Gilbert *et al.* [6].

[6]: Gilbert et al. (2019), "Definitional Proof-Irrelevance without K"

5.4.2 Extending CC^obs with the Power of Proof-Relevant Impredicativity

Since Ω cannot support subsingleton elimination, it seems that the only way to recover the full power of impredicativity in CC^obs is to extend the system with an impredicative universe of proof-*relevant* propositions ℙ. Now this new universe cannot replace Ω, because proof-irrelevant axioms play an important role in the definition of the observational equality. Therefore, in this section we study an extension of CC^obs that supports both Ω and ℙ in the same system.

We use Prop for the universe of propositions in pCIC, and ℙ for the universe of proof-relevant propositions in CC^obs.

$$\textsc{Univ-Prop}\ \frac{\vdash \Gamma}{\Gamma \vdash \Omega : \mathcal{U}_0} \qquad \textsc{Univ-Prop-Rel}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{P} : \mathcal{U}_0}$$

Having two different universes of propositions is a bit confusing! But their roles do not overlap that much: while Ω is used to enforce proof-irrelevant logical constraint on types (including the observational equality), ℙ is only here as a tool for impredicative definitions. Since ℙ is an inhabitant of the universe $\mathcal{U}_0$, it means we can form observational equalities between inhabitants of ℙ. How should they compute?

A first idea is to mimic the rule for Ω and define the observational equality as logical equivalence. This way, we get propositional extensionality for the proof-relevant propositions.

Since the observational equality is a proof-irrelevant proposition, we need the squash to convert that equivalence to an element of Ω.

$$\textsc{Eq-}\mathbb{P}\ \frac{\Gamma \vdash A : \mathbb{P} \qquad \Gamma \vdash B : \mathbb{P}}{\Gamma \vdash A \sim_{\mathbb{P}} B \Rightarrow \|(A \to B) \wedge (B \to A)\| : \Omega}$$

Unfortunately, this definition is problematic from a computational perspective: if t is an inhabitant of some proof-relevant proposition A, then we should be able to obtain an inhabitant of B from a proof of $A \sim_{\mathbb{P}} B$. But since the proof of equality is a *squashed* equivalence, we have no way to use it to compute an element of B.

A second idea is to mimic the observational equality on the predicative universes, *i.e.* to pick reduction rules according to the head constructors of the propositions. Then, we extend the cast operator to work with proof-relevant propositions, too

$$\textsc{Cast-}\mathbb{P}\ \frac{\Gamma \vdash e : A \sim_{\mathbb{P}} B : \Omega \quad \Gamma \vdash t : A : \mathbb{P}}{\Gamma \vdash \mathsf{cast}(A, B, e, t) : B : \mathbb{P}}$$

along with reduction rules for $\mathsf{cast}(A, B, e, t)$ when A and B have matching head constructors, just like the predicative cast. This idea seems more reasonable, but it turns out to have the very unfortunate consequence of making conversion checking undecidable.

Theorem 5.4.1 *If we extend cast to the universe of proof-relevant impredicative propositions ℙ that supports inductive definitions, then conversion becomes undecidable.*

The proof of this fact will occupy us for the remainder of this section.

Outline of the Proof of Undecidability Our proof is based on the argument of Abel and Coquand [5], that we enhance to a proof of undecidability. The main idea is to use impredicativity and cast to approximate a fixed-point combinator.

[5]: Abel et al. (2020), "Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality"

Given a proof-relevant proposition $A : \mathbb{P}$ and a function $f : A \to A$, impredicativity makes it possible to give a type to the term

$$\Delta_f := \lambda x.\, f\,(x\,x)$$

which is one half of the fixed point combinator $Y_f := \Delta_f\,\Delta_f$. We cannot complete the combinator though, as the type of Δ_f does not allow self-application. But in an inconsistent context, cast can be used to convert between any two types whatsoever. Thus we can apply Δ_f to a type-cast of Δ_f, which results in a term that is somewhat similar to a fixed-point combinator.

Then, using this pseudo-fixed point combinator, we can build a term that loops through the iterates of any integer function:

Recall that $\mathbb{N}_{\mathrm{p}}$ is the inductive type of natural numbers in $\mathbb{P}$.

Lemma 5.4.2 *Let g be a closed term of type $\mathbb{N}_{\mathrm{p}} \to \mathbb{N}_{\mathrm{p}}$.*

In an inconsistent context, we can form a term dec_g of type $\mathbb{N}_{\mathrm{p}}$ which is convertible to 0 if there is a positive integer n such that $g^n\ 1 \equiv 0$, and diverges otherwise.

Of course, this is too much to ask of a conversion checking algorithm. Depending on the definition of g, deciding whether dec_g is convertible to 0 might amount to deciding whether a Turing machine halts. Therefore conversion and typing are undecidable for $\mathrm{CC}^{\mathrm{obs}} + \mathbb{P}$.

Remark that we can perform the exact same construction in Ω if we add proof-irrelevant natural numbers. But it does not cause any decidability issue, as we have a simple way to check for convertibility of proof-irrelevant terms: having the same type is sufficient! For this reason, we never perform reduction in the proof-irrelevant layers, and need not worry about these potentially divergent terms.

Definition of an Approximate Fixed Point Combinator In an inconsistent context Γ, we can use $\bot$−elim to derive a term

$$e \ : \ \Pi(X\,Y : \mathbb{P}).\, X \sim Y$$

that allows us to freely cast between any two types. We will use it to derive an approximate fixed point for any closed term f of type

$(\mathbb{N}_p \to \mathbb{N}_p) \to (\mathbb{N}_p \to \mathbb{N}_p)$. Consider the following definitions:

$$\begin{array}{lcl} \bot_p & : & \mathbb{P} \\ \bot_p & := & \Pi(X : \mathbb{P}).\, X \end{array}$$

$$\begin{array}{lcl} \Delta_f & : & \bot_p \to \mathbb{N}_p \to \mathbb{N}_p \\ \Delta_f & := & \lambda(x : \bot).\, f\ (x\ (\bot_p \to \mathbb{N}_p \to \mathbb{N}_p)\ x) \end{array}$$

$$\begin{array}{lcl} \Delta'_f & : & \bot_p \\ \Delta'_f & := & \lambda(X : \mathbb{P}).\, \mathsf{cast}(\bot_p \to \mathbb{N}_p \to \mathbb{N}_p, X, e\ (\bot_p \to \mathbb{N}_p \to \mathbb{N}_p)\ X, \Delta_f) \end{array}$$

$$\begin{array}{lcl} Y_f & : & \mathbb{N}_p \to \mathbb{N}_p \\ Y_f & := & \Delta_f\ \Delta'_f \end{array}$$

The term Y_f is the one that will play the role of a fixed point for f. It is only an *approximate* fixed point though, because it is not quite convertible to $f\ Y_f$. Instead, a tedious but straightforward computation using the definitional equality of CC^{obs} + $\mathbb{P}$ shows that

$$\begin{array}{rcll} \Gamma \vdash Y_f & \equiv & \Delta_f\ \Delta'_f & : \quad \mathbb{N}_p \to \mathbb{N}_p \\ & \Rightarrow^* & f\ (\alpha\ (\Delta_f\ (\beta\ \Delta'_f))) & \\ & \Rightarrow^* & f\ (\alpha\ (f\ (\alpha^2\ (\Delta_f\ (\beta^3\ \Delta'_f))))) & \\ & \Rightarrow^* & f\ (\alpha\ (f\ (\alpha^2\ (f\ (\alpha^4\ (\Delta_f\ (\beta^7\ \Delta'_f)))))))) & \\ & \Rightarrow^* & \ldots & \end{array}$$

where we define the auxiliary functions α and β as:

$$\begin{array}{lcl} \alpha & : & (\mathbb{N}_p \to \mathbb{N}_p) \to (\mathbb{N}_p \to \mathbb{N}_p) \\ \alpha & := & \lambda x.\, \mathsf{cast}(\mathbb{N}_p \to \mathbb{N}_p, \mathbb{N}_p \to \mathbb{N}_p, e\ (\mathbb{N}_p \to \mathbb{N}_p)\ (\mathbb{N}_p \to \mathbb{N}_p), x) \end{array}$$

$$\begin{array}{lcl} \beta & : & \bot_p \to \bot_p \\ \beta & := & \lambda x.\, \mathsf{cast}(\bot_p, \bot_p, e\ \bot_p\ \bot_p, x). \end{array}$$

Thus, Y_f is only a fixed point combinator up to a bunch of reflexive cast. This behavior is sufficient for our purposes, since all the α disappear when the function Y_f is applied to an actual integer. Now, given a closed term g of type $\mathbb{N}_p \to \mathbb{N}_p$ that terminates on all canonical inputs, we pose

$$\begin{array}{lcl} f & : & (\mathbb{N}_p \to \mathbb{N}_p) \to (\mathbb{N}_p \to \mathbb{N}_p) \\ f & := & \lambda(p : \mathbb{N}_p \to \mathbb{N}_p).\, \lambda(n : \mathbb{N}_p).\, \mathbb{N}\text{–}\mathsf{elim}(\mathbb{N}_p, 0, p\ (g\ n), n) \end{array}$$

and we obtain that $Y_f\ 1$ reduces to 0 if there exists a n such that $g^n\ 1 \equiv 0$, but diverges (for any reduction strategy) otherwise.

Here, the values 0 and 1 should be understood as the corresponding integers in $\mathbb{N}_p$.

Conversion is Undecidable CC^{obs} is expressive enough to encode Turing machines so that knowing whether there exists a n such that $g^n\ 1 = 0$ is undecidable. To complete the proof of undecidability of conversion, we still need to show that a term of type $\mathbb{N}_p$ that diverges for all reduction strategies is not convertible to 0. In other words, we need to show that conversion is not degenerate.

To establish this, we use a lemma of Geuvers, adapted to the Calculus of Inductive Constructions by Werner [56, lemma 2.19]. The lemma corresponds to a weak form of confluence: any well-typed term t that is $\beta\eta\iota$-equal to a weak head normal form t' actually reduces to an η-expanded form of t' for untyped $\beta\iota'$ reduction, a slightly modified version of untyped $\beta\iota$ reduction. If we take $t' = 0$ then we obtain that a well-typed term t that is $\beta\eta\iota$-equal to 0 has a normal form, and thus cannot be divergent.

[56]: Werner (1994), "Une Théorie des Constructions Inductives"

In order to apply this lemma to $\mathsf{CC}^{\mathrm{obs}}$, we need to extend the proof to strict propositions and type-casting operators. The extension is not too difficult: strict propositions do not contribute to the reduction behavior, and type-casting is very similar to an eliminator of an inductive type.

Is this situation salvageable? As we see, there is no obvious way to define how the observational equality should compute between elements of $\mathbb{P}$. Is the system $\mathsf{CC}^{\mathrm{obs}} + \mathbb{P}$ doomed, then?

Long story short, I do not know. Maybe there is a way to define a system that is better compartmentalized, so that both Ω and $\mathbb{P}$ can interact with the predicative layer without interacting together...

The Univalent Equality

6 Prefascist Types

As we conclude our explorations in the world of observational type theory and proof-irrelevant equalities, we turn to the radically different world of the univalent equality and of the cubical models of univalent type theory [21, 57]. We start our new journey with a chapter about *presheaves* in intensional type theory.

Presheaves are a versatile object in category theory that plays a significant role in the meta-theory of dependent type theory. In particular, presheaves form the basis of the cubical models of univalent type theory which will be the focus of the next chapter. Thus in this preparatory chapter, we discuss the relation between presheaves and intensional type theory.

We start this chapter with a short overview of the basic properties of presheaves in set theory in section 6.1. Then in section 6.2, we explain some well-known difficulties with defining presheaves of types in intensional type theory. In section 6.3 we introduce prefascist types, an alternative definition of presheaves in type theory proposed by Pédrot [32] to build models of type theory in type theory.

[32]: Pédrot (2020), "Russian Constructivism in a Prefascist Theory"

Given the nature of the topic, the reader should expect significantly more category theory in this chapter than in the rest of this thesis. We assume some amount of familiarity with objects such as functors, limits, or adjunctions.

6.1 Set-theoretic Presheaves

In this section, we recall the definition of presheaves and go over some of their uses. The reader who is already well acquainted with presheaves might want to skip this section and go directly to section 6.2, where we investigate presheaves from the angle of intensional type theory. But until then, we will be working informally in a constructive set theory with Grothendieck universes.

Grothendieck universes allow us to safely manipulate objects such as the set of all *small* sets, which is a *large* set. Grothendieck universes work very much like the predicative universe hierarchy in type theory.

Given any category $\mathscr{C}$, the category $\mathrm{Psh}(\mathscr{C})$ of *presheaves* on $\mathscr{C}$ is the category of functors from the opposite category $\mathscr{C}^{\mathrm{op}}$ to the category of small sets.

Definition 6.1.1 $\mathrm{Psh}(\mathscr{C}) := \mathrm{Hom}_{\mathrm{Cat}}(\mathscr{C}^{\mathrm{op}}, \mathrm{Set})$

In this definition, the $\mathrm{Hom}_{\mathrm{Cat}}$ notation should be understood in the 2-categorical sense: this object is a category, not a set. In other words, a presheaf is a functor from $\mathscr{C}^{\mathrm{op}}$ to Set, and a morphism of presheaves is a natural transformation.

In the usual fashion of category theory, this definition is slick and abstract, which results in a remarkably versatile object. But in counterpart, the definition alone is not too helpful in building intuition about presheaves—what are they good for?

Presheaves[1] were initially designed to capture the idea of a set continuously parametrized by a topological space X, by taking the lattice of open subsets of X as the index category $\mathscr{C}$. The definition was later generalized to an arbitrary category, but this topological legacy has left traces in the terminology, with words such as section and restriction. However in this thesis we will mostly be interested in presheaves for different reasons, one of them being that presheaves allow us to freely add colimits to a category.

1: sheaves, actually

If F is a presheaf on $\mathscr{C}$, then the elements of $F(p)$ are called sections of F over p, and the image of a morphism $F(\alpha)$ is called the restriction along α.

6.1.1 Presheaves as a Categorical Cocompletion

Suppose that $\mathscr{C}$ is locally small. Recall the definition of the Yoneda embedding よ:

A category is locally small if all of its hom-sets are small sets

Definition 6.1.2 *The Yoneda embedding is a functor from $\mathscr{C}$ to $\mathrm{Psh}(\mathscr{C})$ defined by*

$$\begin{aligned} よ &: \mathscr{C} \to \mathrm{Psh}(\mathscr{C}) \\ よ(c) &:= \mathrm{Hom}_{\mathscr{C}}(\_, c) \end{aligned}$$

The action of よ on morphisms is given by post-composition, which makes it into a proper functor from $\mathscr{C}$ to its presheaf category $\mathrm{Psh}(\mathscr{C})$.

Furthermore, the Yoneda lemma teaches us that this functor is fully faithful, and thus the representable presheaves can be understood as a copy of $\mathscr{C}$ sitting inside the larger category $\mathrm{Psh}(\mathscr{C})$.

A presheaf is called representable when it is in the image of よ

There is a precise sense in which $\mathrm{Psh}(\mathscr{C})$ is generated from this internal copy of $\mathscr{C}$ by freely adding all small colimits:

Theorem 6.1.1 *The category $\mathrm{Psh}(\mathscr{C})$ admits all small colimits, which can be computed point-wise.*

Theorem 6.1.2 *Any object of $\mathrm{Psh}(\mathscr{C})$ can be obtained as a small colimit of representable presheaves.*

Theorem 6.1.3 *Given a category $\mathscr{D}$ that has all small colimits and a functor $F : \mathscr{C} \to \mathscr{D}$, there is a unique cocontinuous functor $\widehat{F_0} : \mathrm{Psh}(\mathscr{C}) \to \mathscr{D}$ such that $F = \widehat{F_0} \circ よ$.*

Proofs can be found in MacLane et al. [58].

[58]: MacLane et al. (1994), *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*

Put together, these three theorems seem to paint a surprisingly simple picture of $\mathrm{Psh}(\mathscr{C})$: we can think of a presheaf as an amalgamated sum of objects from $\mathscr{C}$. This is why combinatorial models of spaces such as *simplicial sets* and *cubical sets* are defined as presheaves on carefully chosen categories. We will come back to this in the next chapter.

However, mental images are a double edged sword and can be just as misleading as they are enlightening. In particular, the reader should note that the Yoneda embedding does not preserve colimits that may already exist in $\mathscr{C}$, and as a consequence we should be careful to have different pictures for $よ(x + y)$ and $よ(x) + よ(y)$.

6.1.2 Presheaves as Generalized Categories of Sets

In addition to being cocomplete, categories of presheaves also have small limits and exponential objects [58].

[58]: MacLane et al. (1994), *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*

Theorem 6.1.4 *The category* $\mathrm{Psh}(\mathscr{C})$ *admits all small limits, which can be computed point-wise. Moreover, the Yoneda embedding preserves limits that exist in* $\mathscr{C}$*.*

Theorem 6.1.5 *The category* $\mathrm{Psh}(\mathscr{C})$ *is cartesian closed.*

This is only natural: presheaves are more or less indexed small sets, and as such they retain a large part of the structure of the category Set. In fact, they retain much more structure than just limits and colimits: we can basically replicate any construction from constructive set theory inside a category of presheaves—we can form powersets, the image of a morphism, dependent products, etc. However, the principle of excluded middle and the axiom of choice are not valid in most categories of presheaves.

The category of presheaves on $\mathscr{C}$ is the prototypical example of a *topos*, a category that shares most of the abstract categorical properties of Set.

In particular, presheaves have enough structure to interpret dependent products, dependent sums and inductive types. As a consequence, we can build models of Martin-Löf type theory in categories of presheaves [59].

[59]: Hofmann (1997), "Syntax and semantics of dependent types"

6.2 Type-theoretic Presheaves

Hopefully by now, the reader is convinced that presheaves are a simple yet versatile idea in category theory and thus a natural candidate for a translation into type theory. But while concrete first-order objects such as integers or finite trees are usually pretty straightforward to implement in type theory, higher-order objects that deal with predicates or functions usually come with their lot of complications—and presheaves are no exception.

In this section, we explain the difficulties with defining presheaves of types in intensional Martin-Löf Type Theory. We use the syntax of the Agda proof assistant for our type-theoretic definitions, without assuming Streicher's axiom K.

6.2.1 First Definition

Let us assume that we are working with a small category $\mathscr{C}$, and that we managed to encode it nicely inside type theory:

```
postulate
  Obj : Set
  Hom : Obj → Obj → Set
  id : (a : Obj) → Hom a a
  _∘_ : {a b c : Obj} (f : Hom b c) (g : Hom a b) → Hom a c
```

Arguments between curly braces are optional arguments, which will usually be inferred from the context.

Furthermore, we assume that the unit laws and the associativity of composition in $\mathscr{C}$ hold *definitionally*:

```
postulate
  ∘-id-left : {a b : Obj} (f : Hom a b) → (id b) ∘ f = f
  ∘-id-right : {a b : Obj} (f : Hom a b) → f ∘ (id a) = f
  ∘-assoc : {a b c d : Obj}
            (f : Hom c d) (g : Hom b c) (h : Hom a b)
            → f ∘ (g ∘ h) = (f ∘ g) ∘ h
```

By which we mean that the equality that appears in these three postulates is not the propositional equality (≡) but rather the definitional equality (=): the two sides are convertible.

In this chapter, we use the Agda conventions. This means that we use a triple equal symbol ≡ for the propositional equality. It has nothing to do with the definitional equality of chapter 3 or the reducible equality of chapter 4. The definitional equality of Agda is written with the usual equality symbol =.

Of course, using the definitional equality in Agda code like this is not allowed. This should rather be understood as a notational shorthand for the assumption that we have an actual implementation of $\mathscr{C}$ with instances of Hom, id and _∘_ that satisfy these three definitional equations. It could be the case for instance if we managed to express $\mathscr{C}$ as a full subcategory of Set, in which case the morphisms are functions, whose composition is associative by defintion. This might seem like an arbitrary and restrictive constraint on the category $\mathscr{C}$, but in Subsection 6.3.3 we will see that all small categories can in fact be presented in this way in a type theory with strict propositions.

Now, how should we go about defining the type of presheaves on $\mathscr{C}$? If we mimic the set-theoretic definition of a presheaf, we might write something that starts like this:

```
record presheaf : Set₁ where
  field
    F₀ : Obj → Set
    F₁ : {a b : Obj} → Hom a b → F₀ b → F₀ a
    {- ... -}
```

This defines the action of the presheaf on objects and morphisms, but we also need to verify some equations, as functors are supposed to preserve identities and composition. We can try to state them using the propositional equality _≡_:

```
    {- ... -}
    F_id : (a : Obj) → F₁ (id a) ≡ (λ x → x)
    F_comp : {a b c : Obj} (f : Hom a b) (g : Hom b c)
             → F₁ (g ∘ f) ≡ (λ x → F₁ f (F₁ g x))
```

and from there, all reasonable equalities such as

$$\mathsf{F}_1\ ((f \circ g) \circ h) \equiv (\mathsf{F}_1\ h \circ \mathsf{F}_1\ g) \circ \mathsf{F}_1\ f \tag{6.1}$$

can be proved through successive rewritings using F_id and F_comp. And sure enough, this definition does correspond to presheaves when we interpret type theory in the usual set-theoretic model [59]. But as bare MLTT lacks the extensional features of set theory, this definition hides some kinks that we might want to iron out.

[59]: Hofmann (1997), "Syntax and semantics of dependent types"

6.2.2 A Story about Higher Coherences

Interestingly, if we try to prove equation 6.1, we can devise two chains of rewritings that take the left hand side to the right hand side:

$$\begin{array}{ccccc} \mathsf{F}_1\ ((f \circ g) \circ h) & = & \mathsf{F}_1\ h \circ \mathsf{F}_1\ (f \circ g) & = & \mathsf{F}_1\ h \circ (\mathsf{F}_1\ g \circ \mathsf{F}_1\ f) \\ \| & & & & \| \\ \mathsf{F}_1\ (f \circ (g \circ h)) & = & \mathsf{F}_1\ (g \circ h) \circ \mathsf{F}_1\ f & = & (\mathsf{F}_1\ h \circ \mathsf{F}_1\ g) \circ \mathsf{F}_1\ f \end{array}$$

The horizontal equalities are instances of F_comp, and the two vertical equalities hold definitionally.

and these two chains of equalities result in two different witnesses of equation 6.1. We might hope to prove that these two witnesses are in fact equal ; but it is not possible without assuming additional axioms (remember that we are working in intensional MLTT here, where identity proofs are by no means unique!). This state of affairs is rather unsatisfying, and it certainly looks like that kind of nuisance that will show up uninvited in our proofs later on.

To understand the source of this irritating phenomenon, it is helpful to look at it under the lens of Homotopy Type Theory. As we explained in chapter 2, in HoTT we think of types as being *spaces* instead of sets, and the two different notions of equality (definitional and propositional) have very different interpretations:

- the definitional equality $a = b$ corresponds to an actual, on-the-nose equality between the two points, but
- the propositional equality $a \equiv b$ is the space of paths connecting a to b.

Since MLTT is a subset of HoTT, our tentative definition can be interpreted as talking about spaces and should ideally make sense from this perspective. Our definition would in fact work for presheaves of spaces if F_id and F_comp were definitional equalities, but when we relax these equalities to mere paths (propositional equalities) we are effectively adding superfluous information to the type—as we have seen, there are now several ways to use F_id and F_comp to prove what should really be a equality "on the nose" in the corresponding spaces. Thus instead of presheaves of spaces, we are defining something like presheaves of spaces with superfluous free loops attached to them.

Of course, we do not have access to the definitional equality internally in type theory, so we have to find another way to eliminate these extra paths. We can try to add new fields to the record presheaf that enforce propositional equations between the redundant equality witnesses, as follows:

$$\begin{aligned} \mathsf{F\_coh} : \forall f\ g\ h \rightarrow\ & (\mathsf{F\_comp}\ h\ (f \circ g)) \cdot ((\mathsf{F}_1\ h) \circ (\mathsf{F\_comp}\ g\ f)) \\ \equiv\ & (\mathsf{F\_comp}\ (g \circ h)\ f) \cdot ((\mathsf{F\_comp}\ h\ g) \circ (\mathsf{F}_1\ f)) \end{aligned}$$

_·_ denotes the transitivity of equality / composition of paths.

Indeed, it is possible to find a finite number of equations that account for all superfluous paths introduced by F_id and F_comp. But this approach will not get us very far: F_coh introduces superfluous paths between paths, which results in a type of presheaves with superfluous 2-dimensional loops. Going further in this direction only leads us

towards an infinite tower of equalities of ever increasing dimensions, with no apparent way to make it fit into a finite definition.

Here, we recognize a well-known pattern, as difficulties with infinite towers of coherences show up for numerous definitions in Homotopy Type Theory. The most famous such definition is that of *semi-simplicial types*, which was identified in the early years of HoTT [60], and remains an open problem as of today [61].

[61]: Buchholtz (2022), "Update on semisimplicial types in homotopy type theory"

Semi-simplicial types are presheaves on the category of finite ordinals and strictly increasing maps Δ_+. As we will explain shortly, if our type theory supports strict propositions, Δ_+ can be presented as a category that satisfies the composition and the unit laws definitionally, which means that the definition of semi-simplicial sets is a special case of the problem we are trying to solve. This is bad news!

6.2.3 Possible Workarounds

Thanks to the perspective of Homotopy Type Theory, we now understand that defining presheaves of types on a general category is delicate because of the intricate higher-dimensional structure that may be encoded in identity types. Even though the problem of simplicial types remains open, there are well-known ways to avoid coherence problems with presheaves in type theory.

Presheaves of hsets If types in HoTT correspond to spaces, there is a certain subclass of types that can be identified as sets. These are the *hsets*: types X equipped with a proof that all paths between any two points of X are equal

```
record hset : Set₁ where
  field
    X : Set
    is_hset : {a b : X} → (e1 e2 : a ≡ b) → e1 ≡ e2
```

Interestingly, the is_hset constraint removes all the non-trivial loops from X without introducing any higher-dimensional coherence problems ([HoTT], theorem 7.1.7). Such is the mystery of coherence towers: sometimes we can find a finite definition that makes all higher coherence issues vanish in one go, and sometimes this definition remains elusive.

[HoTT]: Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

Anyway it is straightforward to define presheaves of hsets, and they can be used to develop large swaths of category theory without fearing coherence issues [62]. However these proofs are less general than what we could hope for if we had a working definition for presheaves of spaces, because they only apply to the subclass of presheaves where all types are hsets.

[62]: Bauer et al. (2017), "The HoTT Library: A Formalization of Homotopy Type Theory in Coq"

Uniqueness of Identity Proofs (UIP) Going further in that direction, we can modify the type theory itself to enforce the uniqueness of identity proofs, so that every type is a hset. For instance, in the observational type theory $\mathrm{CC}^{\mathrm{obs}}$ from chapter 3 identity types are strict propositions, which means all identity proofs are definitionally equal.

$\mathrm{CC}^{\mathrm{obs}}$ gets very close to a type-theoretic account of constructive set theory, so we can reasonably expect that doing mathematics with presheaves will not pose significant problems.

Two-level type theories Instead of curtailing the identity types, two-level type theories feature two distinct propositional equalities: a homotopical equality that encodes paths, as well as a strict equality that encodes the actual equality of points and satisfies UIP [63, 64].

[63]: Voevodsky (2013), "A simple type system with two identity types"
[64]: Annenkov et al. (2017), "Two-Level Type Theory and Applications"

In such a framework, we can write a definition for presheaves using the strict equality and avoid coherence problems, while still supporting types with arbitrarily complex homotopical structures—at the price of additional complexity and manipulations of *fibrancy conditions*.

6.3 Prefascist Types

All the definitions of presheaves that we just listed follow the same general pattern: they keep the shape of the set-theory-inspired definition, and then they find a way to force the propositional equalities involved in the definition to behave like strict equalities, *i.e.* to prevent them from introducing relevant path information.

However, this is not the only possible approach: in a 2020 paper [32], Pédrot designs a syntactical model of MLTT in MLTT + sProp such that every type is interpreted as a presheaf which is functorial up to *definitional equality*. In order to do so, Pédrot introduces prefascist types, an original take on the definition of a presheaf.

[32]: Pédrot (2020), "Russian Constructivism in a Prefascist Theory"

Prefascist types owe their eyebrow-raising name to the French word for presheaf, *préfaisceau*, and to the humor of Pierre-Marie Pédrot.

In this section, we explain the definition of prefascist types and show some of their properties. As with section 6.2, we assume that we are working with a *strict* category, that is a category $\mathscr{C}$ that satisfies the associativity and unit laws definitionally. Moreover, we will extend MLTT with a sort of strict propositions Prop, as implemented by Coq, Agda and Lean [6]. It works just like the universe Set, except that any type A in Prop is proof-irrelevant, meaning that all terms with type A are convertible. We also add a strict truncation operator $\|\_\|_s$ that reflects proof-relevant types in Prop.

In chapter 2 we used sProp for the sort of strict propositions, and we reserved Prop for the sort of non-strict propositions of Coq. But since we adopt Agda's conventions in this chapter, we will use Prop for strict propositions.

As an illustration for the use of Prop, we can use it to define the category Δ_+ of finite ordinals and increasing maps as a strict category. If we define the monotonicity predicate as a strict proposition, then morphisms of Δ_+ are a pair of a function and a computationally irrelevant monotonicity proof, so that composition reduces to regular function composition, which is definitionally associative. In Subsection 6.3.3, we will explain that we can handle all small categories in this way.

6.3.1 What are Prefascist Types?

In order to arrive at a definition of prefascist types, we start by investigating strict presheaves of types, *i.e.* presheaves of types that satisfy the functoriality equations definitionally. Even though we do not have an internal definition for their type, we can still assume we have an object F that satisfies the functoriality equations up to definitional equality:

```
postulate
  F₀ : Obj → Set
  F₁ : {a b : Obj} → Hom a b → F₀ b → F₀ a
  F_id : (a : Obj) → F₁ (id a) = (λ x → x)
  F_comp : {a b c : Obj} (f : Hom a b) (g : Hom b c)
    → F₁ (g ∘ f) = (λ x → F₁ f (F₁ g x))
```

Once again, it is impossible to use the definitional equality in Agda syntax. This should really be read as assuming actual implementations of F_0 and F_1 that satisfy the two equations F_id and F_comp by definition.

