Extending type theory with syntactic models

Simon Pierre Boulier

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THESE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPERIEURE MINES-TELECOM ATLANTIQUE
BRETAGNE PAYS DE LA LOIRE - IMT ATLANTIQUE
COMUE UNIVERSITE BRETAGNE LOIRE

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

Par

Simon BOULIER

Extending Type Theory with Syntactic Models

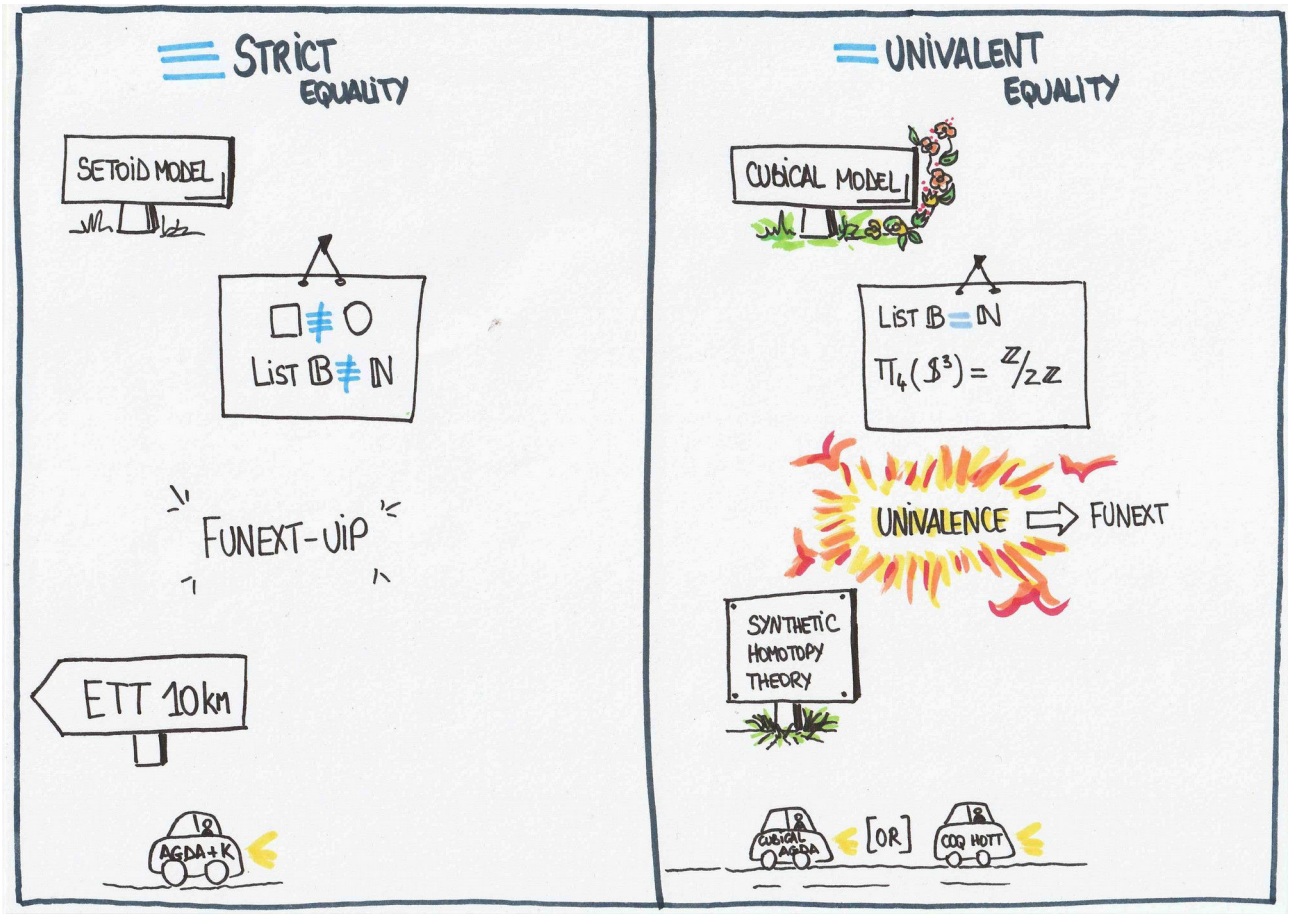
Thèse présentée et soutenue à Nantes, le 29 novembre 2018
Unité de recherche : LS2N
Thèse N° : 2018IMTA0110

Rapporteurs avant soutenance :

Thierry Coquand	Professor, University of Gothenburg
Hugo Herbelin	Directeur de Recherche, INRIA Paris

Composition du Jury :