Although we cannot define it internally in MLTT, we will write $\mathrm{sPsh}(\mathscr{C})$ for the category of strict type-theoretic presheaves and strict natural transformations. An inhabitant of $\mathrm{sPsh}(\mathscr{C})$ is given by two terms F_0 and F_1 of MLTT that satisfy the equations F_id and F_comp definitionally, and a morphism is defined in a similar way. Note that our category $\mathrm{sPsh}(\mathscr{C})$ of type-theoretic presheaves only contains *internal presheaves*: we do not consider arbitrary set-theoretic functors from $\mathscr{C}$ to the category of types, but only functors that can be expressed in MLTT.

Now let us assume that we are working with specific instances of F_0 and F_1. We remark that given x in $\mathsf{F}_0\ a$, we can use F_1 to get an inhabitant of $\mathsf{F}_0\ b$ whenever we have a morphism from b to a. In other words, having the presheaf structure laying around means that elements of $\mathsf{F}_0\ a$ are really promoted to elements of the "completed" type family $\widehat{\mathsf{F}_0}$, which is defined as follows:

$$\begin{array}{lcl} \widehat{\mathsf{F}_0} & : & (a : \mathsf{Obj}) \to \mathsf{Set} \\ \widehat{\mathsf{F}_0}\ a & := & \forall\ b\ (f : \mathsf{Hom}\ b\ a) \longrightarrow \mathsf{F}_0\ b \end{array}$$

And the promoted version of $x : \mathsf{F}_0\ a$ is given by

$$\hat{x} := \lambda\ b\ f \longrightarrow \mathsf{F}_1\ f\ x.$$

Tracking all restrictions like this allows us to exhibit a natural presheaf structure for $\widehat{\mathsf{F}_0}$ by defining functoriality with a simple composition. If f is a morphism from b to a, we define

$$\begin{array}{lcl} f \cdot \_ & : & \widehat{\mathsf{F}_0}\ a \longrightarrow \widehat{\mathsf{F}_0}\ b \\ f \cdot \hat{x} & := & \lambda\ c\ g \longrightarrow \hat{x}\ c\ (f \circ g). \end{array}$$

Now, remember how we insisted that $\mathscr{C}$ satisfies associativity and the unit law up to definitional equality. This constraint on $\mathscr{C}$ automatically makes $\widehat{\mathsf{F}_0}$ a strict presheaf:

$$\begin{aligned} g \cdot (f \cdot \hat{x}) &= (f \circ g) \cdot \hat{x} \\ (\mathsf{id}\ a) \cdot \hat{x} &= \hat{x}. \end{aligned}$$

Thus, $\widehat{\mathsf{F}_0}$ can be seen as an object of $\mathrm{sPsh}(\mathscr{C})$, and since F_1 satisfies the functoriality equations up to a definitional equality, our embedding $x \mapsto \hat{x}$ commutes with restriction: $f \cdot \hat{x}$ is convertible to the embedding of $\mathsf{F}_1\ f\ x$. This means that we have a *strict* natural tranformation $F \to \widehat{\mathsf{F}_0}$, which

effectively replaces the functoriality given by F_1 with a functoriality operation derived from composition in $\mathscr{C}$.

The strict presheaves that can be expressed as $\widehat{X}$ for some $X : \mathrm{Obj} \to \mathrm{Set}$ (that we will call cofree presheaves) are remarkably easy to handle, since their functoriality operation is automatically provided by composition and will satisfy the equations up to a definitional equality. And as we just explained, every strict presheaf can be embedded into a cofree presheaf. This leads us to the definition of a *prefascist type* as a sub-presheaf of a cofree presheaf:

In the next section, we present presheaves as coalgebras, and presheaves of the form $\widehat{X}$ will be cofree coalgebras.

```
record prefascist : Set₁ where
  field
    F₀ : (a : Obj) → Set
    Fε : (a : Obj) → (∀ b (f : Hom b a) → F₀ b) → Prop
```

The record that defines a prefascist type contains a type family F_0 on the objects of $\mathscr{C}$, and a proof-irrelevant predicate on the "completed" family $\widehat{F_0}$. This definition is intended to represent the subpresheaf of $\widehat{F_0}$ that is obtained by only considering the elements of $\widehat{F_0}\ a$ that satisfy the proof-irrelevant predicate $F\varepsilon$ for all their restrictions.

Given a prefascist type F, we can define the type of elements of F over an object a as follows:

```
record elem (F : prefascist) (a : Obj) : Set where
  open prefascist F
  field
    x₀ : ∀ b (f : Hom b a) → F₀ b
    xε : ∀ b (f : Hom b a) → Fε b (f · x₀)
```

This record contains an element x_0 of $\widehat{F_0}\ a$ and a proof $x\varepsilon$ that all its restrictions satisfy the predicate $F\varepsilon$. We can define a functoriality operation on the elements of a prefascist type, by using composition on x_0 and $x\varepsilon$. If f is a morphism from b to a, we define

```
f · _          :  elem F a → elem F b
f · ⟨ x₀ , xε ⟩  :=  ⟨ f · x₀ , f · xε ⟩
```

And since composition in $\mathscr{C}$ is strictly associative, the prefascist types are strict presheaves. Finally, morphisms of prefascist types are defined as follows:

```
record pf_hom (F G : prefascist) : Set where
  open prefascist
  field
    θ₀ : (a : Obj)
         → (x0 : ∀ b (f : Hom b a) → F₀ F b)
         → (xε : ∀ b (f : Hom b a) → Fε F b (f · x0))
         → F₀ G a
    θε : (a : Obj)
         → (x0 : ∀ b (f : Hom b a) → F₀ F b)
         → (xε : ∀ b (f : Hom b a) → Fε F b (f · x0))
         → Fε G a (λ b f → θ₀ b (f · x0) (λ c g → xε c (f ∘ g)))
```

This definition might look daunting at first, but it simply imitates the definition of an element of G with an element of F in the context: "morphisms are generalized elements". We invite the reader to check that morphisms are in fact strict natural transformations (meaning that the naturality holds definitionally).

Thus, our prefascist types form a remarkable subcategory of $\mathrm{sPsh}(\mathscr{C})$ that can be defined in a minor extension of MLTT. Naturally, this begs the question of which presheaves can be encoded as prefascist sets. In the next section, we will see that prefascist types are equivalent to the full category of presheaves in a set-truncated theory, but unfortunately falls short of it when homotopical information becomes relevant.

6.3.2 Categorical Perspective

In this section, we revisit prefascist types on $\mathscr{C}$ from the perspective of category theory. We assume that the reader is familiar with the theory of monads and adjunctions.

While the problem of defining the category of presheaves of types on $\mathscr{C}$ in type theory is rather delicate, there are particular cases in which this problem becomes much simpler. In particular, if the category is discrete, then presheaves of types on $\mathscr{C}$ are just types indexed by $\mathrm{Obj}(\mathscr{C})$ and morphisms are indexed functions. This category is simple enough to define in intensional type theory, and all of the equations are (vacuously) verified up to definitional equality.

A category is discrete if it has only identity morphisms.

Now, remark that every category $\mathscr{C}$ contains a discrete subcategory $\mathscr{C}_0$, which has the same objects as $\mathscr{C}$ but only identity morphisms. The embedding $\mathscr{C}_0 \hookrightarrow \mathscr{C}$ gives rise to a "pullback" functor

$$\imath^* : \mathrm{sPsh}(\mathscr{C}) \to \mathrm{sPsh}(\mathscr{C}_0)$$

which simply forgets the functoriality of a strict presheaf. This functor has a right adjoint $\imath_*$, given by

$\imath^*$ also has a left adjoint, thus both $\imath^*$ and $\imath_*$ preserve limits.

$$\begin{array}{lcl} \imath_* & : & \mathrm{sPsh}(\mathscr{C}_0) \to \mathrm{sPsh}(\mathscr{C}) \\ \imath_* X(a) & := & \forall\ (b : \mathrm{Obj})\ (f : \mathrm{Hom}\ b\ a) \longrightarrow X\ b \end{array}$$

which is the same as our construction $\widehat{X}$ from the previous section.

The presheaf comonad By composing both sides of the adjunction, we get a comonad $\imath^*\imath_*$ on $\mathrm{sPsh}(\mathscr{C}_0)$, which we can define internally in MLTT. The comultiplication of $\imath^*\imath_*$ corresponds to composition of morphisms in $\mathscr{C}$:

$$\begin{array}{lcl} \nu_X & : & \forall\ (a : \mathrm{Obj}) \to \imath^*\imath_* X(a) \to \imath^*\imath_*\imath^*\imath_* X(a) \\ \nu_X(a) & := & \lambda\ (x : \imath^*\imath_* X(a))\ (b : \mathrm{Obj})\ (f : \mathrm{Hom}\ b\ a) \\ & & \qquad (c : \mathrm{Obj})\ (g : \mathrm{Hom}\ c\ b)\ .\ x\ c\ (f \circ g) \end{array}$$

And its counit is derived from the identity morphism of $\mathscr{C}$:

$$\begin{array}{lcl} \varepsilon_X & : & \forall\ (a : \mathrm{Obj}) \to \imath^*\imath_* X(a) \to X(a) \\ \varepsilon_X(a) & := & \lambda\ (x : \imath^*\imath_* X(a))\ .\ x\ a\ (\mathrm{id}\ a). \end{array}$$

Since $\mathscr{C}$ is a strict category, it follows that $\imath^*\imath_*$ satisfies the comonad equations definitionally: the associativity of the comultiplication corresponds to associativity of the composition in $\mathscr{C}$ and the unit laws for the comultiplication correspond to the unit laws for $\mathscr{C}$.

The comonad $\imath^*\imath_*$ is particularly interesting to us, because it has the category of presheaves as its category of coalgebras.

Theorem 6.3.1 *The category of strict type-theoretic presheaves* $\mathrm{sPsh}(\mathscr{C})$ *is equivalent to the category of strict type-theoretic coalgebras of the comonad* $\imath^*\imath_*$*. In other words,* $\imath^* \dashv \imath_*$ *is a comonadic adjunction.*

Proof. A strict coalgebra for $\imath^*\imath_*$ is by definition a collection of types F indexed by the objects of $\mathscr{C}$, along with a collection of functions

$$F(a) \to \forall\, b\ (f : \mathsf{Hom}\ b\ a) \ \to\ F(b)$$

that satisfy the two coalgebra equations definitionally, which are respectively the compatibility with composition and identities. □

This decomposition of the theory of presheaves into the adjunction $\imath^* \dashv \imath_*$ is already implicitly present in the work of Jaber *et al.* [65] where they analyze presheaves under the lens of call-by-push-value.

[65]: Jaber et al. (2016), "The Definitional Side of the Forcing"

Cofree presheaves As we already saw, defining the category of coalgebras of $\imath^*\imath_*$ (*i.e.* the category of presheaves of $\mathscr{C}$) in MLTT is problematic because of the coalgebra equations. However there is an important subcategory that we can define in type theory: the co-Kleisli category of the adjunction, which is equivalent to the category of cofree coalgebras. The objects of the co-Kleisli category are presheaves on $\mathscr{C}_0$, and the hom-types are given by

$$\mathrm{Hom}_{\mathrm{coK}}(F, G) := \forall\, a\ \to \imath^*\imath_* F(a) \to G(a).$$

The composition of two morphisms θ and ξ in the co-Kleisli category is obtained by computing $\theta \circ (\imath^*\imath_*\xi)$ in $\mathrm{sPsh}(\mathscr{C}_0)$ and composing it with the comultiplication. This composition is strictly associative, because all the laws of the comonad $\imath^*\imath_*$ hold definitionally.

The subcategory of cofree presheaves is convenient to define and manipulate in type theory, but it supports only a small portion of the structure of the category $\mathrm{Psh}(\mathscr{C})$. In general, the coproduct of two cofree presheaves is not a cofree presheaf.

Prefascist types In the previous section we noticed that any presheaf $X : \mathrm{Psh}(\mathscr{C})$ has a natural transformation that maps it into a cofree presheaf, given by the unit of the adjunction $\imath^* \dashv \imath_*$.

$$\eta_X : X \to \imath_*\imath^* X$$

This led us to define a prefascist type as a sub-presheaf of a cofree presheaf, *i.e.* a pair of a family of types $F_0 : \mathrm{Psh}(\mathscr{C}_0)$ and a propositional predicate on $\imath^*\imath_* F_0$ that is preserved by restriction.

In the the record prefascist of the previous section, we did not ask for $\mathsf{F}\varepsilon$ to be preserved by the restriction of $\widehat{\mathsf{F}_0}$. We made up for this in the definition of elem by asking that $\mathsf{F}\varepsilon$ applies to all the restrictions of the elements.

And thus we come back to our question: can every strict presheaf X be presented as a prefascist type? The following lemma states that it is indeed the case if we are working with an observational type theory such as the theory $\mathrm{CC}^{\mathrm{obs}}$ of chapter 3.

Theorem 6.3.2 *Assume that our type theory supports function extensionality and has a proof-irrelevant equality.*

Then the category of strict presheaves $\mathrm{sPsh}(\mathscr{C})$ *is equivalent to the category of prefascist types, whose objects are terms of type* prefascist *and whose morphisms are terms of type* pf_hom *quotiented by propositional equality.*

Proof. We construct an equivalence of categories by hand. Given a strict presheaf X, the unit η_X is a monomorphism from X to a cofree presheaf. Thus we can define a prefascist type as follows:

$$\begin{array}{lcl} \mathsf{F}_0 & := & \imath^* X \\ \mathsf{F}\varepsilon & := & \lambda\ (a : \mathrm{Obj})\ (x : \imath^* \imath_* \imath^* X(a))\ .\ x \equiv \eta_X\ (x\ (\mathrm{id}\ a)). \end{array}$$

Conversely, given a prefascist type Y : prefascist, we already know that it naturally has the structure of a strict presheaf, by defining

$$Y(a) := \mathrm{elem}\ Y\ a.$$

Now it only remains to check that this defines an equivalence of categories, which is straightforward since we have function extensionality and UIP. □

Thus, if we work in observational type theory, prefascist types are an interesting alternative to the naive definition of presheaves: we get an equivalent category, and we gained definitional functoriality laws in the process.

Unfortunately, this equivalence will not hold if we work without UIP and interpret types as spaces. In that case, the unit η_X of the adjunction $\imath^* \dashv \imath_*$ is *not* a monomorphism in general, and it is not true that every presheaf is a sub-presheaf of a cofree presheaf. In the ∞-groupoid model, the category of prefascist types sits strictly inbetween the category of presheaves of hsets and the category of presheaves of types. We have no clear characterization of which presheaves can be expressed as prefascist types.

It is still true that $\eta_X\ a\ x \equiv \eta_X\ a\ y$ implies $x \equiv y$, but this is not sufficient to define a monorphism in the absence of UIP.

6.3.3 Strictifying Categories

Up until now, we worked under the assumption that we have a definition of a category $\mathscr{C}$ that satisfies the associativity of composition and the unit laws up to definitional equality. This seems a bit artificial: most categories we can encounter in the wild do not have such a strong property, and what good are our prefascist types if we can only define them on a very limited subset of categories? However in this section we show that any small category whose morphism types are *hsets* can be presented as a strict category, given that we have access to a sort Prop of strict propositions.

Theorem 6.3.3 *Assume we are given a non-strict category $\mathscr{C}$ whose hom-types are set-truncated.*

Then we can define a strict category $\mathscr{C}'$ with a faithful functor from $\mathscr{C}$ to $\mathscr{C}'$, and given some mild additional assumptions we can prove that this functor is in fact an equivalence of categories.

Proof. A non-strict category consists of a type of objects Obj, a *hset* of morphisms for any two objects a and b, and a composition and identities with equations that hold only up to a propositional equality:

We do not risk any higher coherence issues by using propositional equality, since we assumed that the morphism types of $\mathscr{C}$ are set-truncated.

```
postulate
  Obj : Set
  Hom : Obj → Obj → hset
  id : (a : Obj) → Hom a a
  _∘_ : {a b c : Obj} (f : Hom b c) (g : Hom a b) → Hom a c
  ∘-id-left : {a b : Obj} (f : Hom a b) → (id b) ∘ f ≡ f
  ∘-id-right : {a b : Obj} (f : Hom a b) → f ∘ (id a) ≡ f
  ∘-assoc : {a b c d : Obj}
            (f : Hom c d) (g : Hom b c) (h : Hom a b)
            → f ∘ (g ∘ h) ≡ (f ∘ g) ∘ h
```

Informally, the idea is to use the Yoneda lemma to replace the hom-set Hom a b with the isomorphic hset $\mathrm{Hom}_{\mathrm{Psh}(\mathscr{C})}(よ(a), よ(b))$. This way, morphisms become natural transformations of presheaves, which can be written as dependent functions with a naturality condition in Prop. This trick allows us to define a strict category $\mathscr{C}'$ from $\mathscr{C}$ as follows:

```
Obj'                :  Set
Obj'                :=  Obj

Hom'                :  Obj' → Obj' → Set
Hom' a b            :=  Σ (θ : ∀ (c : Obj) → Hom c a → Hom c b) .
                          ∀ (c d : Obj) (f : Hom d c) (g : Hom c a)
                            → ‖ (θ c g) ∘ f ≡ θ d (g ∘ f) ‖ₛ

id'                 :  (a : Obj') → Hom' a a
id' a               :=  ⟨λ (c : Obj) (f : Hom c a) . f ; λ _ _ _ _ . refl ⟩

_∘'_                :  {a b c : Obj'} → Hom' b c → Hom' a b → Hom' a c
⟨θ_f ; h_f⟩ ∘' ⟨θ_g ; h_g⟩  :=  ⟨λ (d : Obj) (h : Hom d a) . θ_f d (θ_g d h) ; [...]⟩
```

With this definition, composition of two morphisms becomes a composition of functions with a manipulation of proof-irrelevant conditions, which is definitionally associative. Likewise, the unit laws correspond to composition with an identity function, which is definitionally unital. Therefore our replacement for $\mathscr{C}$ is a strict category.

It is easily checked that the Yoneda embedding is a faithful functor. But is $\mathscr{C}'$ actually equivalent to $\mathscr{C}$? It is the case if we interpret types as classical sets, or as ∞-groupoids. However, proving the equivalence in the internal language of type theory is not possible in general, because

we need enough structure to recover a proof of $a \equiv b$ from a proof of $\| a \equiv b \|_s$ —for instance, it could be the case if we are working in observational type theory. □

6.3.4 The Prefascist Translation

In the context of a theory with UIP, most of the structure from categories of presheaves can be replicated for prefascist types in type theory. For instance, the category of prefascist types has products, pullbacks, exponential object, coproducts, *etc.* In his prefascist paper [32], Pédrot uses prefascist types to define a presheaf model of MLTT as a *syntactic translation* [66].

[32]: Pédrot (2020), "Russian Constructivism in a Prefascist Theory"

[66]: Boulier et al. (2017), "The next 700 syntactical models of type theory"

A prototypical syntactic translation is given by two mappings $[\_]$ and $[\![\_]\!]$ from the syntax of MLTT to the syntax of MLTT such that

1. if the judgment $\Gamma \vdash t : A$ is derivable in MLTT, then the judgment $[\![\Gamma]\!]_{\mathsf{Con}} \vdash [t] : [\![A]\!]$ is derivable in MLTT,
2. if $\Gamma \vdash A \equiv B$ then $[\![\Gamma]\!]_{\mathsf{Con}} \vdash [\![A]\!] \equiv [\![B]\!]$,
3. if $\Gamma \vdash t \equiv u : A$ then $[\![\Gamma]\!]_{\mathsf{Con}} \vdash [t] \equiv [u] : [\![A]\!]$, and
4. the translation of the empty type $[\![\bot]\!]$ has no inhabitants in the empty context.

$[\![\_]\!]_{\mathsf{Con}}$ applies $[\![\_]\!]$ to all the types in a context.

The prefascist model of Pédrot is slightly more involved: in particular, the target theory is not MLTT but sMLTT, a theory that extends MLTT with a sort of strict propositions and supports a proof-irrelevant equality with large elimination *à la* Lean. With this additional structure in the target theory, the prefascist model interprets every type as a prefascist set, and every term of type A as a natural transformation from the interpretation of the context to the interpretation of A.

In other words, prefascist types make it possible to phrase the traditional presheaf models in the language of intensional type theory, in a way that preserves computation.

7 A Model in Cubical Presheaves

As we explained in Subsection 2.2.2, the central piece of Homotopy Type Theory is the univalence axiom, which identifies the type of equalities between two types A and B with the type of isomorphisms between A and B.

But postulating univalence as an axiom has the unfortunate consequence of breaking some of the nice computational properties of intensional type theory. This is all the more disappointing insofar as the univalence principle has constructive models, as evidenced by Bezem *et al.* who built a model of univalence by interpreting types as *fibrant cubical sets* [21].

[21]: Bezem et al. (2014), "A Model of Type Theory in Cubical Sets"

This conundrum was solved by the introduction of cubical type theories [27, 57], which provide an axiom-free univalent system by reifying some of the structure of the fibrant cubical sets as syntactical constructions. In this chapter, our aim is to give an alternative solution for computing with fibrant cubical sets, by presenting the cubical model of Cohen *et al.* [57] as a syntactic translation from MLTT+Univalence to a simpler type theory.

[27]: Angiuli et al. (2019), "Syntax and Models of Cartesian Cubical Type Theory"
[57]: Cohen et al. (2018), "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom"

Our strategy is based on the prefascist translation of Pédrot, which is a syntactic account of the interpretation of type theory in categories of presheaves. Since the cubical sets of Cohen *et al.* are presheaves on the *category of de Morgan cubes*, we can use the prefascist translation to obtain a model of MLTT in the category of cubical sets. However this is not quite sufficient to validate the univalence axiom. Indeed, the cubical model of Cohen *et al.* does not exactly follow the generic recipe to interpret MLTT in categories of presheaves: in particular, the equality is interpreted as the set of *cubical paths*, a construction that exploits the specific properties of the category of cubes. And in order to interpret the elimination principle for the equality, Cohen *et al.* equip their cubical sets with *fibration structures*.

In section 7.1, we explain what are cubical sets and fibration structures, and how they relate to spaces and topology. Then in section 7.2 we sketch an extension of the prefascist translation of Pédrot with fibration structures, which should give us enough structure to interpret the univalence axiom. Finally, in section 7.3 we discuss the theoretical consequences of this translation.

The results presented in this chapter have yet to reach the same degree of maturity as the rest of this thesis. We will be careful to distinguish the parts that have been thoroughly checked from the parts that are less solid.

7.1 Cubical Sets and Fibration Structures

Cubical sets were initially studied by topologists, as a candidate for a definition of "space" that is suitable to do *homotopy theory*, that is to study spaces and maps between spaces up to deformations.

When most mathematicians hear the word space, they tend to think of topological spaces, *i.e.* sets equipped with a lattice of open subsets. But while topological spaces may be very well designed for the needs

of mathematical analysis, it turns out that they are not quite as suitable for the study of deformations. We can try to explain this mismatch from the definition of a homotopy between two continuous maps, which gives a central role to the interval $[0, 1]$; but topological spaces are so general that they have no reason to interact nicely with $[0, 1]$, as witnessed by the inadequacy of path-connectedness for general topological spaces.

A homotopy between two continuous maps $f, g : X \to Y$ is a continuous map $h : X \times [0, 1] \to Y$ such that $f(x) = h(x, 0)$ and $g(x) = h(x, 1)$.

Thus, it should come as no surprise that homotopy theorists like to restrict their attention to a subclass of topological spaces that are more compatible with the unit interval. More specifically, it is common practice to only consider spaces that can be built from (hyper)cubes—cartesian powers of the interval—by gluing them together (figure 7.1). This way, we can be sure that they do not get too wild and the definition of homotopy will be reasonable.

Topological spaces that can be built by iteratively attaching cubical cells of increasing dimension are called CW-complexes. Since we only mention them as motivation, we will not need a precise definition.

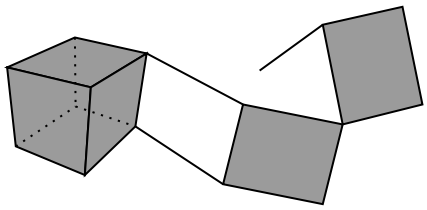


Figure 7.1: Building a space by gluing cubes of various dimensions together

Now, this idea of building objects by taking amalgamated sums of a family of basic "building block" objects should be reminiscent of the previous chapter about presheaves, and the reader might be wondering if we can use presheaves on an appropriate category of cubes to obtain a synthetic model of well-behaved spaces. This is precisely what cubical sets are.

7.1.1 The category of cubes

We now define the category of cubes $\square$. Since our building blocks are cubes of all integer dimensions (including a cube of dimension zero, which is a point), we will have one object per dimension:

By cubes, we mean generalized cubes of all integer dimensions: the point, the segment, the square, the 3D cube, the 4D hypercube... are all cubes.

$$\mathrm{Obj}(\square) := \mathbb{N}.$$

Thus, the objects of $\square$ are integers, that we shall mentally picture as cubes of the corresponding dimension:

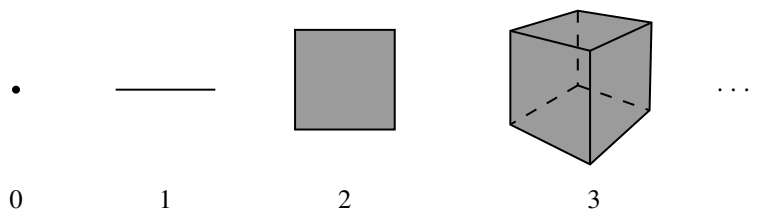


Figure 7.2: Graphical representation of the first four objects of $\square$

Then, the set of morphisms $\mathrm{Hom}_{\square}(m, n)$ between any two such objects should intuitively encode the different ways to send a cube of dimension m to a cube of dimension n. There are many reasonable choices for this definition (see Buchholtz et al. [67] for a survey), and each choice will result in a slightly different category of cubical sets. In this

[67]: Buchholtz et al. (2017), "Varieties of Cubical Sets"

The hope is that these different categories of cubical sets will capture the same notion of space, when considered up to homotopy. But such theorems are usually quite difficult to prove.

Operations of a de Morgan algebra $\mathbb{I}$:

$$\begin{array}{lll} i_0 : \mathbb{I} & \wedge : \mathbb{I} \times \mathbb{I} \to \mathbb{I} & \sim : \mathbb{I} \to \mathbb{I} \\ i_1 : \mathbb{I} & \vee : \mathbb{I} \times \mathbb{I} \to \mathbb{I} & \end{array}$$

Equations of de Morgan algebras:

$x \vee (y \vee z) = (x \vee y) \vee z$	$\vee$ – associativity
$i_0 \vee x = x = x \vee i_0$	$\vee$ – unit
$i_1 \vee x = i_1 = x \vee i_1$	$\vee$ – absorption
$x \vee y = y \vee x$	$\vee$ – symmetry
$x \vee x = x$	$\vee$ – idempotence
$\sim\sim x = x$	$\sim$ – involution
$\sim i_0 = i_1$	$\sim - i_0$
$\sim i_1 = i_0$	$\sim - i_1$
$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$	first distributive law
$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	second distributive law
$x = x \vee (x \wedge y) = x \wedge (x \vee y)$	lattice absorption
$\sim(x \wedge y) = \sim x \vee \sim y$	de Morgan's first law
$\sim(x \vee y) = \sim x \wedge \sim y$	de Morgan's second law

Figure 7.3: Definition of a De Morgan algebra

chapter, we follow Cohen et al. [57] and use the category of *de Morgan* cubes:

Definition 7.1.1 *The category of de Morgan cubes $\square$ is defined by*

$$\begin{aligned} \mathrm{Obj}(\square) &:= \mathbb{N} \\ \mathrm{Hom}_\square(m,n) &:= \mathrm{Vec}(\mathrm{dM}(x_1,...,x_m), n), \end{aligned}$$

meaning that the morphisms from m to n are lists of n terms of $\mathrm{dM}(x_1,...,x_m)$*, the free de Morgan algebra on m variables.*

The definition of a de Morgan algebra is presented in figure 7.3.

The identity of $\mathrm{Hom}_\square(n,n)$ is given by listing the variables in order:

$$\mathrm{Id}_n := \langle x_1, ..., x_n \rangle,$$

and given two morphisms $\langle i_1, ..., i_m \rangle \in \mathrm{Hom}_\square(l,m)$ and $\langle j_1, ..., j_n \rangle \in \mathrm{Hom}_\square(m,n)$, their composition is given by substitution:

$$\begin{aligned} &\langle j_1, ..., j_n \rangle \circ \langle i_1, ..., i_m \rangle := \\ &\quad \langle j_1[x_1 \leftarrow i_1, ..., x_m \leftarrow i_m], ..., j_n[x_1 \leftarrow i_1, ..., x_m \leftarrow i_m] \rangle. \end{aligned}$$

One readily checks that these definitions satisfy the axioms of a category. From there, we can define cubical sets as presheaves:

Definition 7.1.2 *The category of cubical sets is the category of presheaves on de Morgan cubes $\square$.*

Now, what is the connection between de Morgan algebras and cubes? First of all, since cubes are nothing but cartesian powers of the interval (the one-dimensional cube), we can expect that the hom-sets of $\square$ are entirely determined by the morphisms into the interval. And while we could have picked $\mathrm{Hom}_\square(n,1)$ to be the set of all real, continuous maps from $[0,1]^n$ to $[0,1]$, this would result in a cumbersome category

Recall that $\mathrm{Hom}(A, B \times C) \cong \mathrm{Hom}(A,B) \times \mathrm{Hom}(A,C)$.

(we could not expect to have decidable equality on the hom-sets, for instance) and it turns out to be unnecessary. Instead, it will be sufficient for our purposes to only consider a handful of primitive continuous maps into the real interval:

- cartesian projections $[0,1]^n \to [0,1]$,
- constant maps to one of the endpoints of $[0,1]$,
- $\min : [0,1]^2 \to [0,1]$ and $\max : [0,1]^2 \to [0,1]$, and
- $x \mapsto 1 - x : [0,1] \to [0,1]$.

And it happens that the hom-sets we obtain after closing this family of maps under cartesian products and composition are exactly the free de Morgan algebras [67].

[67]: Buchholtz et al. (2017), "Varieties of Cubical Sets"

This restriction seems a bit dramatic: we casually replaced the full set of continuous functions with a small set of piecewise linear functions! But the reader should keep in mind that our goal is to model spaces *up to homotopy*, and we can always deform continuous maps between reasonable spaces into piecewise linear maps between cubical spaces [68].

[68]: Hatcher (2002), *Algebraic Topology*

7.1.2 Unfolding definitions

Now that we defined the category of cubical sets as $\mathrm{Psh}(\square)$, we unfold the definition a little bit to get more intuition. Of course, cubical sets are *combinatorial* objects, meaning that there is no topology involved in their definition, but only sets and maps.

A cubical set X is a functor from $\square^{\mathrm{op}}$ to Set, meaning that we have a set $X(n)$ for every integer n, representing the set of cubes of dimension n of our cubical set. Then, the functoriality condition tells us that any map $f : \mathrm{Hom}_{\square}(m,n)$ in the cube category is sent to a map between sets $X(f) : X(n) \to X(m)$, in a way that respects identities and composition. We now look at a specific family of basic morphisms that generate all morphisms of $\square$, and we analyze their role in cubical sets.

Exchanges and reversals The $\sim$ operation of de Morgan algebras gives rise to reversal maps in the cube category, that mirror n-dimensional along the k-th dimension for all $k \leq n$:

$$\mathrm{rev}^n_k := \langle x_1, \dots, \sim x_k, \dots, x_n \rangle : \mathrm{Hom}_{\square}(n,n)$$

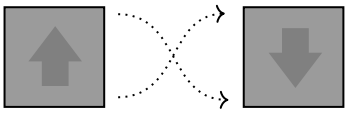


Figure 7.4: Reversal map

In our cubical set, $X(\mathrm{rev}^n_k)$ is thus an involution of the set of cubes of dimension n, that sends every cube to its "mirror image" (which is simply another element of $X(n)$). Note that the mirror cube will be different from the original, meaning that the cubes that constitute cubical sets are in fact *oriented cubes*.

Likewise, since the objects in our cube category are cartesian powers of the interval, we have maps that exchange two dimensions $j \leq k$:

$$\mathrm{exch}^n_{j,k} := \langle x_1, \dots, x_k, \dots, x_j, \dots, x_n \rangle : \mathrm{Hom}_{\square}(n,n)$$

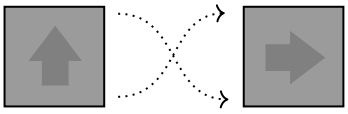


Figure 7.5: Exchange map

These maps also translate to symmetries of the set $X(n)$, that send a cube to the cube that represents its mirror image along a diagonal hyperplane.

Faces and diagonals De Morgan algebras have two constants i_0 and i_1, representing the endpoints of the interval. They give rise to *face maps*, that insert a constant in k-th position for all $k \leq n$:

$$\text{face}^n_{k,0} := \langle x_1, \dots, x_{k-1}, i_0, x_k \dots, x_n \rangle : \text{Hom}_\square(n, n+1)$$
$$\text{face}^n_{k,1} := \langle x_1, \dots, x_{k-1}, i_1, x_k \dots, x_n \rangle : \text{Hom}_\square(n, n+1)$$

In the cubical set, $X(\text{face}^n_{k,0})$ and $X(\text{face}^n_{k,1})$ send a $(n+1)$-dimensional cube to its "hyperfaces", which are cubes of dimension n. Of course, by composing face maps in several dimensions, one can obtain faces of all dimensions $k < n+1$.

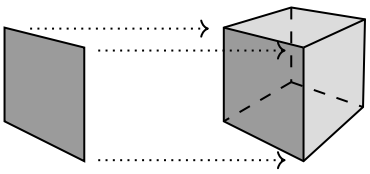


Figure 7.6: Face map

The cartesian structure of the cube category additionally gives rise to another similar family of maps from n to $n+1$, namely the diagonal maps obtained by duplicating the k-th dimension:

$$\text{diag}^n_k := \langle x_1, \dots, x_k, x_k, \dots, x_n \rangle : \text{Hom}_\square(n, n+1)$$

One should think of these maps as embedding a n-dimensional cube into the k-th main diagonal of a $(n+1)$-dimensional cube. We can get all the other diagonals by composing these maps with exchanges and reversals. In a cubical set, the diagonal maps send cubes of dimension $n+1$ to lower dimensional cubes, which should be pictured as their diagonals.

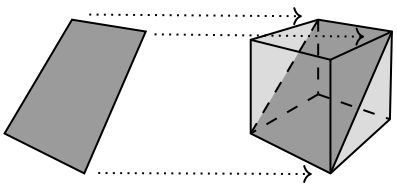


Figure 7.7: Diagonal map

Degeneracies and connections If the cartesian structure of the cubes gave rise to *duplication* maps, it also gives rise to *erasure* maps that delete the k-th coordinate:

$$\text{degen}^n_k := \langle x_1, \dots, x_{k-1}, x_{k+1}, \dots, x_n \rangle : \text{Hom}_\square(n, n-1)$$

Such maps are called degenaracies, and should be pictured as squashing the n-dimensional cube along the k-th dimension. Having these maps in our cube category means that in a cubical set, a cube of dimension n comes with squashed cubes of all dimensions attached to it. This might appear strange at first glance, but one quickly realizes that it is necessary: without degenerate cubes, it would be impossible to map the interval to a point, since morphisms of cubical sets are natural transformations, which must send 1-dimensional cubes to 1-dimensional cubes. With the degenerate cubes, it becomes possible: the 1-dimensional edge of the interval is mapped to the degenerate edge contained in the 0-dimensional point.

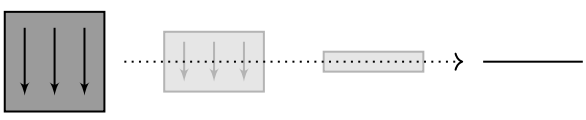


Figure 7.8: Graphical representation of the degeneracies

Finally, the last family of maps come from the de Morgan binary operations $\vee$ and $\wedge$, which correspond to the min and max operations on

the real interval:

$$\begin{aligned}\text{join}^n_k &:= \langle x_1, \dots, x_k \vee x_{k+1}, \dots, x_n \rangle : \text{Hom}_\square(n, n-1) \\ \text{meet}^n_k &:= \langle x_1, \dots, x_k \wedge x_{k+1}, \dots, x_n \rangle : \text{Hom}_\square(n, n-1)\end{aligned}$$

These maps are called *connections*, and we can picture them as contracting a square onto its main diagonal, as if it were a folding fan (figure 7.9). In cubical sets, connections equip cubes with two new families of degenerate cubes, allowing for more maps between cubes. For now, connections may not seem as motivated as the other operations, but they will play an important part in the next section, where we equip cubical sets with *fibration structures*.

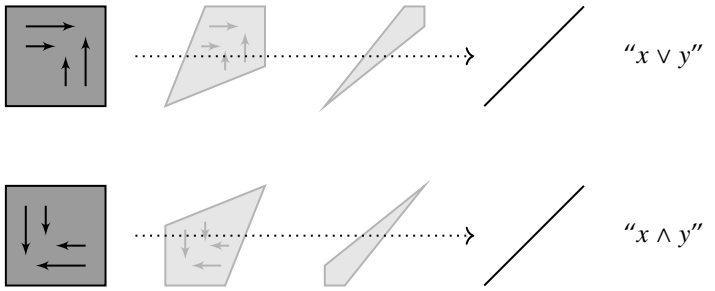


Figure 7.9: Graphical representation of the connections

As we can see, the functoriality condition translates this generating family of $\square$ into a family of operations on cubical sets. Naturally, the equalities between morphisms in $\square$ will translate to *equations* on these operations: reversals are idempotent, faces of degenerate cubes are themselves degenerate cubes, etc. All in all, this presheaf construction amounts to giving a multi-sorted, algebraic presentation of cubical sets.

7.1.3 Fibrant Cubical Sets

With cubical sets, we have a solid candidate for a combinatorial framework to study spaces and deformations: cubical sets come equipped with notions of points, of paths, of 2-dimensional homotopies, etc. However, we are mising a crucial ingredient: the possibility to *compose* paths and higher homotopies. Indeed, in a topological space X, given two paths $p_1, p_2 : [0,1] \to X$ with matching endpoints, we can concatenate them as follows:

$$\begin{aligned}p_1 \cdot p_2 \quad &: \quad [0,1] \to X \\ p_1 \cdot p_2(x) \quad &:= \quad \begin{cases} p_1(2x) & x \leq 1/2 \\ p_2(2x-1) & x > 1/2 \end{cases}.\end{aligned}$$

And likewise with higher-dimensional homotopies. However, in a cubical set X, there is seemingly no way to compose two paths from $X(1)$ that have matching boundaries. This is no small issue: without composition, we cannot even define the fundamental group, a corner stone of homotopy theory!

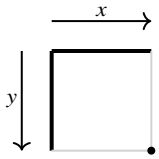
We will remediate to this issue by equipping our cubical sets with *fibration structures*, which are simply rules to compose cubes of different dimension together. But before defining fibration structures, we need to work a bit more to define what is a set of composable cubes.

Definition 7.1.3 *The face lattice of dimension n is defined by*

$$\mathbb{F}(n) := \mathrm{Hom}_{\square}(n, 1)$$

or in other words, an element of $\mathbb{F}(n)$ *is a term of* $\mathrm{dM}(x_1, ..., x_n)$.

As the name suggests, morphisms of $\mathrm{Hom}_{\square}(n, 1)$ can be used to represent collections of faces of a cube of dimension n. The idea is simple: once put in disjunctive normal form, a term of $\mathrm{dM}(x_1, ..., x_n)$ can be described as a join of meets of variables and negated variables. But if we replace x_i with $x_i = 1$ and $\sim x_k$ with $x_k = 0$, we can read that as a union of intersections of hyperfaces, as in figure 7.10.



The element $(\sim x) \vee (\sim y) \vee (x \wedge y)$ contains the faces $x = 0$, $y = 0$ and $(x = 1) \cap (y = 1)$.

Figure 7.10: An element of the face lattice

Next, we associate an *open box* to every element of the face lattice:

Definition 7.1.4 *Given* $f \in \mathbb{F}(n)$*, we build the open box* $\sqcap^n_f$ *as a subobject of the* $(n + 1)$*-dimensional cube* $\text{卐}(n + 1)$*:*

$$\begin{aligned}\sqcap^n_f(k) =& \{x \in \mathrm{Hom}_{\square}(k, n + 1) \mid f \circ \mathrm{degen}^{n+1}_{n+1} \circ x = i_1\} \\ &\cup \{x \in \mathrm{Hom}_{\square}(k, n + 1) \mid \pi_{n+1}\, x = i_0\}.\end{aligned}$$

In other words, we keep only the cubes whose first n coordinates belong to f and those whose last coordinate is i_0.

Open boxes represent the shapes that can be composed together. Note that the shape of two paths with matching endpoints is a special case of an open box: we can simply take $\sqcap^1_{\langle x \rangle}$.

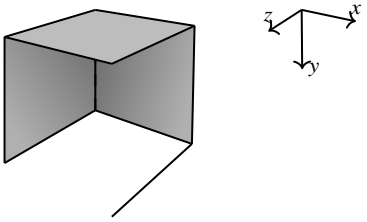


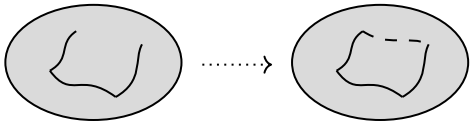
Figure 7.11: The open box corresponding to $(\sim x) \vee (\sim y) \vee (x \wedge y)$ is a subobject of the three-dimensional cube

Given an open box $\sqcap^n_f$ and a cubical set X, we define an open box embedded in X to be a homomorphism of cubical sets $\theta : \sqcap^n_f \to X$. Then, given a morphism $\alpha : \mathrm{Hom}_{\square}(k, n)$, we can extend the restriction operation $X(\alpha)$ to the open boxes embedded in X by applying α to the first n coordinates of the open box.

Definition 7.1.5 *A CCHM fibration structure for a cubical set X is a function* Φ *that takes as inputs an open box* $\theta : \sqcap^n_f \to X$ *and returns a "lid"* $\Phi(\theta) : \text{卐}(n) \to X$ *such that*

- *the lid matches the top of the open box, i.e.* $\Phi(\theta) \circ \mathrm{face}^{\mathrm{n}}_{\mathrm{n},1}$ *and* θ *coincide when they are both defined.*
- *given any morphism* $\alpha : \mathrm{Hom}_{\square}(k, n)$*, the result of applying* Φ *to the sub-open-box of* θ $f \circ \alpha$ *is equal to the result of applying* Φ *to the entire open box and restricting the result along* α.

The map $\theta : \sqcap^n_f \to X$ embeds an open box in X, that is a set of composable cubes, and the lid of the closed box $\Phi(\theta)$ represents the result of the composition.



A cubical set equipped with a CCHM fibration structure is said to be *CCHM-fibrant* (or just fibrant for short).

There are many possible choices for what it means to "compose" cubes, which translate to various fibration structures. In this thesis, we follow the CCHM paper [57], so we will only care about CCHM-fibrant cubical sets.

7.1.4 Taking a Step Back

With fibrant cubical sets, we have finally arrived at what seems to be a solid combinatorial framework for homotopy theory: on the one hand, the category of de Morgan cubes and the fibration structures are elementary, constructive and amenable to computational methods; and on the other hand, fibrant cubical sets experimentally seem to behave just like "well-behaved" topological spaces when considered up to deformation. As we will see in chapter 8, it is actually possible to define plenty of homotopical constructions such as fundamental groups, homotopy pushouts, spheres, suspensions, or even the Hopf fibration.

While this is comforting, it would be more satisfying to have a mathematical result that connects the category of topological spaces and the category of cubical sets "up to deformation". The adequate framework to formalize and prove such theorems is *higher category theory*. Unfortunately, to the best of our knowledge, the problem of whether CCHM-fibrant cubical sets and well-behaved topological spaces are equivalent higher categories is still open as of today.

It is however known that the geometric realization of CCHM-fibrant cubical sets is not a left Quillen adjunct [69]. But Quillen adjunctions are less general than higher equivalences.

The reader who wishes to learn more about combinatorial methods in homotopy theory is invited to consult Friedman [70], which treats the case of *simplicial* sets (which are built out of simplices instead of cubes) in a similarly elementary manner, in greater detail.

[70]: Friedman (2008), "An elementary illustrated introduction to simplicial sets"

7.2 Toward a Cubical Translation

We now turn ourselves to the task of translating MLTT+Univalence into a simpler type theory. Our strategy is to use the prefascist translation of Pédrot [32] to get an interpretation of MLTT in the category of cubical sets, and then to extend it with CCHM fibration structures.

The system MLTT+Univalence is defined to be Martin-Löf type theory extended with the univalence axiom, without the computation rule for the J eliminator on reflexivity.

The translation of Pédrot uses sMLTT as its target type theory, as it relies on proof-irrelevant propositions and a proof-irrelevant equality type with large elimination. For our part, we also want to have function extensionality in order to deal with fibration structures. This would make $\mathrm{CC}^{\mathrm{obs}}$ a natural choice for our target type theory, but we also need the J eliminator to compute on reflexivity—which means we actually need to use the extended system $\mathrm{CC}^{\mathrm{obs+}}$ (see section 5.1).

sMLTT extends MLTT with strict propositions, and an inductive equality valued in strict propositions *à la* Lean.

As we will realize shortly, the definition of a fibration structure will involve complex constructions in the language of $\mathrm{CC}^{\mathrm{obs+}}$. To make sure that we are not overlooking some important type dependency or some missing computation rule, it seems reasonable to define our translation with the help of a proof assistant. We opt for Agda, since we can use its proof-irrelevant propositions and rewrite rules [1] to support the theory of $\mathrm{CC}^{\mathrm{obs+}}$. Thus all the Agda code in this section uses the theory of $\mathrm{CC}^{\mathrm{obs+}}$, which means that we have access to the observational equality $a \sim b$ and the cast operator.

[1]: Cockx et al. (2021), "The Taming of the Rew: A Type Theory with Computational Assumptions"

7.2.1 Defining the Category of Cubes in Type Theory

The first step is to define the category of de Morgan cubes as a *strict* category, *i.e.* as a category with a composition operation that satisfies the associativity law and the unit laws up to a definitional equality. We could define this category in $\mathrm{CC}^{\mathrm{obs}+}$ by using the quotient types to define de Morgan algebras, and then using the Yoneda embedding to strictify the resulting category as explained in Subsection 6.3.3.

However, we can devise a much simpler embedding of the category of de Morgan cubes in the category of types and functions: define the *diamond algebra* $\mathbb{D}$ as the de Morgan algebra with carrier set $\{i_0, i_l, i_r, i_1\}$ where the meet, join and reversal are defined by the four equations

$$i_l \vee i_r = i_1 \qquad i_l \wedge i_r = i_0 \qquad \sim i_l = i_l \qquad \sim i_r = i_r.$$

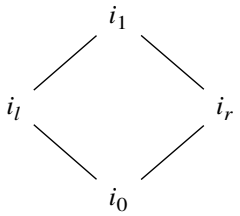


Figure 7.12: The diamond algebra $\mathbb{D}$

Then two morphisms $f, g \in \mathrm{Hom}_{\square}(m, n)$ in the category of de Morgan cubes are equal if and only if their interpretation as functions of type $\mathbb{D}^m \to \mathbb{D}^n$ are pointwise equal [71]. This implies that we can define the hom-sets of the category of cubes as subsets of function types, so that composition of morphisms coincides with function composition and the identity morphism coincides with the identity function.

[71]: Kauffman (1978), "De Morgan Algebras - completeness and recursion"

Consequently, we define an inductive type of codes for morphisms, using the generating family that we described in the previous section:

```
data dM : ℕ → ℕ → Set where
  identity : ∀ n → dM n n
  rev : ∀ k n m → k < m → dM n m → dM n m
  exch : ∀ j k n m → j < k → k < m → dM n m → dM n m
  face0 : ∀ k n m → k ≤ m → dM n m → dM n (suc m)
  face1 : ∀ k n m → k ≤ m → dM n m → dM n (suc m)
  diag : ∀ k n m → k < m → dM n m → dM n (suc m)
  degen : ∀ k n m → k ≤ m → dM n (suc m) → dM n m
  join : ∀ k n m → k ≤ m → dM n (suc (suc m)) → dM n (suc m)
  meet : ∀ k n m → k ≤ m → dM n (suc (suc m)) → dM n (suc m)
```

Then, we define the interpretation of a code as a function on the diamond algebra:

```
interpret : ∀ n m → dM n m → (diamond ^ n) → (diamond ^ m)
interpret = {- omitted -}
```

And finally, we define the type of morphisms $\mathrm{Hom}_{\square}(n, m)$ as the type of pairs of a diamond algebra function and a proof that it corresponds to the interpretation of a code.

```
Obj : Set
Obj = ℕ

record Hom (n : Obj) (m : Obj) : Set where
  field
    map : (diamond ^ n) → (diamond ^ m)
    code : ∃ (dM n m) (λ c → map ~ interpret n m c)
```

Note that the field `code` is proof-irrelevant. If our records satisfy the η-rule, this implies that morphisms behave exactly like functions and thus that they are definitionally associative.

7.2.2 The Universe of Cubical Sets

Now that we have our strict category of de Morgan cubes, we can set the stage for the prefascist translation. Recall the definition of prefascist types from chapter 6:

```
record prefascist {ℓ : Level} : Set (lsuc ℓ) where
  field
    F₀ : (n : Obj) → Set ℓ
    Fε : (n : Obj) → (∀ m (α : Hom m n) → F₀ m) → Prop
```

If we want to use these to build a model of univalent type theory, we first need to define a handful of building blocks. We start with a hierarchy of prefascist types indexed by a level ℓ that plays the role of a universe hierarchy for the category of prefascist types.

From the work of Hofmann and Streicher on presheaf universes [72], we know that the type of sections of a universe over an object n should be the type of prefascist types on the slice category $\square_{/n}$:

[72]: Hofmann et al. (1997), "Lifting Grothendieck Universes"

The slice category over an object n is noted $\square_{/n}$. The objects of the slice category are the morphisms of $\square$ with codomain n, and a morphism of the slice category from $f : a \to n$ to $g : b \to n$ is a $h : a \to b$ such that $g \circ h = f$.

```
record cubicalType₀ (ℓ : Level) (n : Obj) : Set (lsuc ℓ) where
  constructor CT
  field
    CT₀ : ∀ m (α : Hom m n) → Set ℓ
    CTε : ∀ m (α : Hom m n)
        → (∀ m' (α' : Hom m' m) → CT₀ m' (α ∘ α'))
        → Prop
```

This type family will fill the role of the field F_0 for our universal prefascist. Note that it has a natural restriction operation: given $\alpha : \mathrm{Hom}\ m\ n$, we can define restriction as composition with α in both fields.

Recall that we defined the notation $\alpha \cdot x$ as a shorthand for $\lambda\ m'\ \alpha' \to x\ m'\ (\alpha \circ \alpha')$.

```
restr : (α : Hom m n) → (cubicalType₀ ℓ n) → (cubicalType₀ ℓ m)
restr α (CT ct0 ctε) = CT (α · ct0) (α · ctε)
```

Now we can use this restriction operator to devise a predicate cubicalTypeε on families of sections of cubicalType₀, that will be the field Fε of the universal prefascist type.

```
cubicalTypeε : ∀ {ℓ n} → (∀ m α → cubicalType₀ ℓ m) → Prop
cubicalTypeε {ℓ} {n} A0 = ∀ m α → A0 m α ~ restr α (A0 n (id n))
```

And thus we can define the universal prefascist type cubicalType ℓ.

```
cubicalType : ∀ ℓ → prefascist
cubicalType ℓ = { F₀ = cubicalType₀ ℓ ; Fε = cubicalTypeε }
```

This universe hierarchy allows us to model *dependent types* in the category of prefascist types: given a prefascist A, a dependent prefascist on A is a prefascist morphism from A to cubicalType. Furthermore, if we have a prefascist morphism $B \to A$ and a dependent prefascist type on A, we can get a dependent prefascist type on B with a simple morphism composition. This "substitution" operation is strictly associative, which is very important when it comes to building models of dependent type theory [59].

[59]: Hofmann (1997), "Syntax and semantics of dependent types"

7.2.3 Fibration Structures

However, the model of Cohen *et al.* does not only interpret types as arbitrary cubical presheaves, it also equips them with fibration structures. We want to equip our dependent prefascist types with analogous fibration structures, which means that we need to build a universe of *CCHM-fibrant* prefascist types.

Its type of sections is defined as follows:

```
record fibrantType₀ (n : Obj) : Set₁ where
  constructor FT
  field
    FT₀ : ∀ m ⟶ (α : Hom m n) ⟶ Set
    FTε : ∀ m ⟶ (α : Hom m n)
          ⟶ (∀ m′ (α′ : Hom m′ m) ⟶ FT₀ m′ (α ∘ α′))
          ⟶ Prop
    comp₀ : ∀ m ⟶ (α : Hom (suc m) n)
          ⟶ (φ : Hom m 1)
          ⟶ (p0 : ∀ m′ (α′ : Hom m′ (suc m))
                ⟶ (Hφ : is⊤ (φ ∘ degen ∘ α′))
                ⟶ FT₀ m′ (α ∘ α′))
          ⟶ (pε : ∀ m′ (α′ : Hom m′ (suc m))
                ⟶ (Hφ : is⊤ (φ ∘ degen ∘ α′))
                ⟶ FTε m′ (α ∘ α′) (λ m″ α″ ⟶ p0 m″ (α′ ∘ α″) (restr α″ Hφ)))
          ⟶ (s0 : ∀ m′ (α′ : Hom m′ m) ⟶ FT₀ m′ (α ∘ side0 ∘ α′))
          ⟶ (sε : ∀ m′ (α′ : Hom m′ m) ⟶ FTε m′ (α ∘ side0 ∘ α′) (α′ · s0))
          ⟶ (s~ : ∀ m′ (α′ : Hom m′ m)
                ⟶ (Hφ : is⊤ (φ ∘ α′))
                ⟶ s0 m′ α′ ~ p0 m′ (side0 ∘ α′) Hφ)
          ⟶ FT₀ m (α ∘ side1)
    compε : {- same hypotheses as comp0 -}
          ⟶ FTε m (α ∘ side1)
              (λ m′ α′ ⟶ comp₀ m′ (α ∘ (lift α′)) (φ ∘ α′)
                (λ m″ α″ Hφ ⟶ p0 m″ (lift α′ ∘ α″) Hφ)
                (λ m″ α″ Hφ ⟶ pε m″ (lift α′ ∘ α″) Hφ)
                (α′ · s0) (α′ · sε)
                (λ m″ α″ Hφ ⟶ s~ m″ (α′ ∘ α″) Hφ))
    comp~ : {- same hypotheses as comp0 -}
          ⟶ ∀ m′ (α′ : Hom m′ m) (Hφ : is⊤ (φ ∘ α′))
          ⟶ p0 m′ (side1 ∘ α′) Hφ ~
              (comp₀ m′ (α ∘ (lift α′)) (φ ∘ α′)
                (λ m″ α″ Hφ ⟶ p0 m″ (lift α′ ∘ α″) Hφ)
                (λ m″ α″ Hφ ⟶ pε m″ (lift α′ ∘ α″) Hφ)
                (α′ · s0) (α′ · sε)
                (λ m″ α″ Hφ ⟶ s~ m″ (α′ ∘ α″) Hφ))
```

Given a morphism $\alpha : \mathrm{Hom}_{\square}(n, 1)$ in the category of cubes, or equivalently an element of the lattice of faces of the cube of dimension n, the proposition is⊤ α is true when α is the maximal element of the face lattice. This proposition supports a restriction operation, that we write restr.

The morphisms side0, side1, degen correspond respectively to $\mathrm{face}^n_{0,0}$, $\mathrm{face}^n_{0,1}$ and degen^n_0 from the previous section.

Given a morphism $\alpha : \mathrm{Hom}_{\square}(m, n)$ in the category of cubes, the morphism lift $\alpha : \mathrm{Hom}_{\square}(m+1, n+1)$ is the identity on the first coordinate and is equal to α on the remaining coordinates.

Just like in the type cubicalType₀, the fields FT_0 and FTε define a prefascist type on the category $\square_{/n}$, or equivalently a prefascist indexed over $\text{よ}\, n$ by the Yoneda lemma.

The fields comp_0, compε and comp~ compute a lid for an arbitrary open box in the dependent prefascist, which corresponds to the (non-homogeneous) composition in CCHM cubical type theory. Their arguments are

- a morphism α that embeds a cube of dimension $m+1$ in $よ\ n$ and an element of the face lattice φ, which together define an open box $\theta : \sqcap^m_\varphi \to よ\ n$;
- an element of the dependent prefascist retricted to the sides the open box θ, given by $p0$ and $p\varepsilon$;
- an element of the dependent prefascist restricted to the bottom of the open box θ, given by $s0$ and $s\varepsilon$; and
- a proof that the elements on the sides and on the bottom coincide, given by $s\sim$.

From these, comp_0 computes an element of FT_0 over the lid of the open box. The field $\mathsf{comp}\varepsilon$ ensures that restricting a lid is the same as computing a lid for the restricted box, and $\mathsf{comp}\sim$ ensures that the lid coincides with the sides of the box when both are defined.

Just like the type $\mathsf{cubicalType}_0$, our type $\mathsf{fibrantType}_0$ admits a natural restriction operation derived from composition. From this restriction operation, we define the predicate $\mathsf{fibrantType}\varepsilon$ as follows

```
fibrantTypeε : {n : Obj} → (∀ m α → fibrantType₀ m) → Prop
fibrantTypeε {n} A0 = ∀ m α → A0 m α ~ restr α (A0 n (id n))
```

And finally, we can define the prefascist fibrantType.

```
fibrantType : prefascist
fibrantType = { F₀ = fibrantType₀ ; Fε = fibrantTypeε }
```

7.2.4 Building a Model as a Translation

We can extract a model of dependent type theory from the universe fibrantType by interpreting contexts as telescopes of dependent prefascist types, types as dependent prefascist types, and terms as sections.

We phrase this model as a syntactic translation, *i.e.* as a family of functions from the syntax of MLTT to the syntax of $\mathsf{CC}^{\mathsf{obs}+}$:

- To every context of MLTT terms Γ, we associate a family of contexts of $\mathsf{CC}^{\mathsf{obs}+}$ indexed by an object $n : \mathsf{Obj}$, that we write $[\![\Gamma]\!]^n$.
- To every term t of MLTT, we associate two families of terms $[t]^n_0$ and $[t]^n_\varepsilon$ indexed by an object $n : \mathsf{Obj}$.

The intuition behind these functions is that if A is a type of MLTT in context Γ, then the two following judgments should be derivable in $\mathsf{CC}^{\mathsf{obs}+}$:

$$[\![\Gamma]\!]^n \quad \vdash \quad [A]^n_0 \quad : \quad \forall\ m\ (\alpha : \mathsf{Hom}\ m\ n) \to \mathsf{fibrantType}_0\ m$$
$$[\![\Gamma]\!]^n \quad \vdash \quad [A]^n_\varepsilon \quad : \quad \forall\ m\ (\alpha : \mathsf{Hom}\ m\ n) \to \mathsf{fibrantType}\varepsilon\ (\alpha \cdot [A]^n_0)$$

Taken together, these judgments encode an element of fibrantType, or in other words a prefascist type that depends on $[\![\Gamma]\!]^n$.