Président :	Pierre Cointe	Professeur, IMT Atlantique
Examinateurs :	Ambrus Kaposi	Assistant Professor, Eötvös Lorand University
	Assia Mahboubi	Chargée de Recherche, INRIA Rennes
	Thierry Coquand	Professor, University of Gothenburg
	Hugo Herbelin	Directeur de Recherche, INRIA Paris
Dir. de thèse :	Nicolas Tabareau	Chargé de Recherche, INRIA Rennes



Thanks to Coline for the drawing

Abstract

This thesis is about the metatheory of intuitionistic type theory. We consider variants of Martin-Löf type theory or of the Calculus of Constructions, and we pay attention to two recurring questions: the coherence of these systems and the independence of axioms with respect to them. We address this kind of problems by constructing syntactic models, which are models reusing type theory to interpret type theory.
In a first part, we introduce type theory through a minimal system with several possible extensions. We also define a few notions of homotopy type theory. Last, we introduce Categories with Families which are models of type theory. We recall two examples of such models: the standard model and the setoid model.
In a second part, we introduce a particular class of models: syntactic models given by a program translation. In this paradigm, a term is translated to a term, a type to a type and a context to a context. These models are simple in that a lot of constructions are reinterpreted by themselves. We give several program translations:

- the *times bool* translations which are used to negate some extensionality principles by adding a boolean in the right place,
- a translation implementing pattern-matching on the universe,
- several variants of parametricity.

For each one, we precise which axioms are negated or preserved by the translation.
In a third part, we present Template-Coq, a plugin for metaprogramming in Coq. This plugin provides a reification of Coq syntax and typing derivations. We demonstrate how to use it to implement directly some syntactic models.
Last, we consider type theories with two equalities: a strict one and a univalent one. We propose a re-reading of works of Coquand et. al. and of Orton and Pitts on the cubical model by introducing degenerate fibrancy. This notion of fibrancy has a defect: the universe of degenerately fibrant types is not univalent. Nonetheless, degenerate fibrancy seems worthy of attention, in particular because it admits a fibrant replacement. We show how the fibrant replacement can be used to interpret some Higher Inductive Types and to define a model structure on the universe. This last result is then reinterpreted as the construction of a model structure on the category of cubical sets.

Formalizations

Several formalizations, either in Coq or in Agda, are presented in this thesis. They are available at the following address, each one in a separate folder:

https://gitlab.inria.fr/sboulier/thesis-formalizations

Each time that we quote a file, it will refer to this repository. The names are clickable and are direct link to the repository.

Résumé (en français)

Cette thèse s'intéresse à la métathéorie de la théorie des types intuitionniste. Nous considérons des variantes de la théorie des types de Martin-Löf ou du Calcul des Constructions, et portons notre attention sur deux questions récurrentes: la cohérence de ces systèmes et l'indépendance d'axiomes par rapport à ces systèmes. Nous attaquons ces questions par la construction de modèles syntaxiques, qui sont des modèles qui réutilisent la théorie des types pour interpréter la théorie des types.
Dans une première partie, nous introduisons la théorie des types à l'aide d'un système minimal et de plusieurs extensions potentielles. Nous définissons aussi quelques notions de théorie des types homotopique. Enfin, nous introduisons les Catégories de Famille qui sont des modèles de la théorie des types. Nous redonnons deux exemples de tels modèles : le modèle standard et le modèle setoïde.
Dans une seconde partie, nous introduisons une classe de modèles particuliers : les modèles syntaxiques donnés par traduction de programme. Dans ce paradigme, un terme est traduit par un terme, un type par un type et un contexte par un contexte. Ces modèles sont simples au sens où de nombreuses constructions sont réinterprétées par elles-mêmes. Nous donnons plusieurs traductions de programme :

- les traductions *croix bool* qui permettent de nier des principes d'extensionalité en ajoutant un booléen au bon endroit,
- une traduction qui implémente le pattern-matching sur l'univers,
- différentes variantes de la parametricité.

À chaque fois, nous précisons quels sont les axiomes niés ou préservés par ces traductions.
Dans une troisième partie, nous présentons Template-Coq, un plugin de métaprogrammation en Coq. Ce plugin fournit une réification de la syntaxe et des dérivations de typage de Coq. Nous montrons comment l'utiliser pour implémenter directement certains modèles syntaxiques.
Enfin, dans une dernière partie, nous considérons des théories des types avec deux égalités : une stricte et une univalente. Nous proposons une relecture des travaux de Coquand et. al. et Orton et Pitts sur le modèle cubique en introduisant la notion de fibrance dégénérée. Cette notion de fibrance a un défaut : l'univers des types dégénérément fibrants n'est pas univalent. Néanmoins, la fibrance dégénérée semble digne d'intérêt, notamment car elle admet une notion de remplacement fibrant. Nous montrons comment le remplacement fibrant peut être utilisé pour interpréter certains types inductifs supérieurs (HITs) et pour définir une structure de modèle sur l'univers. Ce dernier résultat est ensuite réinterprété comme la construction d'une structure de modèle sur la catégorie des types cubiques.

Formalisations

Plusieurs formalisations, soit en Coq soit en Agda, sont présentées dans cette thèse. Elles sont disponibles à l'adresse suivante, chacune dans un dossier séparé:

https://gitlab.inria.fr/sboulier/thesis-formalizations

Contents

Introduction (en français)

Cette thèse s'inscrit dans le cadre de l'étude de la théorie des types intuitionniste, un formalisme qui est à la fois un système logique et un langage de programmation. C'est un système logique : on y définit ce qu'est un énoncé mathématique (une proposition) et quelles sont les règles de raisonnement valables qui permettent de constituer une preuve d'une proposition. En cela, la théorie des types intuitionniste est une proposition de fondement des mathématiques, alternative au traditionnel tandem logique du premier ordre / théorie des ensembles. Mais la théorie des types intuitionniste est aussi un véritable langage de programmation dans lequel on peut écrire des programmes, que l'on peut ensuite exécuter pour calculer des valeurs. Ce langage de programmation est statiquement typé, chaque expression a un type qui décrit ce qu'elle dénote (un entier , un booléen, ...) et la théorie des types s'inspire donc — si ce n'est fait partie — de la programmation fonctionnelle.
Ce sont justement les types qui font la jonction entre système logique et langage de programmation. Ainsi, une proposition est représentée par un type et une preuve de cette proposition est une expression de ce type. Cette correspondance entre preuves et programmes est appelée correspondance de Curry-Howard, même si elle doit aussi à de nombreux autres auteurs. C'est un des résultats majeurs de l'informatique théorique. Depuis cette découverte dans les années 60, logique mathématique et théorie des langages de programmation s'enrichissent l'une l'autre.
La correspondance de Curry-Howard fait correspondre un langage de programmation à chaque système logique. Ainsi, dans le cas le plus simple, elle met en correspondance les programmes du λ-calcul simplement typé avec les preuves en logique propositionnelle intuitionniste. Ce qui fait la force de cette correspondance c'est qu'elle s'étend à de nombreux systèmes logiques. Au point que, quand l'on considère un nouveau système logique, il est devenu courant de se demander quel est le calcul sous-jacent. Par exemple, le calcul correspondant à la logique intuitionniste du second ordre est le Système F de Girard, des systèmes correspondants à la logique classique sont des calculs avec opérateurs de contrôle, etc.
Ce que nous appelons théorie des types intuitionniste fait référence à un système introduit par Martin-Löf dans les années 70 qui étend notamment la correspondance de Curry-Howard à la logique d'ordre supérieur. Dans la logique d'ordre supérieur, il est possible de quantifier à l'aide des quantificateurs « pour tout » $\forall$ et « il existe » $\exists$, sur les objets mais aussi sur les prédicats, les relations, ... (les objets « supérieurs »). La contrepartie des quantificateurs côté langage de programmation est l'utilisation de types dépendants avec des produits dépendants Π (version dépendante du type des fonctions $\rightarrow$) et des sommes dépendantes Σ (version dépendante du produit cartésien $\times$).
Nous appellerons ce système : théorie des types Martin-Löf (MLTT), théorie des types dépendants, théorie des types intuitionniste, ou tout simplement : théorie des types. La théorie des types telle que nous la connaissons aujourd'hui doit aussi beaucoup au Calcul des Constructions de Coquand et Huet qui introduit un univers des propositions. Ce système est à la base

de l'assistant de preuve Coq, nous y reviendrons. Le Calcul des Constructions a ensuite connu plusieurs évolutions, dont le Calcul des Constructions Inductives qui ajoute la gestion des types inductifs généraux.
Il existe de nombreuses variantes, tant dans la présentation que dans le fond, de la théorie des types intuitionniste. Aussi, par ce nom, nous ne nous référons pas à un système précis mais plutôt à un paradigme. Les constructions que l'on considère doivent ensuite être adaptées à chaque variante.