If the judgment $\Gamma \vdash t : A$ is derivable in MLTT, $[t]^n_0$ and $[t]^n_\varepsilon$ should correspond to a section of the prefascist type associated with A. To define this type of sections, we introduce the following two functions:

$$
\begin{array}{lcl}
[x]^n_0 & := & x_0 \\
[x]^n_\varepsilon & := & x_\varepsilon \\
[\lambda\,(x : A)\,.\,t]^n_0 & := & \lambda\; m\; \alpha\; x_0\; x_\varepsilon\; .\; [t]^m_0\; x_0\; x_\varepsilon\; m\; (\mathsf{id}\; m) \\
[\lambda\,(x : A)\,.\,t]^n_\varepsilon & := & \lambda\; m\; \alpha\; x_0\; x_\varepsilon\; .\; [t]^m_\varepsilon\; x_0\; x_\varepsilon\; m\; (\mathsf{id}\; m) \\
[t\; u]^n_0 & := & \lambda\; m\; \alpha\; .\; [t]^n_0\; m\; \alpha\; (\alpha \cdot [u]^n_0)\; (\alpha \cdot [u]^n_\varepsilon) \\
[t\; u]^n_\varepsilon & := & \lambda\; m\; \alpha\; .\; [t]^n_\varepsilon\; m\; \alpha\; (\alpha \cdot [u]^n_0)\; (\alpha \cdot [u]^n_\varepsilon) \\
[A \to B]^n_0 & := & \mathsf{Arr}_0\; [A]^n_0\; [A]^n_\varepsilon\; [B]^n_0\; [B]^n_\varepsilon \\
[A \to B]^n_\varepsilon & := & \mathsf{Arr}\varepsilon\; [A]^n_0\; [A]^n_\varepsilon\; [B]^n_0\; [B]^n_\varepsilon \\
[\Pi\,(x : A)\,.\,B]^n_0 & := & \mathsf{Prod}_0\; [A]^n_0\; [A]^n_\varepsilon\; [B]^n_0\; [B]^n_\varepsilon \\
[\Pi\,(x : A)\,.\,B]^n_\varepsilon & := & \mathsf{Prod}\varepsilon\; [A]^n_0\; [A]^n_\varepsilon\; [B]^n_0\; [B]^n_\varepsilon \\
[\mathsf{Type}]^n_0 & := & \mathsf{fType}_0 \\
[\mathsf{Type}]^n_\varepsilon & := & \mathsf{fType}\varepsilon \\
\\
[\![\bullet]\!]^n & := & n : \mathsf{Obj} \\
[\![\Gamma,\; x : A]\!]^n & := & [\![\Gamma]\!]^n\;,\; x_0 : \mathsf{El}_0\; [A]^n_0\;,\; x_\varepsilon : \mathsf{El}\varepsilon\; [A]^n_0\; [A]^n_\varepsilon\; x_0
\end{array}
$$

Figure 7.13: Core of the fibrant prefascist translation

$$
\begin{array}{l}
\mathsf{El}_0 : \{n : \mathsf{Obj}\} \to (A0 : \forall\; m\; (\alpha : \mathsf{Hom}\; m\; n) \to \mathsf{fibrantType}_0\; m) \to \mathsf{Set} \\
\mathsf{El}_0\; \{n\}\; A0 = \forall\; m\; (\alpha : \mathsf{Hom}\; m\; n) \to \mathsf{FT}_0\; (A0\; m\; \alpha)\; m\; (\mathsf{id}\; m)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{El}\varepsilon : \{n : \mathsf{Obj}\} \to (A0 : \forall\; m\; (\alpha : \mathsf{Hom}\; m\; n) \to \mathsf{fibrantType}_0\; m) \\
\qquad\qquad \to (A\varepsilon : \forall\; m\; (\alpha : \mathsf{Hom}\; m\; n) \to \mathsf{fibrantType}\varepsilon\; (\alpha \cdot A0)) \\
\qquad\qquad \to (x0 : \mathsf{El}_0\; A0) \to \mathsf{Prop} \\
\mathsf{El}\varepsilon\; \{n\}\; A0\; A\varepsilon\; x0 = \forall\; m\; (\alpha : \mathsf{Hom}\; m\; n) \to \mathsf{FT}\varepsilon\; (A0\; m\; \alpha)\; m\; (\mathsf{id}\; m) \\
\quad (\lambda\; m'\; \alpha' \to \mathsf{cast}\; (\mathsf{FT}_0\; (A0\; m'\; (\alpha \circ \alpha'))\; m'\; (\mathsf{id}\; m')) \\
\qquad\qquad (\mathsf{FT}_0\; (A0\; m\; \alpha)\; m'\; \alpha') \\
\qquad\qquad (A\varepsilon\; m\; \alpha\; m'\; \alpha') \\
\qquad\qquad (x0\; m'\; (\alpha \circ \alpha')))
\end{array}
$$

And then, if the judgment $\Gamma \vdash t : A$ is derivable in MLTT, we have

$$
\begin{array}{lcll}
[\![\Gamma]\!]^n & \vdash & [t]^n_0 & : \quad \mathsf{El}_0\; [A]^n_0 \\
[\![\Gamma]\!]^n & \vdash & [t]^n_\varepsilon & : \quad \mathsf{El}\varepsilon\; [A]^n_0\; [A]^n_\varepsilon .
\end{array}
$$

The translation for a core type theory with function types, dependent products and one universe is given in figure 7.13. Ideally the universe should be translated as a fibrant prefascist type, but the definition of the fibration structure for the universe relies on *Glue types* which are left for future work. For now, we interpret the universe a non-fibrant prefascist type instead:

Since fType_0 is not a fibrant prefascist type, the application $\mathsf{El}_0\; \mathsf{fType}_0$ is technically ill-typed. To avoid wasting too much of our attention on this kind of annoyances, we will silently extend El_0 and $\mathsf{El}\varepsilon$ to non-fibrant prefascist types.

$$
\begin{array}{l}
\mathsf{fType}_0 : \forall\; \{\; n\; \}\; m \to (\alpha : \mathsf{Hom}\; m\; n) \to \mathsf{cubicalType}_0\; \_\; m \\
\mathsf{fType}_0\; \{\; n\; \}\; m\; \alpha = \mathsf{CT}\; (\lambda\; m'\; \alpha' \to \mathsf{fibrantType}_0\; m') \\
\qquad\qquad\qquad\qquad (\lambda\; m'\; \alpha'\; x \to \mathsf{fibrantType}\varepsilon\; x)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fType}\varepsilon : \forall\; \{\; n\; \}\; m \to (\alpha : \mathsf{Hom}\; m\; n) \to \mathsf{cubicalType}\varepsilon\; (\alpha \cdot \mathsf{fType}_0) \\
\mathsf{fType}\varepsilon\; \{\; n\; \}\; m\; \alpha\; m'\; \alpha' = \mathsf{refl}
\end{array}
$$

Without a fibration structure for the universe, we cannot eliminate the propositional equality in large types.

7.2.5 Function Types and Dependent Products

In order to interpret the function types, we replicate Pédrot's construction and supplement it with a fibration structure adapted from the fibration structure of Cohen *et al.*

```
Arr₀ : {n : Obj}
     → (A0 : El₀ fType₀)
     → (Aε : Elε fType₀ fTypeε A0)
     → (B0 : El₀ fType₀)
     → (Bε : Elε fType₀ fTypeε B0)
     → El₀ fType₀
Arr₀ {n} A0 Aε B0 Bε m α = FT
  (λ m′ α′ → ∀ (x0 : El₀ (α ∘ α′ · A0))
               → (xε : Elε (α ∘ α′ · A0) (α ∘ α′ · Aε) x0)
               → FT₀ (B0 m′ (α ∘ α′)) m′ (id m′))
  (λ m′ α′ f0 → ∀ (x0 : El₀ (α ∘ α′ · A0))
                  → (xε : Elε (α ∘ α′ · A0) (α ∘ α′ · Aε) x0)
                  → FTε (B0 m′ (α ∘ α′)) m′ (id m′)
                    (λ m″ α″ → pfcast (f0 m″ α″ (α″ · x0) (α″ · xε))))
  (λ m′ α′ φ p0 pε s0 sε s~ x0 xε →
     let v0 = fill₀ (A0 (suc m′) (α ∘ α′)) m′ rev empty ∅ ∅ x0 xε ∅ in
     let vε = fillε (A0 (suc m′) (α ∘ α′)) m′ rev empty ∅ ∅ x0 xε ∅ in
     pfcast (comp₀ (B0 (suc m′) (α ∘ α′)) m′ (id (suc m′)) φ
       (λ m″ α″ Hφ → pfcast (p0 m″ α″ Hφ (α″ · v0) (α″ · vε)))
       (λ m″ α″ Hφ → pfcast (pε m″ α″ Hφ (α″ · v0) (α″ · vε)))
       (λ m″ α″ → pfcast (s0 m″ α″ (side0 ∘ α″ · v0) (side0 ∘ α″ · vε)))
       (λ m″ α″ → pfcast (sε m″ α″ (side0 ∘ α″ · v0) (side0 ∘ α″ · vε)))
       (λ m″ α″ Hφ → {- omitted -}))
  (λ m′ α′ φ p0 pε s0 sε s~ → {- omitted -})
  (λ m′ α′ φ p0 pε s0 sε s~ → {- omitted -})
```

rev is the morphism in the cube category that flips the first coordinate. empty is the minimal element of the face lattice. ∅ provides us with a partial element defined on empty.

For the sake of brevity, we omit the fields compε and comp~, which are computationally irrelevant anyway.

```
Arrε : {n : Obj}
     → (A0 : El₀ fType₀)
     → (Aε : Elε fType₀ fTypeε A0)
     → (B0 : El₀ fType₀)
     → (Bε : Elε fType₀ fTypeε B0)
     → Elε fType₀ fTypeε (Arr₀ A0 Aε B0 Bε)
Arrε {n} A0 Aε B0 Bε m α m′ α′ = refl
```

To define the λ-abstraction and the function application we can use the exact same definitions as in the translation of Pédrot, since the fibration structure is not involved. These definitions respect the β and η rules: assuming the involved terms are well-typed, the interpretation of $(\lambda x\,.\,t)\ u$ is convertible to the interpretation of $t[u/x]$, and the interpretation of $\lambda x\,.\,t\ x$ is convertible to the interpretation of t.

The construction for dependent products is very similar, if slightly more complex. We have not verified it in a proof assistant yet.

7.2.6 The Natural Numbers

We can extend our core translation to support natural numbers. We interpret the type of natural numbers as a constant prefascist, whose type of sections is equal to $\mathbb{N}$ in all dimensions. A family of sections is deemed compatible if they are all equal to the same natural number.

```
Nat0 : {n : Obj} → El₀ fType₀
Nat0 {n} m α = FT
  (λ m′ α′ → ℕ)
  (λ m′ α′ n → (∀ m″ α″ → n m″ α″ ~ n m′ (id m′)))
  (λ m′ α′ φ p0 pε s0 sε s~ → s0 m′ (id m′))
  (λ m′ α′ φ p0 pε s0 sε s~ → sε m′ (id m′))
  (λ m′ α′ φ p0 pε s0 sε s~ → {- omitted -})

Natε : {n : Obj} → Elε fType₀ fTypeε Nat₀
Natε {n} m α m′ α′ = refl
```

The composition operation is quite straightforward: an open box of natural numbers is necessarily constant, and thus the bottom element can serve as a lid.

As shown by Pédrot, we can interpret 0, the successor function, and the induction principle. Furthermore, the computation rule of the induction principle is interpreted by a definitional equality.

7.2.7 The Cubical Equality

The interpretation of the propositional equality type is more interesting. We cannot add a fibration structure to the equality type used by Pédrot, since this type is not fibrant in general. Instead we want to interpret an equality between x and y as a *path* from x to y.

In the cubical world, if x and y are two n-dimensional sections of a prefascist type A, then a path should be a $(n+1)$-dimensional cube which has x and y as opposing faces:

```
record cubicalPath {n : Obj}
  (A0 : El₀ fType₀) (Aε : Elε fType₀ fTypeε A0)
  (x0 : El₀ A0) (y0 : El₀ A0) : Set where
  constructor CE
  field
    CE₀ : El₀ (degen · A0)
    CEε : Elε (degen · A0) (degen · Aε) CE₀
    leftbndry : (side0 · CE₀) ~ x0
    rightbndry : (side1 · CE₀) ~ y0
```

This definition admits a natural restriction along morphisms of the cube category, that we (once again) write `restr`:

```
restr : (α : Hom m n) → cubicalPath A0 Aε x0 y0
      → cubicalPath (α · A0) (α · Aε) (α · x0) (α · y0)
restr α (CE ce0 ceε lb rb) = CE (lift α · ce0) (lift α · ceε)
                                (ap (λ x → α · x) lb) (ap (λ x → α · x) rb)
```

We use these to define the interpretation of the propositional equality.

```
Eq₀ : {n : Obj}
    → (A0 : El₀ fType₀) → (Aε : Elε fType₀ fTypeε A0)
    → (x0 : El₀ A0) → (xε : Elε A0 Aε x0)
    → (y0 : El₀ A0) → (yε : Elε A0 Aε y0)
    → El₀ fType₀
Eq₀ {n} A0 Aε x0 xε y0 yε m α = FT
  (λ m′ α′ → cubicalPath (α ∘ α · A0) (α ∘ α · Aε) (α ∘ α · x0) (α ∘ α · y0))
  (λ m′ α′ x → x ~ (λ m″ α″ → restr α″ (x m′ (id m′))))
  (λ m′ α′ φ p0 pε s0 sε s~ → CE
      (λ m″ α″ → pfcast (comp₀ A0 (m′+2) (α ∘ α′ ∘ degen')
          m″ (shift α″) (φ ∘ degen ∘ α″ ∨ proj0 ∨ ~ proj0)
          (merge0 p0 x0 y0) (mergeε p0 x0 y0)
          (pfcast (α″ · s0)) (pfcast (α″ · sε)) {- omitted -}))
      {- omitted -})
  (λ m′ α′ φ p0 pε s0 sε s~ → {- omitted -})
  (λ m′ α′ φ p0 pε s0 sε s~ → {- omitted -})

Eqε {n} A0 Aε x0 xε y0 yε m α m′ α′ = refl
```

The map degen' : Hom (n+2) (n+1) applies a degeneracy on the second coordinate, contrary to the map degen which acts on the first coordinate.
The map proj0 : Hom (n+1) 1 is the projection of the first coordinate.
The operator merge takes a partial element on face φ and a partial element on face ψ which are compatible on $\varphi \wedge \psi$ and produces a partial element on face $\varphi \vee \psi$ (in our case, it is generalized to three arguments).

We could have hoped to derive the fibration structure of the equality type from that of *A0* with a few algebraic manipulations, but it turns out to be surprisingly nontrivial because of the boundary conditions.

The proof of reflexivity is derived from a degeneracy morphism:

```
eqrefl₀ : {n : Obj}
        → (A0 : El₀ fType₀) → (Aε : Elε fType₀ fTypeε A0)
        → (x0 : El₀ A0) → (xε : Elε A0 Aε x0)
        → El₀ (Eq₀ A0 Aε x0 xε x0 xε)
eqrefl₀ {n} A0 Aε x0 xε m α = CE
  (α ∘ degen · x0) (α ∘ degen · xε) refl refl
```

and eqreflε is given by reflexivity.

In order to interpret the *J* eliminator, we break it into transport and contractibility of the singletons, following Cohen *et al.* The former can be derived from the fibration structure of the types, and the latter can be proved with the structure of the category of cubes and some heavy boundary condition yoga. We omit the proofs, as they get quite verbose.

Additionally, we were able to formally verify a proof of function extensionality, which can be derived without proving the univalence axiom.

7.2.8 Glue Types

So far, we have the elementary bricks of MLTT down: a universe of fibrant types, dependent products, natural numbers, and a propositional equality. However, our initial goal was to interpret the univalence axiom. How should we go about that?

In the model of Cohen *et al.*, this relies on *Glue types*, a construction that can be used to replace faces of a cubical presheaf with equivalent types. It seems clear that Glue types can be adapted to the world of fibrant prefascist types given the similarity with presheaves, but we have not formalized it yet.

Furthermore, Glue types allow us to define the fibration structure for the universe. As such, they are the obvious next step in this line of research.

7.3 Consequences and Perspectives

Our prefascist translation is not quite complete as it stands, but it hopefully makes a strong argument for the possibility to translate MLTT+Univalence into $\mathsf{CC}^{\mathsf{obs}+}$. In this section, we investigate the theoretical consequences of such a translation, if it were completed.

7.3.1 The system $\mathsf{HoTT}^{\mathsf{pf}}$

Assume that we have an extension of the functions $[\_]^n_0$ and $[\_]^n_\varepsilon$ from Subsection 7.2.4 to the full syntax of MLTT+Univalence, such that

- Given any typing judgment $\Gamma \vdash t : A$ that is derivable in MLTT+Univalence, the translated judgments are derivable in $\mathsf{CC}^{\mathsf{obs}+}$:

$$\llbracket\Gamma\rrbracket^n \vdash [t]^n_0 : \mathsf{El}_0\ [A]^n_0 : \mathscr{U}$$
$$\llbracket\Gamma\rrbracket^n \vdash [t]^n_\varepsilon : \mathsf{El}\varepsilon\ [A]^n_0\ [A]^n_\varepsilon\ [t]^n_0 : \Omega$$

- Given any convertibility judgment $\Gamma \vdash t \equiv u : A$ that is derivable in MLTT+ Univalence, the translated judgment is derivable in $\mathsf{CC}^{\mathsf{obs}}$:

$$\llbracket\Gamma\rrbracket^n \vdash [t]^n_0 \equiv [u]^n_0 : \mathsf{El}_0\ [A]^n_0 : \mathscr{U}$$

Then we can use this translation to define the system $\mathsf{HoTT}^{\mathsf{pf}}$. The syntax of $\mathsf{HoTT}^{\mathsf{pf}}$ is the syntax of MLTT+Univalence, and its typing judgments are defined *via* translation into $\mathsf{CC}^{\mathsf{obs}+}$:

- A context Γ of $\mathsf{HoTT}^{\mathsf{pf}}$ terms is well-formed when the translated context $\llbracket\Gamma\rrbracket^0$ is well-formed in $\mathsf{CC}^{\mathsf{obs}+}$,
- the typing judgment $\Gamma \vdash t : A$ is valid in $\mathsf{HoTT}^{\mathsf{pf}}$ when the two translated judgments $\llbracket\Gamma\rrbracket^n \vdash [t]^n_0 : \mathsf{El}_0\ [A]^n_0 : \mathscr{U}$ and $\llbracket\Gamma\rrbracket^n \vdash [t]^n_\varepsilon : \mathsf{El}\varepsilon\ [A]^n_0\ [A]^0_\varepsilon\ [t]^n_0 : \Omega$ are derivable in $\mathsf{CC}^{\mathsf{obs}+}$,
- the convertibility judgment $\Gamma \vdash t \equiv u : A$ is valid in $\mathsf{HoTT}^{\mathsf{pf}}$ when the translated judgment $\llbracket\Gamma\rrbracket^n \vdash [t]^n_0 \equiv [u]^n_0 : \mathsf{El}_0\ [A]^n_0 : \mathscr{U}$ is derivable in $\mathsf{CC}^{\mathsf{obs}+}$.

The system $\mathsf{HoTT}^{\mathsf{pf}}$ is clearly an extension of MLTT+Univalence, in the sense that every judgment that is valid in the latter is also valid in $\mathsf{HoTT}^{\mathsf{pf}}$. Additionally, thanks to our normalization proof for $\mathsf{CC}^{\mathsf{obs}+}$ from section 5.1, we can show that $\mathsf{HoTT}^{\mathsf{pf}}$ has interesting computational properties.

Theorem 7.3.1 *The conversion and the typing of* $\mathsf{HoTT}^{\mathrm{pf}}$ *are decidable.*

Theorem 7.3.2
If t is a term of type $\mathbb{N}$ in the empty context, then there exists an integer n such that t is propositionally equal to $S^n\ 0$.
Furthermore, there exists a program that can compute n from the term t.

This property is often called *homotopy canonicity*. It follows from an immediate inspection of the translation of the type of integers.

Finally, we can extract some proof-theoretic results:

Theorem 7.3.3 *Every integer function that can be defined in the empty context in* MLTT+Univalence *can also be defined in* $\mathsf{CC}^{\mathrm{obs+}}$.

And given that our normalization proof for $\mathsf{CC}^{\mathrm{obs+}}$ is done in MLTT with small induction-recursion, we can use the method described in Subsection 4.3.2 to show that $\mathsf{CC}^{\mathrm{obs+}}$ cannot define more functions than MLTT. Hence the following result:

Theorem 7.3.4 *Every integer function that can be defined in the empty context in* MLTT+Univalence *can also be defined in* MLTT.

7.3.2 Going further

These observations are interesting, but not exactly revolutionary: translating MLTT+Univalence into cubical type theory provides us with similar properties, in a way that is much more computationally efficient.

Our translation is arguably more modular, though: we can easily extend it to two-level type theories [63, 64] by using universes of non-fibrant prefascist types, and more generally, we can add any primitive that has a constructive interpretation in the cubical model—and we get a proof of decidability and homotopy canonicity for free, without needing to devise computation rules.

[63]: Voevodsky (2013), "A simple type system with two identity types"
[64]: Annenkov et al. (2017), "Two-Level Type Theory and Applications"

Cubical Synthetic Homotopy Theory

Synthetic Cubical Homotopy Theory

8

In the previous chapters, we spent a lot of time studying extensions of intensional type theory, which we moslty narrated as a quest for extensionality principles. But in addition to providing extensionality principles, univalent type theories also open up an entire new world of mathematics: since types acquire the properties of synthetic spaces, we can rephrase classical results from the mathematics of spaces in the language of type theory. And indeed, there is a growing literature on synthetic homotopy theory done in axiomatic homotopy type theory [18, 73–78]. In this chapter, we lay a foundation for synthetic homotopy theory in cubical type theory using the Cubical Agda proof assistant.

We start with a short introduction to homotopy theory and a discussion of the kind of results that we will encounter in this chapter in section 8.1. Then in section 8.2 we give an overview of cubical type theory and its implementation in Cubical Agda, and we use it to prove some basic results about the circle and the torus. In section 8.4 and section 8.5 we study three important constructions from homotopy theory: the suspensions, the homotopy pushout and the Hopf fibration. Finally, we compare our proofs with similar formalizations done in axiomatic HoTT in section 8.6.

All the code that appears in this chapter is written in Agda and has been integrated in the standard library of Cubical Agda. As a consequence we adopt the notations used in that library, which sometimes conflict with notations used in the previous chapters. We will make sure to be explicit about it and introduce every notation in due course.

The developments can be found in the `agda/cubical` library located at:
`https://github.com/agda/cubical`

8.1 Synthetic Homotopy Theory

In its more classical form, homotopy theory is a branch of mathematics that concerns itself with the study of topological spaces up to deformations. For instance, homotopical constructions should make no difference between the shape of a donut (a solid torus) and the shape of a coffee mug, as it is well-known that we can continuously deform one into the other – the hole in the handle of the mug corresponds to the hole in the donut. Homotopy theorists typically study ways to construct new spaces out of simpler ones (such as *suspensions* or *smash products*), invariants that can be used to distinguish spaces (such as the *fundamental group* or *cohomology theories*), or algebraic structures where equations have been weakened to homotopy equivalences (such as ∞*-groups* or $E\infty$*-rings*).

In HoTT, types are interpreted as spaces and equalities are interpreted as paths. Furthermore the univalence axiom tells us that equalities between types correspond to homotopy equivalences, which means that types are handled up to deformation. All in all, this seems like the perfect framework for *synthetic* homotopy theory, *i.e.* establishing

classical results from homotopy theory rephrased to talk about types instead of topological spaces. And indeed the community has developed impressive libraries of synthetic homotopy theory in axiomatic HoTT, including results such as homotopy groups of spheres [18, 73, 74], the Seifert-van Kampen theorem [76], Blakers-Massey theorem [77] and Serre's spectral sequence [78].

[18]: Brunerie (2016), "On the homotopy groups of spheres in homotopy type theory"
[73]: Licata et al. (2013), "Calculating the Fundamental Group of the Circle in Homotopy Type Theory"
[74]: Licata et al. (2013), "$\pi_n(S^n)$ in Homotopy Type Theory"
[76]: Hou (Favonia) et al. (2016), "The Seifert-van Kampen Theorem in Homotopy Type Theory"
[77]: Hou (Favonia) et al. (2016), "A Mechanization of the Blakers-Massey Connectivity Theorem in Homotopy Type Theory"
[78]: Doorn (2018), "On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory"
[79]: Sojakova (2016), "The Equivalence of the Torus and the Product of Two Circles in Homotopy Type Theory"
[80]: Licata et al. (2015), "A Cubical Approach to Synthetic Homotopy Theory"

However, despite these successes some constructions have turned out to be surprisingly difficult. An example of this is the proof that the torus is equivalent to the product of two circles: the first version required an impressive amount of complicated path algebra, worked out by Sojakova [79]. The proof was later simplified by Licata et al. [80] using cubical ideas, but it was still highly nontrivial. The main source of the difficulties in formalizing this proof is the fact that many equalities do not hold definitionally, and thus add to the complexity of the involved path algebra. These problems can be traced back to univalence and HITs being postulated with axioms; in particular the computation rules for the higher constructors of HITs do not hold definitionally.

But this is precisely why Coquand *et al.* developped cubical type theories [26, 27, 57], which provide univalence and HITs with proper computational content. The main goal of this chapter is to show the practical gains of using a system with native support for univalence and higher inductive types for formalizing synthetic homotopy theory. We exemplify this by formalizing the following results in cubical type theory:

- The equivalence of the torus and two circles together with the computation of their respective fundamental groups (section 8.3).
- The equivalence between direct definitions of low dimensional spheres, and alternative definitions using iterated suspensions (Subsection 8.4.1).
- Definition of pushout together with a direct proof of the "3 × 3 lemma" (Subsection 8.4.2).
- Definition of the join of two types and a proof that it is associative (Subsection 8.4.3). Using this we get two proofs, one inspired by HoTT and a new direct cubical proof, that $\mathbb{S}^3$ is equivalent to the join of two circles.
- Definitions of the Hopf fibration and a proof that its total space is $\mathbb{S}^3$ (section 8.5).

While the proofs in cubical type theory often resemble the original HoTT proofs, some work is typically required to make them more "cubical" in order to take full advantage of the cubical primitives. In particular, the use of path-induction that is ubiquitous in HoTT is kept to a minimum, and one typically instead uses the more elementary operations of cubical type theory such as transport and composition. With this strategy, many proofs that involved complicated path algebra in HoTT become much simpler. For instance, the proof that the torus is equivalent to the product of two circles is trivial to prove in cubical type theory using pattern matching and reflexivity as shown in section 8.3.

There are multiple variations and implementations of cubical type theory—the original formulation of Cohen et al. [57] used an interval endowed with the structure of a *De Morgan algebra* while the more

[57]: Cohen et al. (2018), "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom"

recent *cartesian* cubical type theories use an interval with less structure [26, 27]. All of the results in this paper have been formalized with the cubical extension [81] of the Agda proof assistant, which is based on the De Morgan variation of cubical type theory. Despite some technical differences between the underlying theories, all of the proofs could be carried with similar complexity in a system based on cartesian cubical type theory, like redtt [29].

[26]: Angiuli et al. (2018), "Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities"
[27]: Angiuli et al. (2019), "Syntax and Models of Cartesian Cubical Type Theory"
[81]: Vezzosi et al. (2019), "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types"
[29]: The RedPRL Development Team (2018), *The redtt Proof Assistant*

8.2 Cubical Type Theory and Cubical Agda

The goal of this section is to give sufficient background for readers not familiar with cubical type theory and Cubical Agda to be able to understand our code. Readers who are familiar with Cubical Agda can thus skip this section, and readers who wish to understand Cubical Agda in greater depth should have a look at the introductory paper by Vezzosi *et al.* [81].

[81]: Vezzosi et al. (2019), "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types"

8.2.1 The Interval and Path Types

The first addition to upgrade intensional type theory into cubical type theory is an *interval* type I with two endpoints i0 and i1. This type plays the role of the real interval $[0, 1] \subset \mathbb{R}$ in homotopy theory, except that in cubical type theory the interval is a purely formal object: it does not contain anything like real numbers.

There is a technicality here: the interval is a special "non-fibrant" type which does not live in the universe of regular "fibrant" types. It is safe to ignore this detail in first approach.

A variable $i : \mathsf{I}$ should be tought of as a point varying continuously between the two endpoints. The interval is equipped with three basic operations: *minimum* (_∧_ : I → I → I), *maximum* (_∨_ : I → I → I) and *reversal* (~_ : I → I). The interval with these operations has the structure of a *De Morgan algebra*, *i.e.* a bounded distributive lattice (i0, i1, _∧_, _∨_) with a De Morgan involution ~_.

A function out of the interval into a type A should be thought of as a *line* in the space A.

Even though I is non-fibrant, we can form function spaces into regular types such as I → A, and the resulting type is itself a regular type.

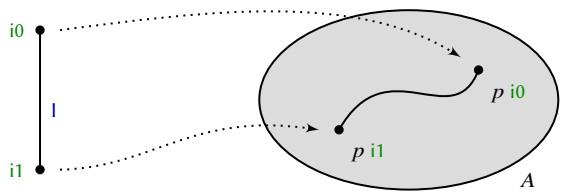


Figure 8.1: Graphical interpretation of a function $p : \mathsf{I} \to A$

Thus in particular, a function p from I to one of Agda's universes of types (Set ℓ) represents a line of types that varies continuously between the two types p i0 and p i1. Similarly a function $p' : \mathsf{I} \to \mathsf{I} \to \mathsf{Set}\ \ell$ is a square of types; and by iterating this process we get cubes and hypercubes of types, hence the *cubical* nature of cubical type theory.

Given a line of types $A : \mathsf{I} \to \mathsf{Set}\ \ell$, we can form the type of *dependent lines* $(i : \mathsf{I}) \to A\ i$. And since it is often useful to specify the endpoints of a line, Cubical Agda provides primitive *path types* that add constraints on the endpoints:

we omit the quantification of the universe level ℓ for readability

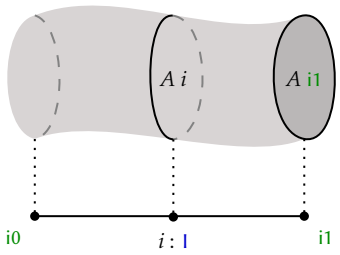


Figure 8.2: Graphical representation of a line of types $A : \mathsf{I} \to \mathsf{Set}\ \ell$, or equivalently a type depending on $i : \mathsf{I}$

```
PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ
```

Paths are introduced by lambda abstractions:

$$\frac{\Gamma,\ i : \mathsf{I} \ \vdash\ t : A\ i}{\Gamma \ \vdash\ \lambda i \to t : \mathsf{PathP}\ A\ t[\mathsf{i0}\ /\ i]\ t[\mathsf{i1}\ /\ i]}$$

Then given $p : \mathsf{PathP}\ A\ a_0\ a_1$, we can apply it to $r : \mathsf{I}$ and obtain $p\ r : A\ r$. And we always have that $p\ \mathsf{i0}$ is convertible to a_0 and $p\ \mathsf{i1}$ is convertible to a_1, even when p is a neutral term.

The type PathP plays the role of propositional equality in cubical type theory. More specifically it has the role of heterogeneous equality, since the two endpoints are in different types; this is similar to the dependent paths in HoTT [HoTT, Sect. 6.2]. We define the type of homogeneous paths/equalities in terms of PathP as follows:

[HoTT]: Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

```
_≡_ : {A : Set ℓ} → A → A → Set ℓ
_≡_ {A = A} x y = PathP (λ _ → A) x y
```

The homogeneous path types are written using a triple equality symbol, following the convention of Cubical Agda. The reader should be careful to not get confused with the inductive equality type of chapter 6, or the definitional equality of chapter 3.

The syntax $\{A = A\}$ in the definition tells Agda to bind the hidden argument A (first occurence) to a variable A (second occurence) that can be used on the right hand side of the definition. Viewing equalities as paths allows us to reason about equality; for instance the constant path represents a proof of equality by reflexivity.

```
refl : {x : A} → x ≡ x
refl {x = x} = λ i → x
```

Using path types as equalities lets us prove theorems that are not provable in standard Agda. For example, the principle of function extensionality (which states that two pointwise equal functions are equal) has a very simple proof:

```
funExt : {f g : A → B} → ((x : A) → f x ≡ g x) → f ≡ g
funExt p i x = p x i
```

When we input this definition to Cubical Agda, the proof assistant checks that $p\ x\ i$ is convertible to $f\ x$ when $i = \mathsf{i0}$ and convertible to $g\ x$ when $i = \mathsf{i1}$. And it is indeed the case, because $p\ x$ has type $f\ x \equiv g\ x$.

The proof of function extensionality for dependent and n-ary functions is equally easy. Since funExt is a theorem and not an axiom, it has computational content: it simply swaps the arguments to p.

We can also use the de Morgan structure on I to construct various useful operations, for example the *reversal* of a path is defined using ~_ and encodes the fact that ≡ is symmetric.

In this definition, the boundary conditions are met because ~i0 = i1 and ~i1 = i0

```
_⁻¹ : {x y : A} → x ≡ y → y ≡ x
p ⁻¹ = λ i → p (~ i)
```

8.2.2 Transport and Composition

Transport One of the basic uses of the propositional equality in type theory is to perform *type coercions*: given an equality/path between two types A and B, we should get a function from A to B. In Cubical Agda this is the job of the transport operator, which is defined in terms of the more general operator transp.

```
transport : A ≡ B → A → B
transport p a = transp (λ i → p i) i0 a
```

The transp operator is a primitive object in cubical type theory, and it satisfies complex computation rules. But transport will be sufficient for the purposes of this chapter.

Using transport, we can substitute a term with another path-equal term.

```
subst : (B : A → Set ℓ) {x y : A} → x ≡ y → B x → B y
subst B p b = transport (λ i → B (p i)) b
```

This function invokes transport with a proof that the family B respects the equality p:

$$\lambda\ i \to B\ (p\ i) : B\ x \equiv B\ y$$

If we combine the transport operator with some simple de Morgan algebra, we can even define an induction principle for paths that resembles the J eliminator for Martin-Löf's inductively defined identity types [2].

[2]: Martin-Löf (1975), "An Intiutionistic Theory of Types: Predicative Part"

```
J : {x : A} (P : ∀ y → x ≡ y → Set ℓ) (d : P x refl)
  {y : A} (p : x ≡ y) → P y p
J P d p = transport (λ i → P (p i) (λ j → p (i ∧ j))) d
```

However, an important difference between the cubical path types and Martin-Löf's identity types (and the path types in axiomatic HoTT) is that cubical path types are not inductive types. And in particular, the above definition does not satisfy the computation rule when applied to refl. Nevertheless, we can still prove that it holds up to a path:

In this sense, the cubical equality is similar to the observational equality of chapter 3: both provide new extensionality principles in exchange for a weakening the computation rule of J to a propositional equality.

```
JRefl : {x : A} (P : ∀ y → x ≡ y → Set ℓ) (d : P x refl) →
        J P d refl ≡ d
```

Readers who are familiar with HoTT might be worrying that this weakened computation rule may complicate proofs; however in our experience this is rarely the case, because most proofs done *via* path induction in HoTT can be done in more direct ways using cubical primitives.

Composition Cubical type theory also provides a primitive operation called *homogeneous composition* which allows us to compose paths and more generally, to compose higher dimensional cubes.

In the type Partial r A, the second argument r is an element of I that is used to select faces from the context. For instance if the context contains two variables i, j of type I, then every term we define naturally has the shape of a square. Then the constraint $i \vee j$ = i1 selects a union of two faces of the square.

In order to describe homogeneous composition, we need to have a notion of partially specified n-dimensional cubes, *i.e.* cubes where

some faces are missing. For this reason, cubical type theory allows us to manipulate *partial terms*: in Cubical Agda, the type Partial r A contains elements of A which are only defined when $(r = \mathsf{i1})$ holds. For instance, if we are working in context that contains a variable $i : \mathbb{I}$ and A is a well-formed type, we can form the type Partial $(i \vee \sim i)$ A of elements of A which are defined when $i = \mathsf{i1}$ or when $i = \mathsf{i0}$.

In Cubical Agda, elements of partial cubical types are introduced with pattern-matching on the constraint. For this purpose Cubical Agda adds support for a new type of patterns, of which $(i = \mathsf{i1})$ and $(i = \mathsf{i0})$ are examples in the following definition.

```
partialBool : ∀ i → Partial (i ∨ ~ i) Bool
partialBool i (i = i1) = true
partialBool i (i = i0) = false
```

To help differentiate them from normal pattern matching on inductive types, we will write partial elements with pattern-matching lambdas.

```
partialBool : ∀ i → Partial (i ∨ ~ i) Bool
partialBool i = λ { (i = i1) → true ; (i = i0) → false }
```

The term partialBool should be thought of as a boolean depending on i with different values when i is i1 and when i is i0. This is only defined on the two endpoints of $\mathbb{I}$ and there is no way to extend this partial type to a regular dependent boolean, or otherwise we would get a path between true and false.

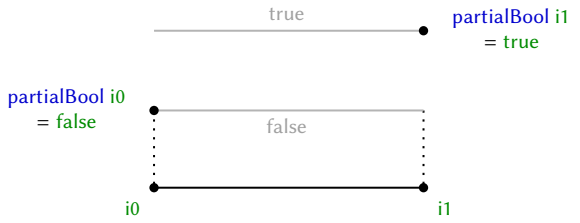


Figure 8.3: Graphical representation of the partial element partialBool

Cubical type theory also provides *cubical subtypes* [57]. Given a type A : Set ℓ, a face constraint encoded by an element of the interval $r : \mathbb{I}$ and a partial term u : Partial r A, we can form the type A [$r \mapsto u$] of total elements of A that extend u. A term v of this type is a term of type A that is definitionally equal to u when the constraint $(r = \mathsf{i1})$ holds.

[57]: Cohen et al. (2018), "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom"

Any term $u : A$ can be seen as a term of type A [$r \mapsto u$] that agrees with itself when $(r = \mathsf{i1})$ holds.

```
inS : {r : I} (a : A) → A [ r ↦ (λ _ → a) ]
```

We can also forget that a partial element agrees with u when $r = \mathsf{i1}$.

```
outS : {r : I} {u : Partial r A} → A [ r ↦ u ] → A
```

And these two operations are inverse to each other when well-typed. Using this cubical infrastructure we can now give the type of the homogeneous composition operation.

```
hcomp : {r : I} (u : I → Partial r A) (t : A [ r ↦ u i0 ]) → A
```

The meaning of this type is a bit opaque in first approach. To get a clearer picture of hcomp, we can start by looking at the special case hcomp {r = i ∨ ~ i} u t, where assume that we are working in a context that contains the interval variable *i*—in other words, *A* is a type indexed over the interval I as pictured in figure 8.4.

Then *t* is a total element of *A* and *u* extends *t* when *i* = i0 or *i* = i1, resulting in the shape of an open box. Cubical Agda makes sure that *t* agrees with *u* i0 when the constraint is satisfied; this is specified by making *t* an extension type. The hcomp operation then gives us the missing side of the box.

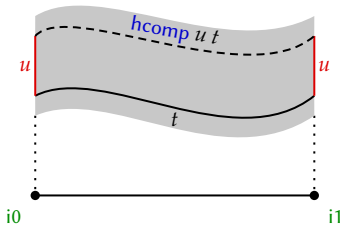


Figure 8.4: Graphical representation of the homogeneous composition hcomp {r = i ∨ ~ i} u t

The general case does pretty much the same, but with an arbitrary number of dimension and an arbitrary face constraint. As an example for the use of hcomp, we can define binary composition of paths:

```
_·_ : {x y z : A} → x ≡ y → y ≡ z → x ≡ z
_·_ {x = x} p q i = hcomp {r = i ∨ ~ i}
                          (λ j → λ { (i = i0) → x
                                   ; (i = i1) → q j })
                          (inS (p i))
```

Pictorially we are given $p : x \equiv y$ and $q : y \equiv z$, and the composite of the two paths is obtained by computing the dashed top of the following square.

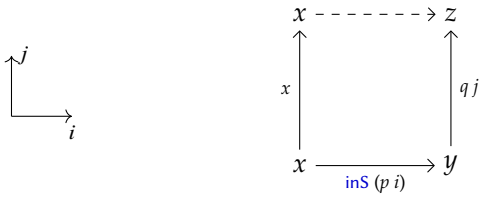


Figure 8.5: Composing two paths using hcomp

By composing paths and higher cubes using hcomp, we can reason about equalities/paths in a very direct way, avoiding the use of path induction.

8.2.3 Higher Inductive Types

Cubical Agda has native support for HITs, which means that we can define the circle using an Agda data declaration.

```
data S¹ : Set where
  base : S¹
  loop : base ≡ base
```

This is a type with a point constructor base and a nontrivial equality / path constructor loop connecting base to itself as shown in the drawing below.

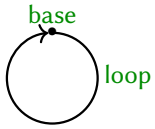


Figure 8.6: Graphical representation of the HIT S¹

Functions out of HITs are written using normal Agda pattern matching equations. The following function wraps the circle twice around itself, by composing the loop constructor with itself:

```
double : S¹ → S¹
double base = base
double (loop i) = (loop ∙ loop) i
```

When we input this definition by pattern matching, Cubical Agda checks that the boundary constraints from the definition of S¹ are respected: since loop i0 = base and loop i1 = base, we want to have double (loop i0) = double base and double (loop i1) = double base.

Both cases define a *definitional* equality, while in axiomatic HoTT this would have had to be defined using the postulated recursion principle for the circle. This means that in HoTT double applied to loop would not reduce automatically, but a proof of equality would instead have to be applied manually. This leads to rather bureaucratic proofs as one then has to handle these explicit applications of the computation rule when reasoning about double. Furthermore, this is not very natural if one wants to use HITs for programming.

In order for the circle to be the free type generated by base and loop, it also needs to have elements of the form loop ∙ loop as in the above definition of double. In general, for HITs hcomp ($\lambda\, i \to \lambda\, \{\ (r = \mathrm{i1}) \to u\}$) u_0 only reduces to u[i1 / i] when r is i1, and is considered to be a canonical element otherwise. The circle hence has constructors of the form hcomp $u\ u_0$ in addition to base and loop.

8.2.4 Glue Types and Univalence

The final ingredient of cubical type theory relevant for this chapter are the Glue types of Cohen et al. [57] that let us give computational content to univalence. Given that a type in cubical type naturally has the shape of a higher dimensional cube, Glue types let us replace the faces of these cubes with some *equivalent* types.

[57]: Cohen et al. (2018), "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom"

There are many ways to define the notion of "equivalence of types" in HoTT; Cubical Agda uses the definition that two types are equivalent if there is a function between them with *"contractible fibers"* following the terminology of Voevodsky [82] and the HoTT book [83]. Spelled out, a map $f : A \to B$ is an equivalence if the preimage of any point in B is a singleton type. We write $A \simeq B$ if there is a chosen equivalence $f : A \to B$ between A and B. A key result is that isomorphisms, in the sense of a section-retraction pair of functions, give rise to equivalences. In particular, the identity function is an equivalence: idEquiv A : $A \simeq A$.

[82]: Voevodsky (2015), "An experimental library of formalized Mathematics based on the univalent foundations"

[83]: Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

Equivalences are formally defined as follows:
isEquiv f = $(y : B) \to$ isContr (fiber $f\ y$)
fiber $f\ y = \Sigma\ (x : A)\ .\ f\ x \equiv y$
isContr $A = \Sigma\ (a : A)\ .\ ((x : A) \to x \equiv a)$

```
Glue : (B : Set ℓ) {r : I} →
       Partial r (Σ[ A ∈ Set ℓ ] (A ≃ B)) → Set ℓ
```

The Glue primitive takes a base type B, a constraint r and a partial family of types A that are equivalent to B when r = i1 holds, and produces type by "gluing" A onto B. The resulting type is definitionally equal to A when the constraint r = i1 holds.

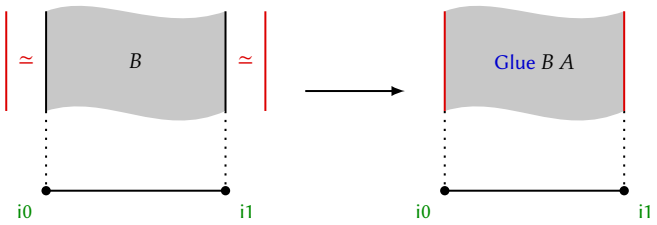


Figure 8.7: Graphical representation of Glue B $\{r = i \vee \sim i\}$ A. We start with a type B indexed over I and we replaced its faces with equivalent types.

Elements of a Glue type are introduced using the glue constructor and eliminated using the unglue operation. Examples of this will be discussed in the proof of theorem 8.5.2.

With Glue types, we can turn an equivalence of types into a path:

```
ua : {A B : Set ℓ} → A ≃ B → A ≡ B
ua {A = A} {B = B} e i =
  Glue B (λ { (i = i0) → (A , e) ; (i = i1) → (B , idEquiv B) })
```

The idea is that we glue A onto B when i is i0 using e and B onto itself when i is i1 using the identity equivalence. The term ua e is a path from A to B as the Glue type reduces when the face conditions are satisfied—when i is i0 this reduces to A and when i is i1 it reduces to B.

The univalence axiom is a little more complex, since it asks for the obvious function $A \equiv B \to A \simeq B$ to be an equivalence. We can also derive it from Glue, but for all of the examples in this chapter the ua function will be sufficient.

8.3 The Circle and Torus

We already gave the definition of the circle as a HIT in Subsection 8.2.3. Similarly, we can define the torus as a datatype in Cubical Agda as follows:

```
data Torus : Set where
  point  : Torus
  line1  : point ≡ point
  line2  : point ≡ point
  square : PathP (λ i → line1 i ≡ line1 i) line2 line2
```

The idea is that the Torus has a base point with two nontrivial path constructors connecting it to itself and a square relating the two paths. This square can be illustrated by:

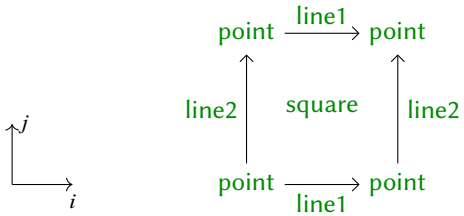


Figure 8.8: The two-dimensional face of the torus

The type of the square constructor captures this by identifying line2 with itself *over* line1. In order to see that this represents a torus imagine square being made of a piece of paper that is folded so that the opposite sides are matched up. The proof that the Torus type is equivalent to two circles in Cubical Agda was given by Vezzosi et al. [81, Section 2.4.1], but we recall it here for completeness. We first write a function from the torus to two circles using pattern matching.

[81]: Vezzosi et al. (2019), "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types"

```
t2c : Torus → S¹ × S¹
t2c point        = (base   , base)
t2c (line1 i)    = (loop i , base)
t2c (line2 j)    = (base   , loop j)
t2c (square i j) = (loop i , loop j)
```

To prove that this is an equivalence we need to define its inverse c2t : S¹ × S¹ → Torus. This function is also defined by pattern matching in the obvious way. Proving that the two maps cancel is then simply done by pattern matching with refl in all cases.

```
c2t-t2c : (t : Torus) → c2t (t2c t) ≡ t
c2t-t2c point        = refl
c2t-t2c (line1 _)    = refl
c2t-t2c (line2 _)    = refl
c2t-t2c (square _ _) = refl
```

The converse, t2c-c2t : (p : S¹ × S¹) → t2c (c2t p) ≡ p, is equally trivial to prove. We can then package this up as an equality using isoToPath which combines ua with the proof that any isomorphism is an equivalence.

```
Torus≡S¹×S¹ : Torus ≡ S¹ × S¹
Torus≡S¹×S¹ = isoToPath (iso t2c c2t t2c-c2t c2t-t2c)
```

This proof is trivial, thanks to the computation rules for all constructors of HITs holding definitionally in Cubical Agda. In axiomatic HoTT, the computation rules for the higher constructors would have to be postulated as axioms, which means that they do not hold definitionally. This is exactly what makes the proofs of c2t-t2c and t2c-c2t surprisingly nontrivial in HoTT [79].

[79]: Sojakova (2016), "The Equivalence of the Torus and the Product of Two Circles in Homotopy Type Theory"

8.3.1 The Loop Spaces of the Circle and Torus

The loop spaces of the circle and torus are defined as follows:

```
ΩS¹ : Set
ΩS¹ = base ≡ base
```

```
ΩTorus : Set
ΩTorus = point ≡ point
```

The goal of this section is to prove that ΩS¹ is equivalent to the integers. This proof is a cubical adaptation of the proof of Licata et al. [73]. We can then combine this with the above equivalence between the torus and two circles to also compute the loop space of the torus. Note that we are computing loop spaces and not fundamental groups. However, as the fundamental group is defined as the set-truncation of the loop space and these loop spaces are both sets, we get that they coincide with the fundamental groups.

[73]: Licata et al. (2013), "Calculating the Fundamental Group of the Circle in Homotopy Type Theory"

The loop space should be called the "fundamental ∞-group".

The first step in computing the loop space of the circle is to define a function computing "winding numbers", i.e. the net number of times an element of ΩS¹ goes around the circle clockwise. To do this we first prove that the successor function on the signed integers Int is an equivalence (its inverse is the predecessor function).

By applying ua we then get a nontrivial equality/path from Int to Int that we call sucPathInt. Using this we can define:

```
helix : S¹ → Set
helix base = Int
helix (loop i) = sucPathInt i

winding : ΩS¹ → Int
winding p = subst helix p (pos 0)
```

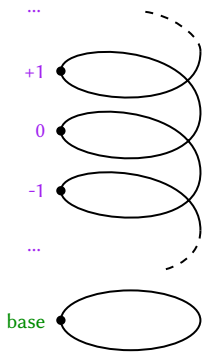


Figure 8.9: Graphical representation of the helix as a dependent type on S¹. A copy of the integers sits over base, and the type equivalence corresponding to the successor function sits over loop.

Applying the winding function to an element of ΩS¹ will compute its winding number. For example, we can compute the winding numbers +3 and −1 as follows:

```
_ : winding (loop · loop · loop) ≡ pos 3
_ = refl

_ : winding (loop ⁻¹ · loop · loop ⁻¹) ≡ negsuc 0
_ = refl
```

The term negsuc n represents the number $-(n+1)$, so that negsuc 0 is indeed −1. Note that none of these examples would have reduced to a numeral in axiomatic HoTT as they would have been stuck on transporting along ua. The proofs of these would hence not be proved by refl, but rather by manually rewriting with the postulated computation rules for univalence.

In order to prove that ΩS¹ ≡ Int we just have to define an inverse function to winding. This is easily done via pattern matching.

```
loopn : Int → ΩS¹
loopn (pos zero) = refl
loopn (pos (suc n)) = loopn (pos n) · loop
loopn (negsuc zero) = loop ⁻¹
loopn (negsuc (suc n)) = loopn (negsuc n) · loop ⁻¹
```

It is then easy to prove that the winding number of an n-fold loop is n.

```
winding-loopn : (n : Int) → winding (loopn n) ≡ n
winding-loopn (pos zero) = refl
winding-loopn (pos (suc n)) i =
  sucInt (winding-loopn (pos n) i)
winding-loopn (negsuc zero) = refl
winding-loopn (negsuc (suc n)) i =
  predInt (winding-loopn (negsuc n) i)
```

Note that we parameterize by i in the second and fourth case of winding-loopn. The reason for this is that we are constructing an element of ≡, i.e. a function out of I. This use of interval variables hence lets us inline cong/ap.

However, when trying to prove the other composition one quickly realizes that it is not as easy as there is no direct induction principle for ΩS¹. Luckily there is an ingenious solution to this offered by HoTT in the form of the *encode-decode method* [83, Section 8.9]. The trick is to generalize the loopn and winding functions as follows:

[83]: Univalent Foundations Program (2013), *Homotopy Type Theory: Univalent Foundations of Mathematics*

```
encode : ∀ x → base ≡ x → helix x
encode x p = subst helix p (pos 0)

decode : (x : S¹) → helix x → base ≡ x
decode base = loopn
decode (loop i) = {-  ...  -}
```

Note that decode base is loopn and that encode base is winding definitionally. The loop case of decode is not too difficult to define cubically, but it does involve a few lines of path algebra. We chose to omit it from our exposition to avoid getting sidetracked, but the interested reader can find it in the formalized proof. The main reason for generalizing the functions as above is that we can now use path induction to prove the following:

```
decodeEncode : (x : S¹) (p : base ≡ x) →
                 decode x (encode x p) ≡ p
decodeEncode x p =
  J (λ y q → decode y (encode y q) ≡ q) (λ x → refl) p
```

The special case decodeEncode base proves the desired composition, i.e. that loopn (winding x) ≡ x for all x : ΩS¹. We may then package this up to get the desired equality:

```
ΩS¹≡Int : ΩS¹ ≡ Int
ΩS¹≡Int = isoToPath (iso winding loopn
                         winding-loopn
                         (decodeEncode base))
```

By combining this result with the equality between the torus and two circles we obtain:

```
ΩTorus≡Int×Int : ΩTorus ≡ Int × Int
```

The loop space of a cartesian product of two types is the cartesian product of the individual loop spaces, since the loop space is more or less a function type.

It is now possible to transport along this equality to compute winding numbers on the torus. However, this will not result in the most efficient

function possible and we can easily write a more direct function for computing these numbers as follows.

```
windingTorus : ΩTorus → Int × Int
windingTorus l = ( winding (λ i → t2c (l i) .fst)
                 , winding (λ i → t2c (l i) .snd))
```

Just like winding this function also computes as expected:

```
_ : windingTorus (line1 · line2) ≡ (pos 1 , pos 1)
_ = refl

_ : windingTorus (line1 ⁻¹ · line2 · line1) ≡ (pos 0 , pos 1)
_ = refl
```

8.4 Suspension, Spheres and Pushouts

In this section we discuss various results about spheres, culminating in two proofs that the 3-dimensional sphere is equal to the join of two circles — a result that we will need when computing the total space of the Hopf fibration later on. On the way to this result we introduce the pushout and join HITs. We also prove some useful results about these, including the "3 × 3 lemma" for pushouts and associativity of the join.

8.4.1 Suspension

The suspension of a type A is built from two distinguished points north and south, along with a path from north to south for every point of A (and a homotopy square for every path in A, etc.):

```
data Susp (A : Set) : Set where
  north : Susp A
  south : Susp A
  merid : (a : A) → north ≡ south
```

The suspension of a type A is depicted below.

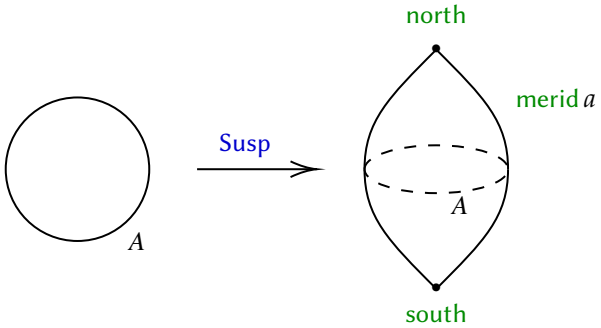


Figure 8.10: The suspension of a circle is a sphere

Remark that the regular sphere is considered 2-dimensional, even though it is the unit sphere of 3-dimensional space. This is because the dimension refers to the number of parameters needed to (locally) describe a point on the object.

In this drawing A is a circle, in which case the suspension of A is a sphere. This construction generalizes well, and we can define the

higher dimensional spheres as iterated suspensions starting from the booleans.

```
_-sphere : ℕ → Set
(0)-sphere = Bool
(suc n)-sphere = Susp ((n)-sphere)
```

We may now prove that our initial definition for the circle is equal to the suspension of the booleans.

Lemma 8.4.1 *The (1)-sphere is equal to the circle S¹.*

Proof. We will prove this by constructing an isomorphism and then converting it to an equality using univalence. We first define a map from (1)-sphere to S¹ in the following way, collapsing merid false to base:

```
s2c : (1)-sphere → S¹
s2c north          = base
s2c south          = base
s2c (merid false i) = base
s2c (merid true i)  = loop i
```

In the other direction, we use composition to go around the (1)-sphere in one go.

```
c2s : S¹ → (1)-sphere
c2s base = north
c2s (loop i) = (merid true ∙ merid false ⁻¹) i
```

To construct the first canceling homotopy, we have to find a homotopy between s2c (c2s loop) = loop ∙ refl and loop. This is proved in one of the lemmas in the cubical library that says that refl is the right unit for _∙_.

```
s2c-c2s : (x : S¹) → s2c (c2s x) ≡ x
s2c-c2s base = refl
s2c-c2s (loop i) j = rUnit loop (~ j) i
```

The second homotopy is slightly more involved. After going back and forth, merid false has been collapsed to the north pole, while merid true has been stretched to go around the whole circle. As such, we need to move the south pole back into place along merid false, and deform the two meridians accordingly:

```
c2s-s2c : (x : (1)-sphere) → c2s (s2c x) ≡ x
c2s-s2c north j = north
c2s-s2c south j = merid false j
c2s-s2c (merid false i) j = h1 i j
c2s-s2c (merid true i) j = h2 i j
```

We need h1 to be a homotopy from the constant path at north to the original path merid false, such that the restriction to (i = i1) matches merid false j and the restriction to (i = i0) matches north. This is easily achieved using _∧_:

```
h1 : I → I → (1)-sphere
h1 i j = merid false (i ∧ j)
```

On the other hand, h2 has to be a homotopy from merid true · merid false ⁻¹ to merid true, with the same condition on the restrictions to (*i* = i0) and (*i* = i1). We can do that using hcomp to paste several homotopies together:

```
h2 : I → I → (1)-sphere
h2 i j = hcomp (λ k → λ { (i = i0) → north
                        ; (i = i1) → merid false (j ∨ ~ k)
                        ; (j = i1) → merid true i })
               (merid true i)
```

The composition can be pictured as follows, showing both the outer edge constraints and the inner faces we used:

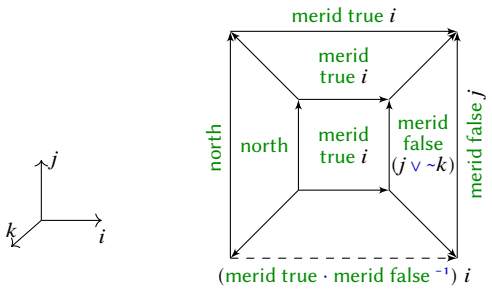


Figure 8.11: Using a two-dimensional hcomp to build a homotopy. The drawing represents an open box viewed from the top.

The (*j* = i0) face of the open cube matches the square used to define the composition merid true · merid false ⁻¹ — this breaks the composition into refl, merid true, and merid false ⁻¹ on the inner edges. Thus, we use the (*i* = i1) face to retract the merid false ⁻¹ part into the south pole, using a connection. The other faces are just constant in directions *j* and *k*. From there, hcomp provides us with the front face of the cube, which is the homotopy we need.

This completes the definition of c2s-s2c, providing us with an isomorphism than can be converted into an equality with ua. □

Inspired by the direct definition of S^1 we can also give direct definitions of some higher spheres, for example S^2 and S^3 are defined as follows.

```
data S² : Set where
  base₂ : S²
  surf₂ : PathP (λ i → base₂ ≡ base₂) refl refl

data S³ : Set where
  base₃ : S³
  surf₃ : PathP (λ j → PathP (λ i → base₃ ≡ base₃) refl refl)
                refl refl
```

As expected we can prove that these are equal to the definitions using iterated suspensions. The proof of this lemma is very similar to the one of lemma 8.4.1 so we omit it.