Assistants de preuves Cette thèse n'aurait pas été la même sans l'utilisation massive d'assistants de preuve. Un assistant de preuve est un logiciel dans lequel on peut formaliser un énoncé mathématique puis en fournir une preuve de façon interactive. Une telle démarche permet de s'assurer que la preuve est correcte dans les moindres détails et que l'on n'a pas fait d'erreur ou d'oubli.
Les assistants de preuve que nous avons utilisés sont Coq et Agda qui implémentent tous les deux des variantes de la théorie des types de Martin-Löf. Il en existe plusieurs autres, comme Lean par exemple, ainsi que pour d'autres systèmes logiques — par exemple HOL Light pour la logique d'ordre supérieur, Mizar pour une théorie des ensembles, etc.
Depuis quelques années des preuves conséquentes sont arrivées à portée des assistants de preuve. Ainsi, en 2012, une équipe menée par Georges Gonthier a annoncé l'achèvement de la formalisation complète du théorème de Feit-Thomson en Coq. Ce théorème des années 60 est un résultat difficile de théorie des groupes et sa démonstration fait plusieurs centaines de pages. Aujourd'hui, de plus en plus de travaux en théorie des langages de programmation sont formalisés et cela donne une certaine confiance en leur correction. Cette pratique reste toutefois encore minoritaire.
Une autre application importante de la théorie des types et des assistants de preuve est la certification de programmes. Un programme certifié est un logiciel pour lequel il a été prouvé mathématiquement qu'il satisfait une certaine spécification. Par exemple, une équipe menée par Xavier Leroy a développé CompCert, un compilateur C écrit en Coq pour lequel il a été prouvé (en Coq aussi) qu'il préserve la sémantique des programmes compilés. L'autre grand exemple de certification de programme est la création de logiciels prouvés sans bug (division par zéro, dépassement de tableaux, …), ce qui est intéressant pour les logiciels critiques dont notre société est aujourd'hui envahie.
Dans notre cas, nous avons utilisé à la fois Coq et Agda pour formaliser des preuves et s'assurer qu'elles étaient correctes. Mais nous les avons également utilisés comme de véritables *assistants* de preuve au sens où nous avons réalisé de nombreuses preuves directement en la formalisant. Nous laissons alors Coq ou Agda nous guider en nous montrant ce qu'il reste à démontrer et nous le guidons en retour en lui expliquant comment faire.

Métathéorie de MLTT Dans cette thèse, nous nous intéressons à la métathéorie de la théorie des types. C'est-à-dire que nous ne cherchons pas à démontrer des théorèmes *à l'intérieur* de la théorie des types mais nous démontrons des théorèmes qui *parlent de* ce système. Nous étudions donc la théorie des types comme un objet mathématique en lui-même.
Cela dit, puisque la théorie des types est notre formalisme de prédilection, nous l'utiliserons aussi comme métathéorie. C'est-à-dire que nous prouvons, en théorie des types, des théorèmes

qui parlent (d'un encodage profond) de la théorie des types. Remarquons qu'une part encore non négligeable de travaux en théorie des types utilise la théorie des ensembles comme métathéorie.
Les questions « métathéoriques » auxquelles nous cherchons à répondre sont notamment celles de la cohérence d'un système, de sa terminaison et de l'indépendance d'axiomes. La cohérence d'un système logique est une propriété cruciale, si ce n'est la plus importante, car elle correspond au fait de ne pas pouvoir démontrer n'importe quel énoncé, notamment les énoncés manifestement faux tels que « $12 + 2 = 15$ ». On cherche donc à montrer cette propriété pour tout nouveau système que l'on introduit. Le théorème d'incomplétude de Gödel refrène cependant un peu nos ardeurs : il énonce qu'un système logique ne peut montrer sa propre cohérence. Autrement dit, étant donné une théorie des types $\mathcal{T}$, si l'on formalise $\mathcal{T}$ dans $\mathcal{T}$ elle-même, alors nous ne pourrons pas démontrer que $\mathcal{T}$ est cohérente. Une des façons de contourner ce théorème dans une certaine mesure, est d'introduire une chaîne de théories $\mathcal{T}_0$, $\mathcal{T}_1$, … telle que chaque théorie démontre la cohérence de la précédente. C'est notamment possible si l'on introduit des théories qui ont de plus en plus d'univers.
À propos de l'indépendance d'axiomes, le but est cette fois de répondre aux questions qui sont le pain quotidien des théoriciens des types. Par exemple :

- « Peut-on prouver que $(\forall\, x.\ f\ x = g\ x) \to f = g$? »
- « Quand P et Q sont des propositions, quelle est la différence entre $P \leftrightarrow Q$ et $P = Q$? »
- « Peut-on prouver qu'un terme de type $\forall\, A.\ A \to A$ est nécessairement l'identité ? »

Traditionnellement, il existe deux principales voies pour étudier la métathéorie d'un système logique :

la voie syntaxique On montre que le système a de bonnes propriétés syntaxiques « directement » : la réduction du sujet, la terminaison forte, l'élimination des coupures, … On manipule donc directement des termes et des arbres de dérivation.

la voie sémantique On traduit le système logique considéré dans une autre structure mathématique. Et, en étudiant comment se comporte la version traduite du système, on peut faire remonter des propriétés et en déduire que le système satisfait telle ou telle propriété. On dit alors que l'on construit un *modèle* de notre système dans lequel on *l'interprète*. L'art de construire des modèles s'appelle la *sémantique dénotationnelle*.