Lemma 8.4.2 *The (2)-sphere is equal to S^2 and the (3)-sphere is equal to S^3.*

One might wonder if we can also give a direct definition of S^n in a similar fashion and prove that it is equal to the (n-1)-sphere. However, it is currently not possible to write the higher constructor corresponding to surf_n in Cubical Agda as this kind of HIT is not supported by any of the proposed schemas for cubical HITs [25, 84]. Interestingly it is in fact possible to *postulate* this HIT in HoTT or cubical type theory, and prove properties about it — for instance Licata et al. [74] prove that $\pi_n(\mathbb{S}^n) = \mathbb{Z}$ using this direct definition.

[25]: Cavallo et al. (2019), "Higher Inductive Types in Cubical Computational Type Theory"
[84]: Coquand et al. (2018), "On Higher Inductive Types in Cubical Type Theory"
[74]: Licata et al. (2013), "$\pi_n(S^n)$ in Homotopy Type Theory"

8.4.2 Pushouts and the 3 × 3 Lemma

Suppose we are given a span of types, consisiting in three types and two functions:

$$B \xleftarrow{f} A \xrightarrow{g} C$$

Then the homotopical pushout of this span is a type that contains both a copy of B and a copy of C, and a path identifying $f\,a$ and $g\,a$ for every a in A. It can be defined as a HIT in the following way.

```
data Pushout {A B C : Set} (f : A → B) (g : A → C) : Set where
  inl : B → Pushout f g
  inr : C → Pushout f g
  push : (a : A) → inl (f a) ≡ inr (g a)
```

Homotopy pushouts are an instance of a *homotopy colimit*, the adequate notion of colimits when morphisms are defined up to a deformation.

A generic homotopical pushout is illustrated below: it consists of an embedded copy of B and C, and a copy of $A \times I$ whose ends have been identified with corresponding points in B and C according to f and g.

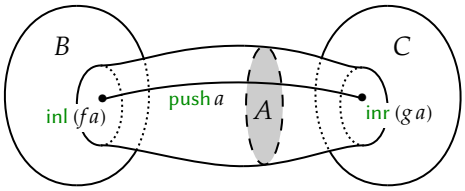


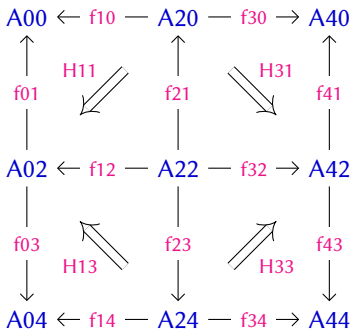
Figure 8.12: The homotopical pushout of the span $B \leftarrow A \rightarrow C$

Pushouts can be used to construct various classical objects such as suspensions, joins and many more—hence their importance. For instance, the suspension of a type A is the homotopical pushout of the following span:

$$1 \xleftarrow{} A \xrightarrow{} 1$$

where 1 is the inductive type with only one constructor, and the arrows are the unique maps to 1.

Since pushouts are so ubiquitous, it is not rare to deal with nested pushouts. In this case, a lemma known as the 3 × 3 lemma comes in very handy to rearrange expressions: suppose we are given a double span of types



where the Hij are homotopies ensuring the commutativity of the diagram: for instance, H11 is of type f10 ∘ f21 ≡ f01 ∘ f12.

Then, there are two canonical ways to build a type from this diagram out of pushouts. We can start by taking the pushouts of the three rows, to get the following span

$$A\square 0 \xleftarrow{f\square 1} A\square 2 \xrightarrow{f\square 3} A\square 4 \tag{8.1}$$

in which we write A□0 for the pushout of the first row A00 ← A20 → A40, A□2 for the pushout of the second row and A□4 for the pushout of the third row. The maps between them, f□1 and f□3, are then defined using pattern matching as follows:

```
f□1 : A□2 → A□0
f□1 (inl a) = inl (f01 a)
f□1 (inr a) = inr (f41 a)
f□1 (push a j) =
  hcomp (λ i → λ { (j = i0) → inl (H11 a (~ i))
                 ; (j = i1) → inr (H31 a (~ i)) })
        (push (f21 a) j)
```

Finally, we write A□○ for the pushout of the span in (8.1). Conversely, we could have started by taking the pushout of the columns and then computed the pushout A○□ of the resulting "horizontal" span. The 3 × 3 lemma for pushouts then states that the results of these two constructions are equal.

On a more abstract level, both A□○ and A○□ are homotopy colimits of the 3 × 3 diagram, and as such they ought to be equivalent.

Lemma 8.4.3 *(3 × 3 lemma) The two pushouts A□○ and A○□ are equal.*

Proof. In order to prove this we construct an isomorphism via pattern matching and apply univalence. To define a map A□○ → A○□, we will first need maps for the two sides of our pushout span:

```
A□0-A○□ : A□0 → A○□
A□0-A○□ (inl x) = inl (inl x)
A□0-A○□ (inr x) = inr (inl x)
A□0-A○□ (push a i) = push (inl a) i
```

The map A□4-A○□ is defined analogously. Using these maps we can define the first map between the total pushouts:

```
A□○-A○□ : A□○ → A○□
A□○-A○□ (inl x) = A□0-A○□ x
A□○-A○□ (inr x) = A□4-A○□ x
A□○-A○□ (push (inl x) i) = inl (push x i)
A□○-A○□ (push (inr x) i) = inr (push x i)
A□○-A○□ (push (push a j) i) = {- ??? -}
```

Note how we did not define a map A□2-A○□ to handle the push case, but instead proceeded via nested pattern matching. The reason is that when Agda checks boundary constraints, it does not perform η-expansion on its own so we have to do it by hand in the function definition.

Most cases boil down to swapping the constructors, but this straightforward strategy will not be sufficient to handle the last case. To fill this hole, we need to construct a square inside A○□ with a prescribed boundary. The most natural guess would be push (push a i) j, but its boundary for i = i0 is push (inl (f21 a)) j, which is not definitionally equal to the prescribed A□0-A○□(f□1 (push a j)).

In order to get the proper square, we will use hcomp to glue the four squares $s1$–$s4$ around our candidate in order to ensure it matches the dashed boundary:

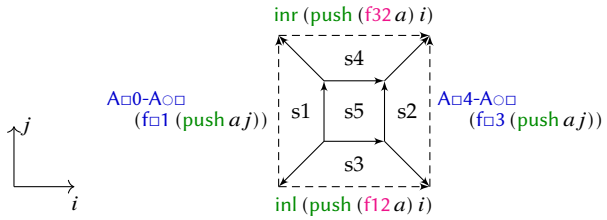


Figure 8.13: Using a two-dimensional hcomp to rectify the boundary of push (push a i) j

Thus we now explain how to construct the squares $s1$–$s5$, in this order.

To construct $s1$, we start by simplifying the boundary condition A□0-A○□ (f□1 (push a j)), which reduces to A□0-A○□ (hcomp ...). Function application commutes with hcomp's up to a homotopy, so A□0-A○□ applied to the composition is homotopic to the composition of the images of the paths. That is, we can find a square with the following boundary:

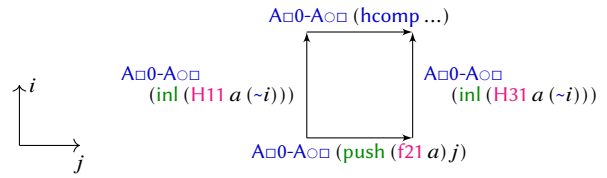


Figure 8.14: Constructing the square $s1$

Once rotated counter-clockwise, we can use this square for $s1$. We proceed similarly for the square $s2$. For the three remaining squares, we are left with the following boundary:

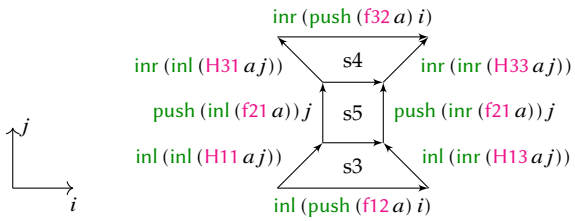


Figure 8.15: Remaining boundary

Just like before, we can use the functoriality of inl to get a square with the following boundary:

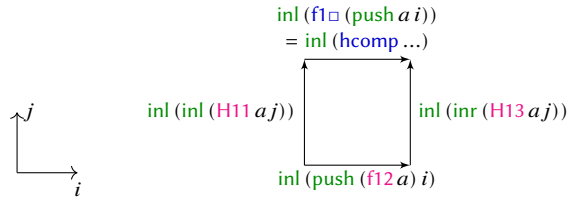


Figure 8.16: The square $s3$

We use this square for $s3$ and the appropriate analogue for $s4$. The only square left to construct now is

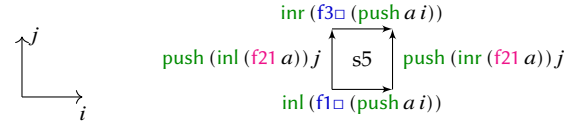


Figure 8.17: Remaining boundary for $s5$

which we can fill with push (push a i) j. By combining these five squares using hcomp we can finally complete our definition of A□○ → A○□.

The opposite direction, A○□ → A□○, is the same up to a transposition of the 3 × 3 span. The proofs that these maps cancel are also similar, but the central fillers are more difficult to illustrate as they require one more dimension. We refer the interested reader to the formalization. By combining all of this we obtain the desired equality between A□○ and A○□. □

The formal proof of the 3 × 3 lemma is under 200 lines of code (LOC) in Cubical Agda. The corresponding result in `HoTT-Agda` is about 3000 LOC. These numbers should of course be taken with a grain of salt as those proofs are not self-contained and rely on other results in the library. Regardless, we believe that it is a reasonably good illustration of how having HITs with definitional computation rules simplifies the path algebra.

This number has been calculated by counting the number of lines (excluding comments) in: `https://github.com/HoTT/HoTT-Agda/tree/master/theorems/homotopy/3x3`

8.4.3 The Join and S^3

Another interesting example of a pushout is the join. The main role of this construction in this chapter is to give us a presentation of S^3 as the join of two circles, which will be useful when proving that the total space of the Hopf fibration is S^3. However, this construction also has many other interesting uses in HoTT as explored by Rijke [85].

[85]: Rijke (2017), "The join construction"

The join of two types A and B is built from one copy of A, one copy of B, and a path from every $a : A$ to every $b : B$.

```
data Join (A : Set) (B : Set) : Set where
  inl : A → Join A B
  inr : B → Join A B
  push : ∀ a b → inl a ≡ inr b
```

It is easy to see that Join A B can alternatively be defined as the pushout of the following span—the inductive definitions only differ by currying.

$$A \xleftarrow{\text{fst}} A \times B \xrightarrow{\text{snd}} B$$

We now proceed by proving a few lemmas relating joins and spheres.

Lemma 8.4.4 Join Bool $A \equiv$ Susp A.

Proof. In Join Bool A the two points inr true and inr false play the role of north and south, with a path connecting them to every point inl a. □

The proof of the following result is taken from Brunerie [18, Proposition 1.8.6], as such we will only sketch it here.

[18]: Brunerie (2016), "On the homotopy groups of spheres in homotopy type theory"

Lemma 8.4.5 Join *is associative:*

$$\text{Join } A \text{ (Join } B \text{ } C) \equiv \text{Join (Join } A \text{ } B) \text{ } C$$

Proof. One starts by applying the 3×3 lemma to the following diagram.

$$\begin{array}{ccccc}
A & \longleftarrow & A \times B & \longrightarrow & B \\
\uparrow & & \uparrow & & \uparrow \\
A \times C & \longleftarrow & A \times B \times C & \longrightarrow & B \times C \\
\downarrow & & \downarrow & & \downarrow \\
A \times C & \longleftarrow & A \times C & \longrightarrow & C
\end{array}$$

All of the arrows in the diagram are the obvious projections, and the homotopies are refl's. One can then show that A□○ ≡ Join (Join A B) C, and that A○□ ≡ Join A (Join B C), which implies the desired result. □

We now have all of the ingredients to relate S^3 and the Join of two circles.

Lemma 8.4.6 Join S^1 $S^1 \equiv S^3$

Proof. This is a composition of equalities we already proved.

$$\begin{aligned}
\mathsf{Join}\ \mathbb{S}^1\ \mathbb{S}^1 &\equiv \mathsf{Join}\ (\mathsf{Susp}\ \mathsf{Bool})\ \mathbb{S}^1 && (8.4.1)\\
&\equiv \mathsf{Join}\ (\mathsf{Join}\ \mathsf{Bool}\ \mathsf{Bool})\ \mathbb{S}^1 && (8.4.4)\\
&\equiv \mathsf{Join}\ \mathsf{Bool}\ (\mathsf{Join}\ \mathsf{Bool}\ \mathbb{S}^1) && (8.4.5)\\
&\equiv \mathsf{Susp}\ (\mathsf{Susp}\ \mathbb{S}^1) && (8.4.4)\\
&\equiv \mathbb{S}^3 && \square
\end{aligned}$$

The above proof is an elegant application of the 3×3 lemma, but it results in rather complicated maps between $\mathsf{Join}\ \mathbb{S}^1\ \mathbb{S}^1$ and $\mathbb{S}^3$. Evan Cavallo has found a more direct cubical proof that $\mathsf{Join}\ \mathbb{S}^1\ \mathbb{S}^1 \equiv \mathbb{S}^3$ by simply defining the maps directly and proving that they cancel. We have ported his proof to Cubical Agda and the resulting proof is very short (~ 60 LOC). However, the proofs that the maps cancel require a 4-dimensional(!) hcomp making it rather difficult to visualize.

For details see the **redtt** proof at: https://github.com/RedPRL/redtt/blob/master/library/cool/s3-to-join.red

8.5 The Hopf Fibration

In this section we define the Hopf fibration $\mathsf{Hopf} : \mathbb{S}^2 \to \mathsf{Set}$. The Hopf fibration is a dependent type over the sphere $\mathbb{S}^2$ such that $\mathsf{Hopf}\ \mathsf{base}_2$ is equal the circle $\mathbb{S}^1$ and such that the total space $\Sigma\ (x : \mathbb{S}^2)\ .\ \mathsf{Hopf}\ x$ is equal to $\mathbb{S}^3$. This construction is useful because it allows us to compute homotopical properties of $\mathbb{S}^3$ from the properties of $\mathbb{S}^2$ and $\mathbb{S}^1$.

Fibrations are a concept from classical topology that encode a space that varies continuously over a base space. In homotopy/cubical type theory, a fibration corresponds to a dependent type.

We will define the Hopf fibration on $\mathsf{Susp}\ \mathbb{S}^1$ instead of $\mathbb{S}^2$, which does not really matter since both types are equal by lemma 8.4.2. Now, recall that $\mathsf{Susp}\ \mathbb{S}^1$ is the pushout of the span

$$1 \longleftarrow \mathbb{S}^1 \longrightarrow 1$$

Therefore, to define the function $\mathsf{Hopf} : \mathsf{Susp}\ \mathbb{S}^1 \to \mathsf{Set}$ we must pick the image of the two poles, which will be $\mathbb{S}^1$, and then give a proof of $\mathbb{S}^1 \equiv \mathbb{S}^1$ indexed by $\mathbb{S}^1$. By univalence, this amounts to defining a function $f : \mathbb{S}^1 \to \mathbb{S}^1 \to \mathbb{S}^1$ such that $f\,x$ is an equivalence for all $x : \mathbb{S}^1$.

Now it happens that $\mathbb{S}^1$ has a natural group structure. This is easy to see if we think of $\mathbb{S}^1$ as the set of unitary complex numbers:

$$e^{2i\pi\theta} \cdot e^{2i\pi\varphi} = e^{2i\pi(\theta+\varphi)}$$

But of course, in cubical type theory $\mathbb{S}^1$ is defined as a HIT and not as the set of unitary complex numbers. The correct analogue of the complex product can be defined by pattern-matching as follows:

```
rot : S¹ → S¹ → S¹
rot base y = y
rot (loop j) base = loop j
rot (loop j) (loop k) =
  hcomp (λ l → λ { (k = i0) → loop (j ∨ ~ l)
                 ; (k = i1) → loop (j ∧ l)
                 ; (j = i0) → loop (k ∨ ~ l)
                 ; (j = i1) → loop (k ∧ l)} ) base
```

The last case can be illustrated by the following diagram, where we only annotated the edges and reduced the (constant) central face to a tiny square:

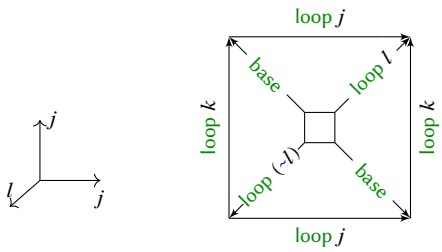


Figure 8.18: Defining the product on the unit circle in cubical type theory

Indeed, our intuition from the complex numbers tells us that rot (loop j) (loop k) is analogous to $e^{2i\pi(j+k)}$. So it is only natural that the $j + k = 1$ diagonal is constant at base, and the $j = k$ diagonal follows loop twice.

This results in a binary operation on S¹, that we also write as _*_ instead of rot. We can even get a complete (higher) abelian group structure on S¹, with an inverse operation that we call inv.

Lemma 8.5.1 *For every x : S¹ we have an equivalence* rotEquiv x : S¹ ≃ S¹ *given by* rot x.

Proof. The inverse of rot x is given by rot (inv x). □

We can now define the Hopf fibration using ua to convert rot into an equality:

```
Hopf : Susp S¹ → Set
Hopf north = S¹
Hopf south = S¹
Hopf (merid x i) = ua (rotEquiv x) i
```

In fact, we could have directly defined the Hopf fibration for S² using Glue to glue the identity equivalence on three of the sides of the 2-cell and the rot equivalence on the fourth side.

```
HopfS² : S² → Set
HopfS² base = S¹
HopfS² (surf i j) =
  Glue S¹ (λ { (i = i0) → (S¹ , idEquiv S¹)
             ; (i = i1) → (S¹ , idEquiv S¹)
             ; (j = i0) → (S¹ , idEquiv S¹)
             ; (j = i1) → (S¹ , rotEquiv (loop i)) } )
```

However, it turns out that the version using Susp S¹ is easier to work with as we have more wiggle room with the constructors.

We can now form the total space of the Hopf fibration using a Σ-type: Σ Hopf = $\sum_{x:\text{Susp } S^1}$ Hopf x. Our main theorem can be stated as:

Theorem 8.5.2 *The total space of the Hopf fibration is* S^3*, that is,* $\Sigma\,\mathrm{Hopf} \equiv S^3$.

Proof. We start by remarking that the space $\Sigma\,\mathrm{Hopf}$ is the homotopical pushout of the following span:

$$S^1 \xleftarrow{\pi_2} S^1 \times S^1 \xrightarrow{\mathrm{rot}} S^1$$

which can be seen by a straightforward unfolding of definitions. And with a change of variables, we can see $\Sigma\,\mathrm{Hopf}$ as the homotopical pushout of this alternative span, where $\mathrm{rot'}\ (x, y) = \mathrm{rot}\ (\mathrm{inv}\ x)\ y$.

This change of variables could be avoided if we tailored our definitions a bit better for this proof. This sort of irritating details is the downside of working with bare cubical primitives instead of universal properties.

$$S^1 \xleftarrow{\mathrm{rot'}} S^1 \times S^1 \xrightarrow{\pi_2} S^1$$

Now by lemma 8.4.6, which states that $S^3 \equiv \mathrm{Join}\ S^1\ S^1$, it suffices to prove that $\Sigma\,\mathrm{Hopf} \equiv \mathrm{Join}\ S^1\ S^1$. But recall that $\mathrm{Join}\ S^1\ S^1$ is the pushout of the following span:

$$S^1 \xleftarrow{\pi_1} S^1 \times S^1 \xrightarrow{\pi_2} S^1$$

To show that these two homotopical pushouts are isomorphic, we will describe a span isomorphism (that is, a natural equivalence between these two spans), and translate it to a Cubical Agda term.

To go from $\mathrm{Join}\ S^1\ S^1$ to $\Sigma\,\mathrm{Hopf}$, we want to use the following natural equivalence:

$$\begin{array}{ccccc}
S^1 & \xleftarrow{\pi_1} & S^1 \times S^1 & \xrightarrow{\pi_2} & S^1 \\
\downarrow{\scriptstyle \mathrm{id}} & & \downarrow{\scriptstyle (x,y) \mapsto (\mathrm{rot'}\ x\ y, y)} & & \downarrow{\scriptstyle \mathrm{id}} \\
S^1 & \xleftarrow[\mathrm{rot'}]{} & S^1 \times S^1 & \xrightarrow[\pi_2]{} & S^1
\end{array}$$

which can be written as a map between pushouts as follows:

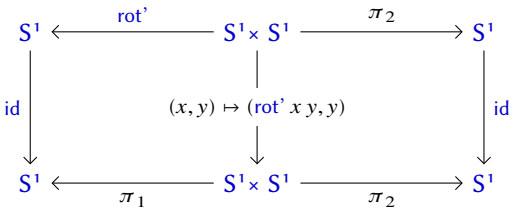
```
j2h : Join S¹ S¹ → ΣHopf
j2h (inl x) = (north , x)
j2h (inr y) = (south , y)
j2h (push x y i) = (merid (rot' (x , y)) i , {- ??? -})
```

Now our diagram says that the hole should be filled with y, but the types do not match: y has type S^1 while the hole expects a term of type $\mathrm{Glue}\ S^1\ (\lambda\ \{\ (i = \mathrm{i0}) \to (S^1, \mathrm{rotEquiv}\ (\mathrm{rot'}\ (x , y)))\ ;\ (i = \mathrm{i1}) \to (S^1, \mathrm{idEquiv}\ S^1)\ \})\ i$. To convert y into an inhabitant of this Glue type, we can use the glue primitive as follows:

```
let p : rot' (x , y) * x ≡ y
    p = lem-rot' x y
in glue (λ { (i = i0) → x ; (i = i1) → y }) (p i)
```

We supply it with a term $p\ i$ of type S^1, as well as a partial term which coincides with $p\ i$ after application of the equivalences rotEquiv and idEquiv. The auxiliary lemma lem-rot' $x\ y$ is defined by induction on x and y.

Conversely, to go from Σ Hopf to Join S^1 S^1, we use the following natural isomorphism:



To do this we need to map out of a Glue type. This is made possible through the unglue primitive in Cubical Agda. Given y : ua (rotEquiv x) i the term unglue $(i \vee \sim i)\ y$ is an element of S^1 that is x * y when i is i0 and y when i is i1. Using this we can write the inverse map.

```
h2j : ΣHopf → Join S¹ S¹
h2j (north , y) = inl y
h2j (south , y) = inr y
h2j (merid x i , y) =
  hcomp (λ j → λ { (i = i0) → inl (lem-rot-inv x y j)
                 ; (i = i1) → inr y })
        (push (unglue (i ∨ ~ i) y * inv x)
              (unglue (i ∨ ~ i) y) i)
```

The lemma lem-rot-inv proves that x * y * inv $x \equiv y$ for all $x, y : S^1$. Now that we have both directions of our equivalence, it remains to prove that they cancel. This requires some rather involved path algebra, and we refer the interested reader to the formalization. □

8.6 Comparison with Axiomatic HoTT

We have in this chapter shown how some of the main results in synthetic homotopy theory can be formalized in cubical type theory. We have used a variation of cubical type theory implemented by the Cubical Agda system, however it would have been possible to formalize all of these examples with comparable complexity in other cubical systems. Indeed, some of these examples have also been formalized in the **redtt** system [29] and in the precursor of Cubical Agda called cubicaltt [28].

One might wonder to what extent the lack of reversals and connections in **redtt**, which is based on cartesian cubical type theory [26, 27], affects the length of proofs. In our experience the lack of this additional structure on the interval is often made up for by the more powerful composition operations of cartesian cubical type theory. For instance, the direct proof that Join S^1 $S^1 \equiv S^3$ is of more or less exactly the same complexity as the Cubical Agda proof. However, in order to draw any definite conclusions more experiments are necessary.

[29]: The RedPRL Development Team (2018), *The **redtt** Proof Assistant*

[28]: Cohen et al. (2015), "Cubicaltt"

[26]: Angiuli et al. (2018), "Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities"

[27]: Angiuli et al. (2019), "Syntax and Models of Cartesian Cubical Type Theory"

The results in this chapter have also been formalized in the axiomatic HoTT libraries available in the major proof assistants based on type theory: `HoTT-Agda` [86], `Coq-HoTT` [62], `Lean-HoTT` [87]. It is very difficult to make an accurate quantitative comparison of the complexity between the formalized results as they have been performed in different systems based on different type theories. However, it is interesting to note that all of these HoTT libraries contain cubical sublibraries for conveniently reasoning about squares and cubes inspired by the work of Licata et al. [80]. In a cubical system like Cubical Agda we do not need to write such a library as the cubical primitives provide us with it for free.

We omit the `UniMath` and `Agda-Unimath` libraries as they do not focus on synthetic homotopy theory.

[80]: Licata et al. (2015), "A Cubical Approach to Synthetic Homotopy Theory"

Despite the difficulty of comparing the complexity of formal proofs between different systems we have made some estimates of the size of some of the proofs (in terms of lines of code) in table 8.1. Some of the libraries contain multiple proofs of the relevant results and in such cases we picked the one that most closely resemble the cubical proof. We only include those examples where we can make reasonably accurate estimates of the proof size, but the numbers in the table should still be taken with a large grain of salt as they do not count self-contained proofs and many of the results rely on cubical sublibraries that are not necessary in Cubical Agda. The line count also involve relevant comments and we haven't counted the definitions of the involved HITs. The length and style of proofs also vary quite a bit between the various systems in general, for instance, both Coq and Lean proofs are written using tactics while Agda proofs are typically not.

The numbers in the table have been computed from the `master` branches of the libraries on 2019-12-16. TODO: update?

Table 8.1: Results in the HoTT libraries

	`HoTT-Agda`	`Coq-HoTT`	`Lean-HoTT`	Cubical Agda
$\Omega(\mathbb{S}^1) = \mathbb{Z}$	90	160	80	50
$\mathbb{T} = \mathbb{S}^1 \times \mathbb{S}^1$	150	150	-	25
3×3 lemma	3000	-	-	200
Join assoc. via 3×3	320	-	-	240
Join assoc. direct	210	-	230	90

The table indicates that not too much is gained in the proof that the loop space of the circle is the integers by doing it cubically, while the proof that the torus is equivalent to two circles is about 6 times longer in `HoTT-Agda` and `Coq-HoTT` compared to the cubical proof presented here. The major difference is for the 3×3 lemma for pushouts where the `HoTT-Agda` proof is about 15 times longer than the cubical proof. This result is not yet formalized in `Coq-HoTT` and `Lean-HoTT`, however a `Coq-HoTT` formalization is currently underway. The `HoTT-Agda` library has two proofs that Join is associative. The longer one, 320 lines, is more or less the same as the one we discussed in this paper and relies on the 3×3 lemma for pushouts. There is very little gain from the cubical machinery in this proof as it is simply a matter of reorganizing data so that we can apply an already proved result. The other `HoTT-Agda` and `Lean-HoTT` proofs are more direct and construct the maps in an ingenious way following Cavallo [75, Theorem 4.21]. Evan Cavallo has recently proved this result directly in Cubical Agda as well, leading to a proof in only 90 lines of code.

[75]: Cavallo (2015), "Synthetic cohomology in homotopy type theory"

Another interesting observation is that when working in a cubical system we often state the theorems as paths while in HoTT one often

instead just uses equivalences. These are of course equivalent by univalence, but by invoking the univalence axiom in HoTT one hides the computational content of these equivalences making them harder to work with. In a cubical system where univalence has computational content this is not the case and it is in fact often more convenient to convert the equivalences into paths using univalence as we may then use the cubical primitives to manipulate them. This indicates that cubical type theory might be better suited for doing *univalent* mathematics than axiomatic HoTT.

Appendix

A

Inference Rules of $\mathrm{CC}^{\mathrm{obs}}$

A.1 Syntactic Sugar

$$\begin{array}{lcl} \mathsf{ap} & : & \Pi(f : A \to B).\, x \sim_A y \to f\,x \sim_B f\,y \\ \mathsf{ap}\,f\,e & := & \mathsf{transp}(A, x, \lambda z\,.\, f\,x \sim_B f\,z, \mathsf{refl}, y, e) \end{array}$$

$$\begin{array}{lcl} \_^{-1} & : & x \sim_A y \to y \sim_A x \\ e^{-1} & := & \mathsf{transp}(A, x, \lambda z\,.\, z \sim_A x, \mathsf{refl}, y, e) \end{array}$$

$$\begin{array}{lcl} \_ \cdot \_ & : & x \sim_A y \to y \sim_A z \to x \sim_A z \\ e \cdot e' & := & \mathsf{transp}(A, y, \lambda t\,.\, x \sim_A t, e, z, e') \end{array}$$

$$\begin{array}{lcl} (x : \_)\ \&\ \_ & : & \Pi(A : \Omega).\, (A \to \Omega) \to \Omega \\ (x : A)\ \&\ B & := & \Pi(X : \Omega).\, (\Pi(x : A).\, B \to X) \to X \end{array}$$

$$\begin{array}{lcl} \langle \_, \_ \rangle & : & \Pi(a : A).\, B \to (a : A)\ \&\ B \\ \langle a, b \rangle & := & \lambda\, X\, H\,.\, H\, a\, b \end{array}$$

Given a term t of type $(x : A)\ \&\ B$:

$$\begin{array}{lcllcl} \mathsf{fst}(t) & := & t\,A\,(\lambda\, a\, b\,.\, a) & : & A \\ \mathsf{snd}(t) & := & t\,(B\,\mathsf{fst}(t))\,(\lambda\, a\, b\,.\, b) & : & B\,\mathsf{fst}(t) \end{array}$$

$$\begin{array}{lcl} \exists(x : \_).\, \_ & : & \Pi(A : \mathscr{U}_i).\, (A \to \Omega) \to \Omega \\ \exists(x : A).\, B & := & \Pi(X : \Omega).\, (\Pi(x : A).\, B \to X) \to X \end{array}$$

$$\begin{array}{lcllcl} \top & := & \bot \to \bot & : & \Omega \\ * & := & \lambda(x : \bot).\, x & : & \top \end{array}$$

$$\begin{array}{lcl} \|\_\| & : & \mathscr{U}_i \to \Omega \\ \|A\| & := & \Pi(X : \Omega).\, (A \to X) \to X \end{array}$$

Given a term t of type A:

$$|t| \quad := \quad \lambda\, X\, H\, .\, H\, t \quad : \quad \|A\|$$

A.2 Contexts and Conversion

$$\textsc{Ctx-Nil}\ \frac{}{\vdash \bullet} \qquad \textsc{Ctx-Cons}\ \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathfrak{s}}{\vdash \Gamma, x : A : \mathfrak{s}} \qquad \textsc{Var}\ \frac{\vdash \Gamma \qquad x : A : \mathfrak{s} \in \Gamma}{\Gamma \vdash x : A : \mathfrak{s}} \qquad \textsc{conv}\ \frac{\Gamma \vdash t : A : \mathfrak{s} \qquad \Gamma \vdash A \equiv B : \mathfrak{s}}{\Gamma \vdash t : B : \mathfrak{s}}$$

$$\textsc{Refl}\ \frac{\Gamma \vdash t : A : \mathfrak{s}}{\Gamma \vdash t \equiv t : A : \mathfrak{s}} \qquad \textsc{Sym}\ \frac{\Gamma \vdash t \equiv u : A : \mathfrak{s}}{\Gamma \vdash u \equiv t : A : \mathfrak{s}} \qquad \textsc{Trans}\ \frac{\Gamma \vdash t \equiv t' : A : \mathfrak{s} \qquad \Gamma \vdash t' \equiv u : A : \mathfrak{s}}{\Gamma \vdash t \equiv u : A : \mathfrak{s}}$$

$$\textsc{Red-Conv}\ \frac{\Gamma \vdash t \Rightarrow u : A : \mathscr{U}_i}{\Gamma \vdash t \equiv u : A : \mathscr{U}_i} \qquad \textsc{Proof-Irrelevance}\ \frac{\Gamma \vdash t, u : A : \Omega}{\Gamma \vdash t \equiv u : A : \Omega}$$

A.3 Proof-Irrelevant Types

A.3.1 Impredicative Π-Types

$$\Pi\textsc{-Irr-Form}\ \frac{\Gamma, x : A : \mathfrak{s} \vdash B : \Omega}{\Gamma \vdash \Pi^{\mathfrak{s},\Omega}(x : A).\, B : \Omega} \qquad \textsc{Fun-Irr}\ \frac{\Gamma, x : A : \mathfrak{s} \vdash t : B : \Omega}{\Gamma \vdash \lambda(x : A).\, t : \Pi(x : A).\, B : \Omega} \qquad \textsc{App-Irr}\ \frac{\Gamma \vdash t : \Pi(x : A).\, B : \Omega \qquad \Gamma \vdash u : A : \mathfrak{s}}{\Gamma \vdash t\, u : B[x := u] : \Omega}$$

A.3.2 False Proposition

$$\bot\textsc{-Form}\ \frac{\vdash \Gamma}{\Gamma \vdash \bot : \Omega} \qquad \bot\textsc{-Elim}\ \frac{\Gamma \vdash A : \mathfrak{s} \qquad \Gamma \vdash t : \bot : \Omega}{\Gamma \vdash \bot\text{-elim}(A, t) : A : \mathfrak{s}}$$

A.3.3 The Observational Equality

$$\textsc{Eq-Form}\ \frac{\Gamma \vdash t : A : \mathscr{U}_i \qquad \Gamma \vdash u : A : \mathscr{U}_i}{\Gamma \vdash t \sim_A u : \Omega} \qquad \textsc{Refl}\ \frac{\Gamma \vdash t : A : \mathscr{U}_i}{\Gamma \vdash \mathsf{refl}(t) : t \sim_A t : \Omega}$$

$$\textsc{Transport-}\Omega\ \frac{\Gamma \vdash t, t' : A : \mathscr{U}_i \quad \Gamma, x : A \vdash B : \Omega \qquad \Gamma \vdash u : B[x := t] : \Omega \quad \Gamma \vdash e : t \sim_A t' : \Omega}{\Gamma \vdash \mathsf{transp}(A, t, B, u, t', e) : B[x := t'] : \Omega}$$

$$\textsc{Cast}\ \frac{\Gamma \vdash e : A \sim_{\mathfrak{s}} B : \Omega \quad \Gamma \vdash t : A : \mathfrak{s}}{\Gamma \vdash \mathsf{cast}(A, B, e, t) : B : \mathfrak{s}} \qquad \textsc{Cast-Refl}\ \frac{\Gamma \vdash t : A : \mathscr{U}_i \qquad \Gamma \vdash e : A \sim_{\mathscr{U}_i} A : \Omega}{\Gamma \vdash \mathsf{castrefl}(A, t) : t \sim_A \mathsf{cast}(A, A, e, t) : \Omega}$$

A.4 Proof-Relevant Types

A.4.1 Dependent products

Π-Rel-Form
$$\frac{\Gamma, x : A : \mathfrak{s} \vdash B : \mathcal{U}_i}{\Gamma \vdash \Pi^{\mathfrak{s},\mathcal{U}_i}(x : A).\, B : \mathcal{U}_{\max(i,\mathfrak{s})}}$$

Fun-Rel
$$\frac{\Gamma, x : A : \mathfrak{s} \vdash t : B : \mathcal{U}_i}{\Gamma \vdash \lambda(x : A).\, t : \Pi^{\mathfrak{s},\mathcal{U}_i}(x : A).\, B : \mathcal{U}_{\max(i,\mathfrak{s})}}$$

App-Rel
$$\frac{\Gamma \vdash t : \Pi^{\mathfrak{s},\mathcal{U}_i}(x : A).\, B : \mathcal{U}_{\max(i,\mathfrak{s})} \qquad \Gamma \vdash u : A : \mathfrak{s}}{\Gamma \vdash t\, u : B[x := u] : \mathcal{U}_i}$$

β-red
$$\frac{\Gamma, x : A : \mathfrak{s} \vdash t : B : \mathcal{U}_i \qquad \Gamma \vdash u : A : \mathfrak{s}}{\Gamma \vdash (\lambda(x : A).\, t)\, u \Rightarrow t[x := u] : B[x := u] : \mathcal{U}_i}$$

η-Eq
$$\frac{\Gamma \vdash t, u : \Pi^{\mathfrak{s},\mathcal{U}_i}(x : A).\, B : \mathcal{U}_{\max(i,\mathfrak{s})} \qquad \Gamma, x : A : \mathfrak{s} \vdash t\, x \equiv u\, x : B : \mathcal{U}_i}{\Gamma \vdash t \equiv u : \Pi^{\mathfrak{s},\mathcal{U}_i}(x : A).\, B : \mathcal{U}_{\max(i,\mathfrak{s})}}$$

Eq-Fun
$$\frac{\Gamma \vdash f, g : \Pi^{\mathfrak{s},\mathcal{U}_i}(x : A).\, B : \mathcal{U}_{\max(i,\mathfrak{s})}}{\Gamma \vdash f \sim_{\Pi AB} g \Rightarrow \Pi^{\mathfrak{s},\Omega}(x : A).\, f\, x \sim_B g\, x : \Omega}$$

Eq-Π
$$\frac{\Gamma \vdash A, A' : \mathfrak{s} \qquad \Gamma, x : A \vdash B : \mathcal{U}_i \qquad \Gamma, x : A' \vdash B' : \mathcal{U}_i \qquad a := \mathsf{cast}(A', A, e, a')}{\Gamma \vdash \begin{array}{l}\Pi(x : A).\, B \sim_{\mathcal{U}} \Pi(x : A').\, B' \Rightarrow \\ (e : A' \sim_{\mathfrak{s}} A)\ \&\ \Pi(a' : A').\, B[x := a] \sim_{\mathcal{U}} B'[x := a']\end{array} : \Omega}$$

Cast-Π
$$\frac{\Gamma \vdash e : \Pi(x : A).\, B \sim_{\mathcal{U}} \Pi(x : A').\, B' \qquad \Gamma \vdash f : \Pi(x : A).\, B : \mathcal{U} \qquad a := \mathsf{cast}(A', A, \mathsf{fst}(e), a')}{\Gamma \vdash \begin{array}{l}\mathsf{cast}(\Pi(x : A).\, B, \Pi(x : A').\, B', e, f) \Rightarrow \\ \lambda(a' : A').\, \mathsf{cast}(B[x := a], B'[x := a'], \mathsf{snd}(e)\, a', f\, a)\end{array} : \Pi(x : A').\, B' : \mathcal{U}_i}$$

A.4.2 Dependent sums

Σ-Form
$$\frac{\Gamma, x : A : \mathcal{U}_i \vdash B : \mathcal{U}_j}{\Gamma \vdash \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}$$

Pair
$$\frac{\Gamma, x : A : \mathcal{U}_i \vdash B : \mathcal{U}_j \qquad \Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash u : B[x := t] : \mathcal{U}_j}{\Gamma \vdash \langle t\,;\, u\rangle : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}$$

Fst
$$\frac{\Gamma \vdash t : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}{\Gamma \vdash \mathsf{proj}_1(t) : A : \mathcal{U}_i}$$

Snd
$$\frac{\Gamma \vdash t : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}{\Gamma \vdash \mathsf{proj}_2(t) : B[x := \mathsf{proj}_1(t)] : \mathcal{U}_j}$$

Fst-Pair
$$\frac{\Gamma, x : A : \mathcal{U}_i \vdash B : \mathcal{U}_j \qquad \Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash u : B[x := t] : \mathcal{U}_j}{\Gamma \vdash \mathsf{proj}_1(\langle t\,;\, u\rangle) \Rightarrow t : A : \mathcal{U}_i}$$

Snd-Pair
$$\frac{\Gamma, x : A : \mathcal{U}_i \vdash B : \mathcal{U}_j \qquad \Gamma \vdash t : A : \mathcal{U}_i \qquad \Gamma \vdash u : B[x := t] : \mathcal{U}_j}{\Gamma \vdash \mathsf{proj}_2(\langle t\,;\, u\rangle) \Rightarrow u : B[x := t] : \mathcal{U}_j}$$

η-Depsum
$$\frac{\Gamma \vdash t : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}{\Gamma \vdash t \equiv \langle \mathsf{proj}_1(t)\,;\, \mathsf{proj}_2(t)\rangle : \Sigma^{\mathcal{U}_i,\mathcal{U}_j}(x : A).\, B : \mathcal{U}_{\max(i,j)}}$$

Eq-Pair
$$\frac{\Gamma \vdash t, u : \Sigma^{\mathscr{U}_i,\mathscr{U}_j}(x : A).\, B : \mathscr{U}_{\max(i,j)} \qquad b' := \mathsf{cast}(B[x := \mathsf{proj}_1(t)], B[x := \mathsf{proj}_1(u)], \mathsf{ap}\ B\ e, \mathsf{proj}_2(t))}{\Gamma \vdash \begin{array}{l} t \sim_{\Sigma AB} u \Rightarrow \\ (e : \mathsf{proj}_1(t) \sim_A \mathsf{proj}_1(u)) \,\&\, (b' \sim_B \mathsf{proj}_2(u)) \end{array} : \Omega}$$

Eq-Σ
$$\frac{\Gamma \vdash A, A' : \mathbb{s} \qquad \Gamma, x : A \vdash B : \mathscr{U}_i \qquad \Gamma, x : A' \vdash B' : \mathscr{U}_i \qquad a' := \mathsf{cast}(A, A', e, a)}{\Gamma \vdash \begin{array}{l} \Sigma(x : A).\, B \sim_{\mathscr{U}} \Sigma(x : A').\, B' \Rightarrow \\ (e : A \sim_{\mathbb{s}} A') \,\&\, \Pi(a : A).\, B[x := a] \sim_{\mathscr{U}} B'[x := a'] \end{array} : \Omega}$$

Cast-Σ
$$\frac{\Gamma \vdash e : \Sigma(x : A).\, B \sim_{\mathscr{U}} \Sigma(x : A').\, B' \qquad \Gamma \vdash t : \Sigma(x : A).\, B : \mathscr{U} \qquad a' := \mathsf{cast}(A, A, \mathsf{fst}(e), \mathsf{proj}_1(t))}{\Gamma \vdash \begin{array}{l} \mathsf{cast}(\Sigma(x : A).\, B, \Sigma(x : A').\, B', e, t) \Rightarrow \\ \langle a' \,;\, \mathsf{cast}(B[x := \mathsf{proj}_1(t)], B'[x := a'], \mathsf{snd}(e), \mathsf{proj}_2(t)) \rangle \end{array} : \Sigma(x : A').\, B' : \mathscr{U}_i}$$

A.4.3 Natural Numbers

$$\text{ℕ-Form}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} : \mathscr{U}_0} \qquad \text{Zero}\ \frac{\vdash \Gamma}{\Gamma \vdash 0 : \mathbb{N} : \mathscr{U}_0} \qquad \text{Suc}\ \frac{\Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathsf{S}\ n : \mathbb{N} : \mathscr{U}_0}$$

ℕ-Elim
$$\frac{\Gamma \vdash A : \mathbb{N} \to \mathbb{s} \qquad \Gamma \vdash t_0 : A\ 0 : \mathbb{s} \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A\ n \to A\ (\mathsf{S}\ n) : \mathbb{s} \qquad \Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathbb{N}\text{–elim}(A, t_0, t_S, n) : A\ n : \mathbb{s}}$$

ℕ-Elim-Zero
$$\frac{\Gamma \vdash A : \mathbb{N} \to \mathscr{U}_i \qquad \Gamma \vdash t_0 : A\ 0 : \mathscr{U}_i \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A\ n \to A\ (\mathsf{S}\ n) : \mathscr{U}_i}{\Gamma \vdash \mathbb{N}\text{–elim}(A, t_0, t_S, 0) \Rightarrow t_0 : A\ 0 : \mathscr{U}_i}$$

ℕ-Elim-Suc
$$\frac{\Gamma \vdash A : \mathbb{N} \to \mathscr{U}_i \qquad \Gamma \vdash t_0 : A\ 0 : \mathscr{U}_i \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A\ n \to A\ (\mathsf{S}\ n) : \mathscr{U}_i \qquad \Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathbb{N}\text{–elim}(A, t_0, t_S, \mathsf{S}\ n) \Rightarrow t_S\ n\ \mathbb{N}\text{–elim}(A, t_0, t_S, n) : A\ (\mathsf{S}\ n) : \mathscr{U}_i}$$

$$\text{Eq-zero}\ \frac{\vdash \Gamma}{\Gamma \vdash 0 \sim_{\mathbb{N}} 0 \Rightarrow \top : \Omega} \qquad \text{Eq-Suc}\ \frac{\Gamma \vdash n : \mathbb{N} : \mathscr{U}_0 \qquad \Gamma \vdash m : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathsf{S}\ m \sim_{\mathbb{N}} \mathsf{S}\ n \Rightarrow m \sim_{\mathbb{N}} n : \Omega} \qquad \text{Eq-zero-suc}\ \frac{\Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash 0 \sim_{\mathbb{N}} \mathsf{S}\ n \Rightarrow \bot : \Omega}$$

$$\text{Eq-suc-zero}\ \frac{\Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathsf{S}\ n \sim_{\mathbb{N}} 0 \Rightarrow \bot : \Omega} \qquad \text{Eq-Nat}\ \frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} \sim_{\mathscr{U}} \mathbb{N} \Rightarrow \top : \Omega} \qquad \text{Cast-Zero}\ \frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathscr{U}_0} \mathbb{N}}{\Gamma \vdash \mathsf{cast}(\mathbb{N}, \mathbb{N}, e, 0) \Rightarrow 0 : \mathbb{N} : \mathscr{U}_0}$$

Cast-Suc
$$\frac{\Gamma \vdash e : \mathbb{N} \sim_{\mathscr{U}_0} \mathbb{N} \qquad \Gamma \vdash n : \mathbb{N} : \mathscr{U}_0}{\Gamma \vdash \mathsf{cast}(\mathbb{N}, \mathbb{N}, e, \mathsf{S}\ n) \Rightarrow \mathsf{S}\ \mathsf{cast}(\mathbb{N}, \mathbb{N}, e, n) : \mathbb{N} : \mathscr{U}_0}$$

A.4.4 Box Types

$$\text{Box-Form}\ \frac{\Gamma \vdash A : \Omega}{\Gamma \vdash \Box A : \mathscr{U}_0} \qquad \text{Box-Proof}\ \frac{\Gamma \vdash t : A : \Omega}{\Gamma \vdash \diamond t : \Box A : \mathscr{U}_0} \qquad \text{Box-Proof-Eq}\ \frac{\Gamma \vdash t, u : \Box A : \mathscr{U}_0}{\Gamma \vdash t \sim_{\Box A} u \Rightarrow \top : \Omega}$$

$$\text{Unbox}\ \frac{\Gamma \vdash A : \Omega \qquad \Gamma \vdash t : \Box A}{\Gamma \vdash \Box\text{–elim}(t) : A} \qquad \text{Box-Eq}\ \frac{\Gamma \vdash A, B : \Omega}{\Gamma \vdash \Box A \sim_{\mathscr{U}} \Box B \Rightarrow A \sim_{\Omega} B : \Omega}$$

UNBOX-BOX
$$\frac{\Gamma \vdash A, B : \Omega \qquad \Gamma \vdash t : A \qquad \Gamma \vdash e : \Box A \sim_{\mathscr{U}} \Box B}{\Gamma \vdash \mathsf{cast}(\Box A, \Box B, e, \diamond t) \Rightarrow \diamond \mathsf{cast}(A, B, e, t) : \Box B}$$

A.4.5 Quotients

QUOTIENT-FORM
$$\frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash R : A \to A \to \Omega : \mathscr{U}_i \qquad \Gamma \vdash R_r : \Pi(x : A).\, R\, x\, x : \Omega \qquad \Gamma \vdash R_s : \Pi(x, y : A).\, R\, x\, y \to R\, y\, x : \Omega \qquad \Gamma \vdash R_t : \Pi(x, y, z : A).\, R\, x\, y \to R\, y\, z \to R\, x\, z : \Omega}{\Gamma \vdash A/(R, R_r, R_s, R_t) : \mathscr{U}_i}$$

QUOTIENT-PROJ
$$\frac{\Gamma \vdash t : A : \mathscr{U}_i \qquad \Gamma \vdash R : A \to A \to \Omega : \mathscr{U}_i \qquad \mathsf{isrel}(R)}{\Gamma \vdash \pi(t) : A/R : \mathscr{U}_i}$$

QUOTIENT-PROJ-EQ
$$\frac{\Gamma \vdash t : A : \mathscr{U}_i \qquad \Gamma \vdash u : A : \mathscr{U}_i \qquad \Gamma \vdash R : A \to A \to \Omega : \mathscr{U}_i \qquad \mathsf{isrel}(R)}{\Gamma \vdash \pi(t) \sim_{A/R} \pi(u) \Rightarrow R\, t\, u : \Omega}$$

QUOTIENT-PROJ-CAST
$$\frac{\Gamma \vdash e : A/R \sim_{\mathscr{U}_i} A'/R' : \Omega \qquad \Gamma \vdash t : A : \mathscr{U}_i}{\Gamma \vdash \mathsf{cast}(A/R, A'/R', e, \pi(t)) \Rightarrow \pi(\mathsf{cast}(A, A', \mathsf{fst}(e), t)) : A/R' : \mathscr{U}_i}$$

QUOTIENT-EQ
$$\frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash R : A \to A \to \Omega \qquad \mathsf{isrel}(R) \qquad \Gamma \vdash A' : \mathscr{U}_i \qquad \Gamma \vdash R' : A' \to A' \to \Omega \qquad \mathsf{isrel}(R') \qquad x' := \mathsf{cast}(A, A', e, x) \qquad y' := \mathsf{cast}(A, A', e, y)}{\Gamma \vdash \begin{array}{l} A/R \sim_{\mathscr{U}_i} A'/R' \Rightarrow \\ (e : A \sim_{\mathscr{U}_i} A') \;\&\; \Pi(x\, y : A).\, R\, x\, y \sim_{\Omega} R\, x'\, y' \end{array} : \Omega}$$

QUOTIENT-ELIM
$$\frac{\Gamma \vdash B : A/R \to \mathfrak{s} \qquad \Gamma \vdash t_\pi : \Pi(x : A).\, B\, \pi(x) \qquad \Gamma \vdash t_\sim : \Pi(x, y : A).\, \Pi(e : R\, x\, y).\, (t_\pi\, x) \sim_{B\, \pi(x)} \mathsf{cast}(B\, \pi(y), B\, \pi(x), B\, e^{-1}, t_\pi\, y) \qquad \Gamma \vdash u : A/R}{\Gamma \vdash \mathsf{Q\text{-}elim}(B, t_\pi, t_\sim, u) : B\, u : \mathfrak{s}}$$

QUOTIENT-PROJ-ELIM
$$\frac{\Gamma \vdash B : A/R \to \mathscr{U}_i \qquad \Gamma \vdash t_\pi : \Pi(x : A).\, B\, \pi(x) \qquad \Gamma \vdash t_\sim : \Pi(x, y : A).\, \Pi(e : R\, x\, y).\, (t_\pi\, x) \sim_{B\, \pi(x)} \mathsf{cast}(B\, \pi(y), B\, \pi(x), B\, e^{-1}, t_\pi\, y) \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Q\text{-}elim}(B, t_\pi, t_\sim, \pi(u)) \Rightarrow t_\pi\, u : B\, (\pi(u)) : \mathscr{U}_i}$$

A.4.6 Inductive Equality

ID-FORM
$$\frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Id}(A, t, u) : \mathscr{U}_i}$$

IDREFL
$$\frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{Idrefl}(t) : \mathsf{Id}(A, t, t)}$$

IDPATH
$$\frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A \qquad \Gamma \vdash e : t \sim_A u}{\Gamma \vdash \mathsf{Idpath}(e) : \mathsf{Id}(A, t, u)}$$

J
$$\frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash B : \Pi(x : A).\, \mathsf{Id}(A, t, x) \to \mathfrak{s} \qquad \Gamma \vdash u : B\, t\, \mathsf{Idrefl}(t) \qquad \Gamma \vdash t' : A \qquad \Gamma \vdash e : \mathsf{Id}(A, t, t')}{\Gamma \vdash \mathsf{J}(A, t, B, u, t', e) : B\, t'\, e}$$

J-IDREFL
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash B : \Pi(x : A).\, \mathsf{Id}(A, t, x) \to \mathcal{U}_j \qquad \Gamma \vdash u : B\ t\ \mathsf{Idrefl}(t)}{\Gamma \vdash \mathsf{J}(A, t, B, u, t, \mathsf{Idrefl}(t)) \Rightarrow u : B\ t\ \mathsf{Idrefl}(t)}$$

ID-PROOF-EQ
$$\frac{\Gamma \vdash e, e' : \mathsf{Id}(A, t, u)}{\Gamma \vdash e \sim_{\mathsf{Id}(A,t,u)} e' \Rightarrow \top : \Omega}$$

ID-EQ
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A \qquad \Gamma \vdash A' : \mathcal{U}_i \qquad \Gamma \vdash t' : A' \qquad \Gamma \vdash u' : A'}{\Gamma \vdash \begin{array}{l} \mathsf{Id}(A, t, u) \sim_{\mathcal{U}} \mathsf{Id}(A', t', u') \Rightarrow \\ (e : A \sim_{\mathcal{U}} A')\ \&\ \mathsf{cast}(A, A', e, t) \sim_{A'} t' \wedge \mathsf{cast}(A, A', e, u) \sim_{A'} u' \end{array} : \Omega}$$

CAST-IDREFL
$$\frac{\Gamma \vdash A, A' : \mathcal{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash t', u' : A' \qquad \Gamma \vdash e : \mathsf{Id}(A, t, t) \sim \mathsf{Id}(A', t', u')}{\Gamma \vdash \begin{array}{l} \mathsf{cast}(\mathsf{Id}(A, t, t), \mathsf{Id}(A', t', u'), e, \mathsf{Idrefl}(t)) \Rightarrow \\ \mathsf{Idpath}(\mathsf{fst}(\mathsf{snd}(e))^{-1} \cdot \mathsf{snd}(\mathsf{snd}(e))) \end{array} : \mathsf{Id}(A', t', u')}$$

J-IDPATH
$$\frac{\begin{array}{c} \Gamma \vdash A : \mathcal{U}_i \\ \Gamma \vdash t : A \qquad \Gamma \vdash B : \Pi(x : A).\, \mathsf{Id}(A, t, x) \to \mathcal{U}_j \qquad \Gamma \vdash b : B\ t\ \mathsf{Idrefl}(t) \qquad \Gamma \vdash t' : A \qquad \Gamma \vdash e : t \sim_A t' \end{array}}{\Gamma \vdash \begin{array}{l} \mathsf{J}(A, t, B, b, t', \mathsf{Idpath}(e)) \Rightarrow \\ \mathsf{cast}(B\ t\ \mathsf{Idrefl}(t), B\ t'\ \mathsf{Idpath}(e), \mathsf{eq_J}(A, t, B, t', e), b) \end{array} : B\ t'\ \mathsf{Idpath}(e)}$$

CAST-IDPATH
$$\frac{\Gamma \vdash A, A' : \mathcal{U}_i \qquad \Gamma \vdash t, u : A \qquad \Gamma \vdash e : t \sim_A u \qquad \Gamma \vdash t', u' : A' \qquad \Gamma \vdash e' : \mathsf{Id}(A, t, t) \sim \mathsf{Id}(A', t', u')}{\Gamma \vdash \begin{array}{l} \mathsf{cast}(\mathsf{Id}(A, t, u), \mathsf{Id}(A', t', u'), e', \mathsf{Idpath}(e)) \Rightarrow \\ \mathsf{Idpath}(\mathsf{fst}(\mathsf{snd}(e'))^{-1} \cdot \mathsf{ap}\ (\mathsf{cast}(A, A', \mathsf{fst}(e'), -))\ e \cdot \mathsf{snd}(\mathsf{snd}(e'))) \end{array} : \mathsf{Id}(A', t', u')}$$

A.4.7 Universe of Propositions

UNIV-PROP
$$\frac{\vdash \Gamma}{\Gamma \vdash \Omega : \mathcal{U}_0}$$

EQ-Ω
$$\frac{\Gamma \vdash A : \Omega \qquad \Gamma \vdash B : \Omega}{\Gamma \vdash A \sim_\Omega B \Rightarrow (A \to B) \wedge (B \to A) : \Omega}$$

EQ-Ω
$$\frac{\vdash \Gamma}{\Gamma \vdash \Omega \sim_{\mathcal{U}_0} \Omega \Rightarrow \top : \Omega}$$

CAST-UNIV
$$\frac{\Gamma \vdash e : \Omega \sim_{\mathcal{U}_0} \Omega \qquad \Gamma \vdash A : \Omega}{\Gamma \vdash \mathsf{cast}(\Omega, \Omega, e, A) \Rightarrow A : \Omega}$$

A.4.8 Predicative Universe Hierarchy

UNIV-REL
$$\frac{\vdash \Gamma}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}$$

EQ-UNIV-≠
$$\frac{\vdash \Gamma \qquad \mathrm{hd}\, A \neq \mathrm{hd}\, B}{\Gamma \vdash A \sim_{\mathcal{U}} B \Rightarrow \bot : \Omega}$$

EQ-UNIV
$$\frac{\vdash \Gamma}{\Gamma \vdash \mathcal{U}_i \sim_{\mathcal{U}i+1} \mathcal{U}_i \Rightarrow \top : \Omega}$$

CAST-UNIV
$$\frac{\Gamma \vdash e : \mathcal{U}_i \sim_{\mathcal{U}i+1} \mathcal{U}_i \qquad \Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash \mathsf{cast}(\mathcal{U}_i, \mathcal{U}_i, e, A) \Rightarrow A : \mathcal{U}_i}$$

A.5 Congruence Rules

Π-Cong-Irr
$$\frac{\Gamma \vdash A \equiv A' : \mathfrak{s} \qquad \Gamma, x : A \vdash B \equiv B' : \Omega}{\Gamma \vdash \Pi(x : A).\, B \equiv \Pi(x : A').\, B' : \Omega}$$

⊥-Elim-Cong
$$\frac{\Gamma \vdash A \equiv A' : \mathscr{U}_i \qquad \Gamma \vdash t \equiv t' : \bot}{\Gamma \vdash \bot\text{-elim}(A, t) \equiv \bot\text{-elim}(A', t') : A}$$

Eq-Cong
$$\frac{\Gamma \vdash A \equiv A' : \mathscr{U}_i \qquad \Gamma \vdash t \equiv t' : A \qquad \Gamma \vdash u \equiv u' : A}{\Gamma \vdash t \sim_A u \equiv t' \sim_{A'} u' : \Omega}$$

Cast-Cong
$$\frac{\Gamma \vdash A \equiv A' : \mathscr{U}_i \qquad \Gamma \vdash B \equiv B' : \mathscr{U}_i \qquad \Gamma \vdash e \equiv e' : A \sim_{\mathscr{U}_i} B \qquad \Gamma \vdash t \equiv t' : A}{\Gamma \vdash \mathsf{cast}(A, B, e, t) \equiv \mathsf{cast}(A', B', e', t') : B}$$

Π-Cong-Rel
$$\frac{\Gamma \vdash A \equiv A' : \mathfrak{s} \qquad \Gamma, x : A \vdash B \equiv B' : \mathscr{U}_i}{\Gamma \vdash \Pi(x : A).\, B \equiv \Pi(x : A').\, B' : \mathscr{U}_{\max(\mathfrak{s}, i)}}$$

App-Cong
$$\frac{\Gamma \vdash t \equiv t' : \Pi(x : A).\, B \qquad \Gamma \vdash u \equiv u' : A}{\Gamma \vdash t\, u \equiv t'\, u' : B[x := u]}$$