Dans cette thèse, nous cherchons une voie intermédiaire en construisant des modèles très syntaxiques. La construction de modèles syntaxiques pour la théorie des types de Martin-Löf est un des leitmotivs de cette thèse. Ainsi, dans le chapitre 2, nous développons des modèles syntaxiques donnés par traduction de programme. Dans le chapitre 3, nous montrerons comment implémenter certains modèles syntaxiques dans l'assistant de preuve Coq. Et dans le chapitre 4, nous donnons un modèle pour une théorie des types avec deux égalités.
Plusieurs des modèles que l'on présente dans cette thèse ont trait à l'interprétation de l'égalité en théorie des types. Celle-ci est en effet pleine de subtilité et il n'est pas encore établi quelles sont les propriétés que l'on doit attendre d'elle. Récemment, l'avènement de la théorie des types homotopique a mis en lumière le fait qu'il existe au moins deux égalités bien différentes :

l'égalité stricte qui vérifie UIP — *Uniqueness of Identity Proofs* — et se rapproche de l'égalité utilisée en théorie des ensembles, et l'égalité univalente qui vérifie l'axiome d'univalence, un principe puissant qui permet de transporter les propriétés entre deux objets isomorphes. Cette distinction égalité stricte / égalité univalente est l'autre fil rouge de cette thèse.

Chapitre 1

Dans le premier chapitre, nous introduisons la théorie des types que nous utiliserons tout au long de cette thèse ainsi que quelques constructions typiques. Nous donnerons la syntaxe et la sémantique d'un système précis que nous appellerons $\lambda\Pi_\omega$. Ce système est assez minimal : il ne possède que des produits dépendants et une hiérarchie d'univers. Ce système doit être considéré comme une brique de base que l'on retrouve au cœur des variantes plus riches de la théorie des types. Nous présenterons d'ailleurs quelques unes des extensions possibles de $\lambda\Pi_\omega$: l'η-conversion, l'univers imprédicatif Prop, la cumulativité, les types inductifs et coinductifs, etc. Ces extensions permettent de se rapprocher d'une théorie plus réaliste telle celles implémentées par Coq ou Agda.

HoTT Dans ce chapitre, nous introduirons aussi des notions de théorie des types homotopique (HoTT). Comme déjà évoqué, la théorie des types homotopique est une branche récente de la théorie des types dans laquelle on étudie la structure des types égalité

$$t = u$$

En effet, malgré leur apparente simplicité ceux-ci possèdent une structure très riche : celle d'∞-groupoïde. En théorie des types homotopique, on s'intéresse aux types identité itérés : étant donné deux preuves d'égalité p et q de $t = u$, on s'intéresse au type $p = q$. Puis, étant donné deux preuves d'égalité e et f de $p = q$, on s'intéresse au type $e = f$, etc. Les types identité successifs ne sont pas forcément habités par contre cette itération leur donne une structure d'∞-groupoïde. Cette structure est révélée par l'interprétation homotopique de la théorie des types : un type est interprété par un espace topologique et une preuve d'égalité par un chemin dans cet espace topologique. Deux éléments du type sont alors égaux s'il existe un chemin de l'un à l'autre, deux preuves d'égalité sont égales s'il existe une homotopie (un chemin de chemin) entre les deux chemins correspondants et ainsi de suite.
Cette interprétation homotopique a été rendue précise par le modèle simplicial de Voevodsky où un type est interprété par un ensemble simplicial, puis par le modèle cubique de Coquand *et. al.* où, cette fois, l'on utilise des ensembles cubiques.
La théorie des types homotopique est notamment articulée autour de l'axiome d'univalence qui implique que deux types équivalents (penser « isomorphes ») sont égaux. Ce principe est très puissant puisqu'il permet de transporter toutes les propriétés d'un objet à un autre objet isomorphe. Par exemple, si un groupe G_1 est isomorphe à un groupe G_2 et que l'on sait que G_1 est résoluble alors G_2 l'est aussi. En effet, l'axiome d'univalence implique que deux groupes isomorphes sont en fait égaux pour l'égalité univalente. Pour des structures plus compliquées que celle de groupe, cette technique peut même permettre de s'assurer que l'on a trouvé la bonne notion d'isomorphisme.

Catégorie de familles Enfin, nous introduirons les catégories de familles qui sont un formalisme usuel pour décrire ce qu'est un modèle de la théorie des types. Les catégories de familles ont été introduites par Dybjer en 1995. Dybjer en a donné la définition à la fois en théorie des ensembles et en théorie des types, nous suivrons donc naturellement cette seconde approche. Par contre, contrairement à lui, nous utiliserons une égalité stricte (qui se rapproche beaucoup de celle de la théorie des ensembles), ce qui nous permet de nous passer de setoïdes.
Les catégories de familles ont aussi reçu un nouvel éclairage avec les travaux de Altenkirch et Kaposi qui ont montré que l'on peut construire une catégorie de familles initiale à l'aide d'un type inductif quotient (QIT) et que cette catégorie initiale se révèle être la syntaxe « intrinsèquement typée » de la théorie des types. C'est-à-dire une syntaxe dans laquelle il n'est pas possible d'écrire un terme mal typé. Par opposition, nous appellerons syntaxe « extrinsèque », la syntaxe habituelle dans laquelle on définit d'abord les termes non typés puis une relation de typage. Nous verrons que l'interprétation de la syntaxe extrinsèque dans la syntaxe intrinsèque n'est pas aussi évidente qu'il n'y paraît.
Nous donnerons deux exemples de catégories de familles : le modèle standard qui est un modèle trivial donné par l'identité, et le modèle setoïde de Altenkirch qui est un modèle de l'égalité stricte. Nous présenterons plusieurs contributions à ce modèle : une définition alternative des familles de setoïde, l'interprétation d'un univers clos par produits dépendants, et une preuve que l'extensionnalité des fonctions vérifie des règles de réduction dans ce modèle.

Chapitre 2

Le second chapitre est consacré aux modèles syntaxiques donnés par traduction de programme. Une traduction de programme est la traduction d'une théorie des types source vers une théorie des types cible, à la façon d'une phase de compilation. Un terme est traduit par un terme, un type par un type et un contexte par un contexte.
Les traductions de programme sont donc des modèles sauf que, plutôt que d'interpréter notre théorie cible dans une structure éloignée de la théorie des types (un espace vectoriel, une catégorie à poils bleus, …), on l'interprète dans une structure très proche : une autre théorie des types. L'avantage d'avoir une théorie des types pour cible est que l'on peut interpréter la substitution et la conversion par elles-mêmes :

$$[M\{x := N\}] = [M]\big\{x := [N]\big\} \qquad\qquad M \simeq_\beta N \Rightarrow [M] \simeq_\beta [N]$$

Une autre caractéristique des traductions de programme est qu'elles sont définies sur la présyntaxe (les programmes non forcément bien typés). On peut alors se passer du cadre des catégories de familles et montrer directement un théorème de correction qui prend la forme de préservation du typage :

$$\Gamma \vdash t : A \quad \Rightarrow \quad [\![\Gamma]\!] \vdash [t] : [\![A]\!]$$

Chaque traduction n'a pas vocation à justifier entièrement une théorie des types puisque la théorie cible est généralement proche de la source. Par contre, le but est d'avoir une chaîne de compilation

$$\mathcal{T}_0 \to \mathcal{T}_1 \to \ldots \to \mathcal{T}_n$$

où chaque traduction justifie un principe logique additionnel. Ultimement $\mathcal{T}_n$ est une théorie « minimale » qu'il faut justifier de façon syntaxique (avec une preuve de normalisation).
Les traductions de programme ont plusieurs avantages par rapport aux catégories de familles :

elles donnent des modèles simples Les traductions permettent de construire des modèles à peu de frais car beaucoup de constructions sont interprétées de façon transparente : la conversion par la conversion, les univers par les univers, etc. Certaines traductions sont même très « locales » : elles agissent sur un constructeur et laissent tous les autres inchangés.

elles n'utilisent que la théorie des types Si la théorie des types se veut un nouveau fondement des mathématiques, elle doit être capable de s'étudier elle-même : « type theory should eat itself » [1]. Les traductions de programme permettent de construire des modèles sans recourir à la théorie des ensembles, que ce soit dans la cible ou la métathéorie utilisée. On appelle cette approche *moniste* : le seul paradigme que l'on utilise est celui de la théorie des types.

elles sont flexibles Les traductions se composent et s'étendent facilement. En outre, elles sont compatibles avec de nombreuses variations (par exemple, univers à la Tarski *vs.* univers à la Russell) du moment que les théories source et cible utilisent le même paradigme.

elles sont implémentables Étant donné un terme, il est direct de calculer sa traduction. Nous avons d'ailleurs développé un plugin pour Coq qui réalise cela. Ce n'est pas si simple pour les modèles traditionnels formulés à l'aide de catégories de familles car cela nécessite la traduction « extrinsèque vers intrinsèque ».

elles utilisent une métathéorie faible Contrairement aux catégories de famille, la distinction entre la métathéorie et la théorie cible est claire et la métathéorie nécessaire pour les traductions de programme que l'on donne est vraiment très faible. Essentiellement, la métathéorie doit permettre de faire des inductions sur la syntaxe mais guère plus.