Σ-Cong
$$\frac{\Gamma \vdash A \equiv A' : \mathscr{U}_i \qquad \Gamma, x : A \vdash B \equiv B' : \mathscr{U}_j}{\Gamma \vdash \Sigma(x : A).\, B \equiv \Sigma(x : A').\, B' : \mathscr{U}_{\max(i,j)}}$$

Snd-Cong
$$\frac{\Gamma \vdash t \equiv t' : \Sigma(x : A).\, B}{\Gamma \vdash \mathsf{proj}_2(t) \equiv \mathsf{proj}_2(t') : B[x := \mathsf{proj}_1(t)]}$$

Pair-Cong
$$\frac{\Gamma, x : A : \mathscr{U}_i \vdash B : \mathscr{U}_j \qquad \Gamma \vdash t \equiv t' : A \qquad \Gamma \vdash u \equiv u' : B[x := t]}{\Gamma \vdash \langle t, u \rangle \equiv \langle t', u' \rangle : \Sigma(x : A).\, B}$$

Fst-Cong
$$\frac{\Gamma \vdash t \equiv t' : \Sigma(x : A).\, B}{\Gamma \vdash \mathsf{proj}_1(t) \equiv \mathsf{proj}_1(t') : A}$$

ℕ-Elim-Cong
$$\frac{\Gamma \vdash A \equiv A' : \mathbb{N} \to \mathscr{U}_i \qquad \Gamma \vdash t_0 \equiv t_0' : A\ 0 \qquad \Gamma \vdash t_S \equiv t_S' : \Pi(n : \mathbb{N}).\, A\ n \to A\ (\mathsf{S}\ n) \qquad \Gamma \vdash n \equiv n' : \mathbb{N}}{\Gamma \vdash \mathbb{N}\text{-elim}(A, t_0, t_S, n) \equiv \mathbb{N}\text{-elim}(A', t_0', t_S', n') : A\ n}$$

Suc-Cong
$$\frac{\Gamma \vdash n \equiv n' : \mathbb{N}}{\Gamma \vdash \mathsf{S}\ n \equiv \mathsf{S}\ n' : \mathbb{N}}$$

Box-Cong
$$\frac{\Gamma \vdash A \equiv A' : \Omega}{\Gamma \vdash \Box A \equiv \Box A' : \mathscr{U}_0}$$

Box-Proof-Cong
$$\frac{\Gamma \vdash t \equiv t' : A}{\Gamma \vdash \diamond t \equiv \diamond t' : \Box A}$$

Quotient-Cong
$$\frac{\begin{array}{c}\Gamma \vdash A \equiv A' : \mathscr{U}_i \qquad \Gamma \vdash R \equiv R' : A \to A \to \Omega \qquad \Gamma \vdash R_r \equiv R_r' : \Pi(x : A).\, R\, x\, x \\ \Gamma \vdash R_s \equiv R_s' : \Pi(x, y : A).\, R\, x\, y \to R\, y\, x \qquad \Gamma \vdash R_t \equiv R_t' : \Pi(x, y, z : A).\, R\, x\, y \to R\, y\, z \to R\, x\, z\end{array}}{\Gamma \vdash A/(R, R_r, R_s, R_t) \equiv A'/(R', R_r', R_s', R_t') : \mathscr{U}_i}$$

Quotient-Proj-Cong
$$\frac{\Gamma \vdash t \equiv t' : A \qquad \Gamma \vdash R : A \to A \to \Omega : \mathscr{U}_i \qquad \mathsf{isrel}(R)}{\Gamma \vdash \pi(t) \equiv \pi(t') : A/R}$$

Q-Elim-Cong
$$\frac{\begin{array}{c}\Gamma \vdash B \equiv B' : A/R \to \mathscr{U}_i \qquad \Gamma \vdash t_\pi \equiv t_\pi' : \Pi(x : A).\, B\, \pi(x) \\ \Gamma \vdash t_\sim \equiv t_\sim' : \Pi(x, y : A).\, \Pi(e : R\, x\, y).\, (t_\pi\ x) \sim_{B\ \pi(x)} \mathsf{cast}(B\ \pi(y), B\ \pi(x), B\ e^{-1}, t_\pi\ y) \qquad \Gamma \vdash u \equiv u' : A/R\end{array}}{\Gamma \vdash \mathsf{Q}\text{-elim}(B, t_\pi, t_\sim, u) \equiv \mathsf{Q}\text{-elim}(B', t_\pi', t_\sim', u') : B\, u}$$

Id-Cong
$$\frac{\Gamma \vdash A \equiv A' : \mathscr{U}_i \qquad \Gamma \vdash t \equiv t' : A \qquad \Gamma \vdash u \equiv u' : A}{\Gamma \vdash \mathsf{Id}(A, t, u) \equiv \mathsf{Id}(A', t', u') : \mathscr{U}_i}$$

Idrefl-Cong
$$\frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash t \equiv t' : A}{\Gamma \vdash \mathsf{Idrefl}(t) \equiv \mathsf{Idrefl}(t') : \mathsf{Id}(A, t, t)}$$

Idpath-Cong
$$\frac{\Gamma \vdash A : \mathscr{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A \qquad \Gamma \vdash e \equiv e' : t \sim_A u}{\Gamma \vdash \mathsf{Idpath}(e) \equiv \mathsf{Idpath}(e') : \mathsf{Id}(A, t, u)}$$

J-Cong
$$\frac{\Gamma \vdash A \equiv A' : \mathcal{U}_i \qquad \Gamma \vdash t \equiv t' : A \qquad \Gamma \vdash B \equiv B' : \Pi(x : A).\, \mathsf{Id}(A, t, x) \to \mathcal{U}_j \qquad \Gamma \vdash u \equiv u' : B\ t\ \mathsf{Idrefl}(t) \qquad \Gamma \vdash v \equiv v' : A \qquad \Gamma \vdash e \equiv e' : \mathsf{Id}(A, t, v)}{\Gamma \vdash \mathsf{J}(A, t, B, u, v, e) \equiv \mathsf{J}(A', t', B', u', v', e') : B\ v\ e}$$

A.6 Substitution Rules

App-Subst
$$\frac{\Gamma \vdash t \Rightarrow t' : \Pi(x : A).\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u \Rightarrow t'\ u : B[x := u]}$$

Fst-Subst
$$\frac{\Gamma \vdash t \Rightarrow t' : \Sigma(x : A).\, B}{\Gamma \vdash \mathsf{proj}_1(t) \Rightarrow \mathsf{proj}_1(t') : A}$$

Snd-Subst
$$\frac{\Gamma \vdash t \Rightarrow t' : \Sigma(x : A).\, B}{\Gamma \vdash \mathsf{proj}_2(t) \Rightarrow \mathsf{proj}_2(t') : B[x := \mathsf{proj}_1(t)]}$$

$\mathbb{N}$-Elim-Subst
$$\frac{\Gamma \vdash A : \mathbb{N} \to \mathcal{U}_i \qquad \Gamma \vdash t_0 : A\ 0 \qquad \Gamma \vdash t_S : \Pi(n : \mathbb{N}).\, A\ n \to A\ (\mathsf{S}\ n) \qquad \Gamma \vdash n \Rightarrow n' : \mathbb{N}}{\Gamma \vdash \mathbb{N}\text{-}\mathsf{elim}(A, t_0, t_S, n) \Rightarrow \mathbb{N}\text{-}\mathsf{elim}(A, t_0, t_S, n') : A\ n}$$

Q-Elim-Subst
$$\frac{\Gamma \vdash B : A/R \to \mathcal{U}_i \qquad \Gamma \vdash t_\pi : \Pi(x : A).\, B\ \pi(x) \qquad \Gamma \vdash t_\sim : \Pi(x, y : A).\, \Pi(e : R\ x\ y).\, (t_\pi\ x) \sim_{B\ \pi(x)} \mathsf{cast}(B\ \pi(y), B\ \pi(x), B\ e^{-1}, t_\pi\ y) \qquad \Gamma \vdash u \Rightarrow u' : A/R}{\Gamma \vdash \mathsf{Q}\text{-}\mathsf{elim}(B, t_\pi, t_\sim, u) \Rightarrow \mathsf{Q}\text{-}\mathsf{elim}(B, t_\pi, t_\sim, u') : B\ u}$$

J-Subst
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash B : \Pi(x : A).\, \mathsf{Id}(A, t, x) \to \mathcal{U}_j \qquad \Gamma \vdash u : B\ t\ \mathsf{Idrefl}(t) \qquad \Gamma \vdash t' : A \qquad \Gamma \vdash e \Rightarrow e' : \mathsf{Id}(A, t, t')}{\Gamma \vdash \mathsf{J}(A, t, B, u, t', e) \Rightarrow \mathsf{J}(A, t, B, u, t', e') : B\ t'\ e}$$

Eq-Subst-Type
$$\frac{\Gamma \vdash A \Rightarrow A' : \mathcal{U}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t \sim_A u \Rightarrow t \sim_{A'} u : \Omega}$$

Eq-Subst-Left
$$\frac{\mathsf{hd}(A) \in \{\mathbb{N}, \mathcal{U}_i, Q\} \qquad \Gamma \vdash t \Rightarrow t' : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t \sim_A u \Rightarrow t' \sim_A u : \Omega}$$

Eq-Subst-Right
$$\frac{\mathsf{hd}(A) \in \{\mathbb{N}, \mathcal{U}_i, Q\} \qquad \Gamma \vdash t : A \qquad t \text{ is a whnf} \qquad \Gamma \vdash u \Rightarrow u' : A}{\Gamma \vdash t \sim_A u \Rightarrow t \sim_A u' : \Omega}$$

Cast-Subst-Left
$$\frac{\Gamma \vdash A \Rightarrow A' : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash e : A \sim_{\mathcal{U}_i} B \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{cast}(A, B, e, t) \Rightarrow \mathsf{cast}(A', B, e, t) : B}$$

Cast-Subst-Right
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad A \text{ is a whnf} \qquad \Gamma \vdash B \Rightarrow B' : \mathcal{U}_i \qquad \Gamma \vdash e : A \sim_{\mathcal{U}_i} B \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathsf{cast}(A, B, e, t) \Rightarrow \mathsf{cast}(A, B', e, t) : B}$$

Cast-Subst-Term
$$\frac{\mathsf{hd}(A) = \mathsf{hd}(B) \qquad \mathsf{hd}(A) \in \{\mathbb{N}, Q, \mathsf{Id}, \square\} \qquad \Gamma \vdash e : A \sim_{\mathcal{U}} B \qquad \Gamma \vdash t \Rightarrow t' : A}{\Gamma \vdash \mathsf{cast}(A, B, e, t) \Rightarrow \mathsf{cast}(A, B, e, t') : B}$$

Bibliography

Here are the references in citation order.

[1] Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. "The Taming of the Rew: A Type Theory with Computational Assumptions." In: *Proc. ACM Program. Lang.* 5.POPL (Jan. 2021). DOI: 10.1145/3434341 (cited on pages 1, 13, 111).

[2] Per Martin-Löf. "An Intiutionistic Theory of Types: Predicative Part." In: *Logic Colloquium '73*. Ed. by H. E. Rose and J. Shepherdson. 1975, pp. 73–118 (cited on pages 1, 11, 13, 22, 127).

[3] P. Letouzey. "Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq." PhD thesis. Université Paris-Sud, July 2004 (cited on pages 2, 14).

[4] Leonardo de Moura et al. "The Lean Theorem Prover." In: *Automated Deduction - CADE-25, 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*. 2015 (cited on pages 2, 4, 14, 15).

[5] Andreas Abel and Thierry Coquand. "Failure of Normalization in Impredicative Type Theory with Proof-Irrelevant Propositional Equality." In: *Logical Methods in Computer Science* Volume 16, Issue 2 (June 2020). DOI: 10.23638/LMCS-16(2:14)2020 (cited on pages 3, 4, 9, 14, 15, 20, 76, 86).

[6] Gaëtan Gilbert et al. "Definitional Proof-Irrelevance without K." In: *Proceedings of the ACM on Programming Languages*. POPL'19 3 (Jan. 2019), pp. 1–28. DOI: 10.1145/3290316 (cited on pages 3, 9, 14, 20, 24, 47, 84, 96).

[7] Chung-Kil Hur et al. "The Power of Parameterization in Coinductive Proof." In: *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '13. Rome, Italy: Association for Computing Machinery, 2013, pp. 193–206. DOI: 10.1145/2429069.2429093 (cited on pages 4, 15, 29).

[8] The Coq Development Team. *The Coq proof assistant reference manual*. Version 8.6. 2016 (cited on pages 4, 15).

[9] Agda Development Team. *Agda 2.6.1 documentation*. 2020 (cited on pages 4, 16).

[10] Martin Hofmann. "Extensional concepts in intensional type theory." PhD thesis. University of Edinburgh, 1995 (cited on pages 5, 16, 17, 69).

[11] Thorsten Altenkirch. "Extensional Equality in Intensional Type Theory." In: *14th Symposium on Logic in Computer Science*. 1999, pp. 412–420 (cited on pages 5, 16).

[12] T. Altenkirch. "Extensional equality in intensional type theory." In: *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*. 1999, pp. 412–420. DOI: 10.1109/LICS.1999.782636 (cited on pages 5, 17).

[13] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational equality, now!" In: *Proceedings of the Workshop on Programming Languages meets Program Verification (PLPV 2007)*. 2007, pp. 57–68. DOI: 10.1145/1292597.1292608 (cited on pages 6, 17, 24, 48, 78).

[14] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. "A Cubical Language for Bishop Sets." In: *Logical Methods in Computer Science* 18 (1 Mar. 2022). DOI: 10.46298/lmcs-18(1:43)2022 (cited on pages 6, 17).

[15] Thorsten Altenkirch et al. "Setoid type theory - a syntactic translation." In: *MPC 2019 - 13th International Conference on Mathematics of Program Construction*. Vol. 11825. LNCS. Springer, 2019, pp. 155–196. DOI: 10.1007/978-3-030-33636-3_7 (cited on pages 6, 17).

[16] Conor McBride. *Epigram 2 - Autopsy, Obituary, Apology*. 2020. URL: https://www.youtube.com/watch?v=5vZJVWCgf_4 (cited on pages 6, 17).

[17] Chris Kapulkin and Peter LeFanu Lumsdaine. "The simplicial model of Univalent Foundations (after Voevodsky)." In: (2018) (cited on pages 6, 17).

[HoTT] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013 (cited on pages 6, 7, 17, 18, 25, 95, 126).

[18] Guillaume Brunerie. "On the homotopy groups of spheres in homotopy type theory." PhD thesis. Université de Nice, 2016 (cited on pages 8, 19, 123, 124, 142).

[19] Andrej Bauer et al. "The HoTT Library: A Formalization of Homotopy Type Theory in Coq." In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. CPP 2017. Paris, France: ACM, 2017, pp. 164–172. DOI: 10.1145/3018610.3018615 (cited on pages 8, 19).

[20] Agda-Unimath development team. *The Agda-Unimath Library*. 2022. URL: https://unimath.github.io/agda-unimath/ (cited on pages 8, 19).

[21] 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 für Informatik, 2014, pp. 107–128. DOI: http://dx.doi.org/10.4230/LIPIcs.TYPES.2013.107 (cited on pages 8, 19, 90, 104).

[22] Cyril Cohen et al. "Cubical Type Theory: a constructive interpretation of the univalence axiom." In: *21st International Conference on Types for Proofs and Programs*. 21st International Conference on Types for Proofs and Programs 69. Tallinn, Estonia: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, May 2015, p. 262. DOI: 10.4230/LIPIcs.TYPES.2015.5 (cited on pages 8, 19, 39).

[23] Simon Huber. "Canonicity for Cubical Type Theory." In: *Journal Automated Reasoning* 63.2 (Aug. 2019), pp. 173–210. DOI: 10.1007/s10817-018-9469-1 (cited on pages 8, 19).

[24] Jonathan Sterling and Carlo Angiuli. "Normalization for Cubical Type Theory." In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2021. DOI: 10.1109/lics52264.2021.9470719 (cited on pages 8, 19).

[25] 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. DOI: 10.1145/3290314 (cited on pages 8, 19, 138).

[26] 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, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 6:1–6:17. DOI: 10.4230/LIPIcs.CSL.2018.6 (cited on pages 8, 19, 124, 125, 146).

[27] Carlo Angiuli et al. "Syntax and Models of Cartesian Cubical Type Theory." Preprint. Feb. 2019 (cited on pages 8, 19, 104, 124, 125, 146).

[28] Cyril Cohen et al. "Cubicaltt." https://github.com/mortberg/cubicaltt. 2015 (cited on pages 8, 19, 146).

[29] The RedPRL Development Team. *The redtt Proof Assistant*. 2018. URL: https://github.com/RedPRL/redtt/ (cited on pages 8, 19, 125, 146).

[30] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. "Decidability of Conversion for Type Theory in Type Theory." In: *Proceedings of the ACM on Programming Languages* 2.POPL (Jan. 2018), 23:1–23:29. DOI: 10.1145/3158111 (cited on pages 9, 20, 45, 46, 48, 57, 62).

[31] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. "Observational Equality, Now!" In: *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*. New York, NY, USA: ACM, 2007, pp. 57–68 (cited on pages 9, 20, 61).

[32] Pierre-Marie Pédrot. "Russian Constructivism in a Prefascist Theory." In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '20. Saarbrücken, Germany: Association for Computing Machinery, 2020, pp. 782–794. DOI: 10.1145/3373718.3394740 (cited on pages 10, 21, 90, 96, 103, 111).

[33] Agda-Cubical development team. *A Standard Library for Cubical Agda*. 2022. URL: https://github.com/agda/cubical (cited on pages 10, 21).

[34] Anders Mörtberg and Loïc Pujet. "Cubical Synthetic Homotopy Theory." In: *CPP 2020 - 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. New Orleans, United States: ACM, Jan. 2020, pp. 1–14. DOI: 10.1145/3372885.3373825 (cited on pages 11, 21).

[35] Loïc Pujet and Nicolas Tabareau. "Observational Equality: Now For Good." In: *Proceedings of the ACM on Programming Languages* 6.POPL (Jan. 2022), pp. 1–29. DOI: 10.1145/3498693 (cited on pages 11, 21, 68, 71).

[36] Loïc Pujet and Nicolas Tabareau. "Impredicative Observational Equality." In: *Proceedings of the ACM on Programming Languages* POPL (2023) (cited on pages 11, 21).

[37] Christine Paulin-Mohring. "Introduction to the Calculus of Inductive Constructions." In: *All about Proofs, Proofs for All*. Ed. by Bruno Woltzenlogel Paleo and David Delahaye. Vol. 55. Studies in Logic (Mathematical logic and foundations). College Publications, Jan. 2015 (cited on pages 11, 22).

[38] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types." In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019). DOI: 10.1145/3341691 (cited on page 19).

[39] 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. DOI: 10.4230/LIPIcs.FSCD.2019.31 (cited on page 36).

[40] 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*. LICS '18. Oxford, United Kingdom: Association for Computing Machinery, 2018, pp. 255–264. DOI: 10.1145/3209108.3209197 (cited on page 36).

[41] Simon Boulier and Théo Winterhalter. "Weak Type Theory is Rather Strong." TYPES '19. 2019 (cited on page 39).

[42] Andrew Swan. "An algebraic weak factorisation system on 01-substitution sets: a constructive proof." In: *Journal of Logic and Analysis* (2016). DOI: 10.4115/jla.2016.8.1 (cited on page 39).

[43] Daniel Gratzer. *An inductive-recursive universe generic for small families*. 2022. DOI: 10.48550/ARXIV.2202.05529. URL: https://arxiv.org/abs/2202.05529 (cited on page 44).

[44] William W. Tait. "Intensional Interpretations of Functionals of Finite Type I." In: 32.2 (1967), pp. 198–212 (cited on page 46).

[45] Thorsten Altenkirch, Thomas Streicher, and Martin Hofmann. "Reduction-free normalisation for system F." 1997 (cited on page 46).

[46] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures dans l'arithmétique d'ordre supérieur." Thèse de Doctorat d'État, Université de Paris VII. 1972 (cited on pages 48, 76).

[47] Meven Lennon-Bertrand. "Bidirectional Typing in the Calculus of Inductive Constructions." PhD thesis. Nantes Université, 2022 (cited on page 62).

[48] Andrej Bauer, Philipp G. Haselwarter, and Théo Winterhalter. "A modular formalization of type theory in Coq." In: *23rd International Conference on Types for Proofs and Programs (TYPES)*. Budapest, Hungary, June 2017 (cited on page 63).

[49] Peter Hancock et al. "Small Induction Recursion." In: *Typed Lambda Calculi and Applications*. Ed. by Masahito Hasegawa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 156–172 (cited on page 66).

[50] Benjamin Werner. "On the Strength of Proof-Irrelevant Type Theories." In: 4 (Sept. 2008), pp. 1–20 (cited on page 76).

[51] Meven Lennon-Bertrand. "À bas l'η – Coq's troublesome η-conversion." WITS. 2022 (cited on page 77).

[52] 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. DOI: 10.1145/2502409.2502411 (cited on page 77).

[53] Bob Atkey. "Simplified Observational Type Theory." https://github.com/bobatkey/sott. 2017 (cited on page 80).

[54] Amin Timany and Matthieu Sozeau. "Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)." In: *CoRR* abs/1710.03912 (2017) (cited on page 83).

[55] Chung-Kil Hur. "Agda with the excluded middle is inconsistent." https://sympa.inria.fr/sympa/arc/coq-club/2010-01/msg00007.html. 2010 (cited on page 83).

[56] Benjamin Werner. "Une Théorie des Constructions Inductives." Theses. Université Paris-Diderot - Paris VII, May 1994 (cited on page 88).

[57] Cyril Cohen et al. "Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom." In: *Types for Proofs and Programs (TYPES 2015)*. Vol. 69. LIPIcs. 2018, 5:1–5:34 (cited on pages 90, 104, 106, 110, 124, 128, 130).

[58] S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer New York, 1994 (cited on pages 91, 92).

[59] Martin Hofmann. "Syntax and semantics of dependent types." In: *Semantics and logics of computation*. Ed. by A.M. Pitts and P. Dybjer. Vol. 14. Publ. Newton Inst. Papers from the Summer School held at the University of Cambridge, Cambridge, September 1995. Cambridge: Cambridge University Press, 1997, pp. 79–130 (cited on pages 92, 93, 113).

[60] Voevodsky et al. "Semi-simplicial Types." Univalent Foundations at the IAS special year. 2012 (cited on page 95).

[61] Ulrik Buchholtz. "Update on semisimplicial types in homotopy type theory." Talk at the Workshop on Logic and higher structures, CIRM 2689. (Slides available at https://ulrikbuchholtz.dk/cirm2689.pdf). 2022 (cited on page 95).

[62] Andrej Bauer et al. "The HoTT Library: A Formalization of Homotopy Type Theory in Coq." In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. CPP 2017. Paris, France: ACM, 2017, pp. 164–172. DOI: 10.1145/3018610.3018615 (cited on pages 95, 147).

[63] Vladimir Voevodsky. "A simple type system with two identity types." Talk at Andre Joyal's 70th birthday conference. (Slides available at https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/HTS_slides.pdf). 2013 (cited on pages 96, 121).

[64] Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. "Two-Level Type Theory and Applications." 2017 (cited on pages 96, 121).

[65] Guilhem Jaber et al. "The Definitional Side of the Forcing." In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*. 2016, pp. 367–376 (cited on page 100).

[66] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. "The next 700 syntactical models of type theory." In: *Certified Programs and Proofs (CPP 2017)*. Paris, France, Jan. 2017, pp. 182–194. DOI: 10.1145/3018610.3018620 (cited on page 103).

[67] Ulrik Buchholtz and Edward Morehouse. "Varieties of Cubical Sets." In: (2017). Preprint arXiv:1701.08189v1 [math.CT] (cited on pages 105, 107).

[68] A. Hatcher. *Algebraic Topology*. Algebraic Topology. Cambridge University Press, 2002 (cited on page 107).

[69] Thierry Coquand et al. "Quillen model structure." Mailing list discussion. June 2018 (cited on page 111).

[70] Greg Friedman. "An elementary illustrated introduction to simplicial sets." In: (2008). DOI: 10.48550/ARXIV.0809.4221 (cited on page 111).

[71] Louis Kauffman. "De Morgan Algebras - completeness and recursion." In: *Proceedings of the Eighth International Symposium on Multiple-Valued Logic* (Jan. 1978), pp. 82–86 (cited on page 112).

[72] Martin Hofmann and Thomas Streicher. "Lifting Grothendieck Universes." Unpublished Note. 1997 (cited on page 113).

[73] Daniel Licata and Michael Shulman. "Calculating the Fundamental Group of the Circle in Homotopy Type Theory." In: June 2013, pp. 223–232. DOI: 10.1109/LICS.2013.28 (cited on pages 123, 124, 133).

[74] Daniel R. Licata and Guillaume Brunerie. "$\pi_n(S^n)$ in Homotopy Type Theory." In: *Certified Programs and Proofs*. Ed. by Georges Gonthier and Michael Norrish. Springer International Publishing, 2013, pp. 1–16 (cited on pages 123, 124, 138).

[75] Evan Cavallo. "Synthetic cohomology in homotopy type theory." MA thesis. Carnegie Mellon University, 2015 (cited on pages 123, 147).

[76] Kuen-Bang Hou (Favonia) and Michael Shulman. "The Seifert-van Kampen Theorem in Homotopy Type Theory." In: *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*. Ed. by Jean-Marc Talbot and Laurent Regnier. Vol. 62. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, 22:1–22:16. DOI: 10.4230/LIPIcs.CSL.2016.22 (cited on pages 123, 124).

[77] Kuen-Bang Hou (Favonia) et al. "A Mechanization of the Blakers-Massey Connectivity Theorem in Homotopy Type Theory." In: *31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16*. New York, NY, USA: ACM, July 2016, pp. 565–574 (cited on pages 123, 124).

[78] Floris van Doorn. "On the Formalization of Higher Inductive Types and Synthetic Homotopy Theory." PhD thesis. Carnegie Mellon University, 2018 (cited on pages 123, 124).

[79] Kristina Sojakova. "The Equivalence of the Torus and the Product of Two Circles in Homotopy Type Theory." In: *ACM Transactions on Computational Logic* 17.4 (Nov. 2016), 29:1–29:19 (cited on pages 124, 132).

[80] Daniel R. Licata and Guillaume Brunerie. "A Cubical Approach to Synthetic Homotopy Theory." In: *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS'15*. New York, NY, USA: ACM, July 2015, pp. 92–103 (cited on pages 124, 147).

[81] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. "Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types." In: *Proc. ACM Program. Lang.* 3.ICFP (July 2019), 87:1–87:29. DOI: 10.1145/3341691 (cited on pages 125, 132).

[82] Vladimir Voevodsky. "An experimental library of formalized Mathematics based on the univalent foundations." In: *Mathematical Structures in Computer Science* 25 (2015), pp. 1278–1294 (cited on page 130).

[83] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013 (cited on pages 130, 134).

[84] 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*. LICS '18. Oxford, United Kingdom: ACM, 2018, pp. 255–264. DOI: 10.1145/3209108.3209197 (cited on page 138).

[85] Egbert Rijke. "The join construction." Preprint arXiv:1701.07538v1. 2017 (cited on page 141).

[86] Guillaume Brunerie et al. *Homotopy Type Theory in Agda*. URL: https://github.com/HoTT/HoTT-Agda (cited on page 147).

[87] Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. "Homotopy Type Theory in Lean." In: *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*. Ed. by Mauricio Ayala-Rincón and César A. Muñoz. Vol. 10499. Lecture Notes in Computer Science. Springer, 2017, pp. 479–495. DOI: 10.1007/978-3-319-66107-0_30 (cited on page 147).

DOCTORAT BRETAGNE LOIRE / MATHSTIC

Nantes Université

Titre : Calculer avec des Principes d'Extensionnalité en Théorie des Types

Mots clés : Théorie des Types, Assistants à la Preuve, Preuve de Normalisation, Égalité Observationnelle, Théorie Cubique des Types, Homotopie Synthétique

Résumé : Dans cette thèse, j'étudie plusieurs manières d'étendre la théorie des types intuitionniste avec des principes d'extensionnalité comme par exemple l'extensionnalité des fonctions ou l'axiome d'univalence de Voevodsky, tout en préservant les propriétés calculatoires des preuves.

Dans une première partie, je développe une méta-théorie complète pour l'égalité observationnelle de Altenkirch *et al*. J'obtiens notamment une preuve formelle de normalisation, de canonicité et de décidabilité de la conversion pour une théorie des types observationnelle avec des propositions imprédicatives.

Dans une seconde partie, j'esquisse une traduction de la théorie des types homotopique vers la théorie des types observationnelle, en me basant sur le modèle cubique de Coquand *et al*.

Enfin dans une dernière partie, j'explique comment tirer parti des propriétés calculatoires de la théorie des types cubique pour obtenir des preuves synthétiques élégantes de résultats classiques de la théorie de l'homotopie, notamment la construction de la fibration de Hopf et le lemme 3x3 pour les sommes amalgamées homotopiques.

Title : Computing with Extensionality Principles in Type Theory

Keywords : Type Theory, Proof Assistants, Normalization Proofs, Observational Equality, Cubical Type Theory, Synthetic Homotopy Theory

Abstract : In this thesis, I study several possibilities to extend intuitionistic type theory with extensionality principles such as function extensionality or Voevodsky's univalence axiom, while preserving the computational properties of the proofs.

In the first part, I develop a complete meta-theory for the observational equality of Altenkirch *et al*. In particular, I obtain a formal proof of normalization, canonicity and decidability of the conversion for an observational type theory with impredicative proof-irrelevant propositions.

Then in a second part, I sketch a translation from homotopy type theory to observational type theory based on the model of Coquand et al in cubical sets.

Finally, in the last part I explain how to take advantage of the computational properties of cubical type theory to obtain elegant synthetic proofs of classical results from homotopy theory, in particular the construction of the Hopf fibration and the 3x3 lemma for homotopy pushouts.