Les traductions de programme sont présentes depuis bien longtemps en théorie des types. Elles puisent leurs racines dans les traductions logiques — traduction de Gödel, traduction CPS de Lafont-Streicher-Reus, etc. — et l'on en aperçoit des fragments dans de nombreux articles — par exemple le *subset model* de Hofmann. Mais c'est surtout dans des travaux récents qu'elles apparaissent de façon plus explicite : par exemple, dans des travaux sur le forcing, Dialectica, ou encore la paramétricité. Aussi, nous ne prétendons pas inventer ou découvrir les traductions de programmes mais notre but est de leur donner un cadre pour populariser cette façon élégante de construire des modèles.
Dans ce travail, nous présentons plusieurs exemples de traductions s'appliquant à $\lambda\Pi_\omega$:

- Les traductions « croix bool » qui permettent de nier des principes d'extensionnalité en ajoutant simplement un booléen au bon endroit. Avec cette technique nous montrons comment nier l'extensionnalité des fonctions, l'extensionnalité sur les *streams* et l'extensionnalité propositionnelle.

[1] Titre de l'article [Cha09] de Chapman.

- Une traduction qui justifie une forme de polymorphisme ad-hoc via un opérateur de pattern-matching sur les types. À l'aide de cet opérateur, on peut par exemple définir une fonction de type $\Pi\, A.\ A \to A$ dont le comportement dépend du type A passé en argument.

- La traduction de paramétricité. Nous donnons plusieurs variantes dans les façons de la formuler, dont une, la variante « généreuse » qui nie l'univalence.

Nous montrerons aussi comment ces traductions nient ou préservent certains axiomes tels l'univalence, l'extensionnalité des fonctions, etc.
Ce chapitre a été en grande partie publié dans l'article *The next 700 syntactical models of type theory* (Boulier, Pédrot, Tabareau) [BPT17] publié à CPP 2017.

Chapitre 3

Dans un troisième chapitre nous présentons Template-Coq, un plugin de métaprogrammation pour l'assistant de preuve Coq, ainsi que son application à l'implémentation des traductions de programme.
La métaprogrammation est l'art d'écrire des programmes — écrits dans un métalangage — qui manipulent d'autres programmes — écrits dans un langage objet. Dans le contexte de la théorie des types dépendants, le langage est suffisamment riche pour décrire sa propre syntaxe, langage objet et métalangage peuvent donc être les mêmes. Template-Coq est un plugin pour Coq, originellement développé par Malecha, qui donne une représentation des termes de Coq sous la forme d'un inductif et fournit un mécanisme pour passer d'un terme Coq à sa syntaxe (quoting) et réciproquement (unquoting).
La représentation de la syntaxe est aussi proche que possible de celle réellement implémentée par Coq. Cela permet de programmer en Coq des fonctions manipulant des termes Coq comme il est courant de faire en Lisp. Par exemple, on peut programmer une fonction générant le code d'une fonction show — qui convertit un terme d'un inductif vers une chaîne de caractère — à partir de la représentation de l'inductif. Cette approche peut être vue comme une version en théorie des types de designs de programmation fonctionnelle tels Template Haskell ou MetaML.
Récemment, la réification de Template-Coq a été étendue au typage et aux commandes de Coq. Le typage, ainsi que la réduction, sont donnés sous la forme de familles inductives qui expriment que deux termes sont convertibles ou que, dans un contexte Γ, un terme t est de type A. Cela permet de donner une réelle sémantique aux termes réifiés.
Cela se rapproche des travaux de Altenkirch et Kaposi et de Chapman qui décrivent la syntaxe et la sémantique d'une théorie des types en Agda. Dans ces travaux, seuls les termes bien typés font partie de la syntaxe et l'on travaille directement sur les arbres de dérivation du jugement de typage plutôt que sur des termes non typés. Cependant, l'utilisation de types inductifs-inductifs dans un cas et inductifs-quotient dans l'autre nécessite un métalangage avec ces possibilités, ce qui n'est pas le cas de Coq. Mais surtout, et cela est plus problématique en pratique, l'encapsulation du typage dans la syntaxe rend son utilisation difficile car une fonction qui produit un terme doit obligatoirement venir avec une preuve qui assure qu'elle respecte le typage.

Comme exemple de fonction manipulant la syntaxe, Template-Coq implémente une fonction d'inférence de type (cette fonction prend en entrée un contexte et un terme et fournit le type du terme si celui-ci est inférable) et une fonction de typechecking.
L'implémentation de ces fonctions suit leur implémentation en OCaml dans le code source de Coq. Elle est donc un premier pas vers le bootstrap (*i.e.* écrire Coq en Coq). Le bootstrap n'est pas intéressant seulement pour le principe mais parce qu'il permettrait de certifier réellement des parties du noyau de Coq. Par exemple, l'on pourrait certifier que si la fonction d'inférence renvoie un type alors son argument est effectivement typable par ce type. Barras a déjà fait un travail similaire pour un petit sous-ensemble de Coq.
Template-Coq réifie aussi des commandes de Coq à l'aide d'une monade. Les commandes sont les opérations qui permettent de modifier l'état global courant de Coq : ajouter une définition globale, déclarer un inductif, changer un paramètre de configuration, … La réification des commandes ouvre un nouveau champ des possibles puisqu'elle permet l'écriture de plugins Coq en Coq, ce qui rend l'écriture de plugin plus simple. Nous donnerons un exemple de plugin : un plugin qui ajoute un constructeur à un inductif.
Enfin, nous utiliserons Template-Coq pour implémenter un plugin pour les traductions de programme. Ce plugin permet, étant donné un terme ou un type, de calculer son interprétation dans le modèle syntaxique considéré. Cela permet de justifier des axiomes que l'on ajoute à Coq en vérifiant que leur interprétation est bien habitée. La réification du typage fournie par Template-Coq permet de certifier qu'une traduction préserve bien le typage. Implémenter les traductions en Template-Coq permet de n'avoir qu'une seule implémentation qui est à la fois utilisable et certifiable. Pour l'instant nous avons implémenté quelques traductions simples dont la paramétricité mais nous ne les avons pas encore certifiées.
Ce chapitre a été publié en grande partie dans l'article *Towards Certified Meta-Programming with Typed Template-Coq* (Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau and Nicolas Tabareau) [ABC+18] publié à ITP 2018.

Chapitre 4

Dans ce dernier chapitre, nous nous intéresserons à la théorie des types homotopique, HoTT, et plus précisément aux théories « à deux niveaux » permettant de faire cohabiter une égalité stricte et une égalité univalente.
Comme déjà évoqué, il y a deux égalités radicalement opposées mais toutes les deux intéressantes en théorie des types : l'égalité stricte et l'égalité univalente. L'égalité stricte est très syntaxique : deux objets sont égaux s'ils sont syntaxiquement les mêmes. À l'inverse, l'égalité univalente est plus sémantique : deux objets sont égaux s'ils sont isomorphes. Malheureusement ces deux égalités sont incompatibles.
Voevodsky a été le premier à proposer une théorie qui fait cohabiter les deux en introduisant la notion de type *fibrant*. Ce système, appelé Homotopy Type System, reflète en fait ce qui se passe dans le modèle simplicial de la théorie des types homotopiques. Dans celui-ci, les types sont interprétés par des ensembles simpliciaux qui sont fibrants et l'égalité homotopique s'applique à ceux-ci. L'égalité stricte est interprétée par l'égalité de la métathéorie.
La même interprétation des égalités peut être faite dans le modèle cubique de Coquand *et. al.* où les ensembles simpliciaux fibrants sont remplacés par des ensembles cubiques fibrants.

Récemment, Orton et Pitts ont reformulé le modèle cubique dans une théorie des types supportant un intervalle, c'est-à-dire une théorie avec un type clos $\mathbb{I}$ satisfaisant neuf axiomes. On peut alors, dans cette théorie, définir une catégorie de familles qui correspond au modèle cubique. Pour cela, il faut définir la notion de fibrance à partir de l'intervalle, l'interprétation des types identité, etc.
Dans une première partie de ce chapitre, nous proposons une relecture du travail de Orton et Pitts. En mettant en avant la notion de fibrance dégénérée, nous remarquons qu'ils est possible d'interpréter directement une variante de Homotopy Type System dans la théorie des types avec intervalle, sans passer par une catégorie de familles.
La notion de fibrance dégénérée a un autre avantage : elle admet un remplacement fibrant. Dans une seconde partie, nous définissons ce remplacement fibrant et en explorons les conséquences. En particulier, nous définissons une structure de modèle sur l'univers des prétypes. Ce qui, à travers l'interprétation dans le modèle cubique, donne une caractérisation d'une structure de modèle sur la catégorie des ensembles cubiques. Au passage, nous montrerons comment le remplacement fibrant peut être utilisé pour donner l'interprétation de certains types inductifs supérieurs (HIT) dans le modèle cubique.
La fibrance dégénérée a cependant une grosse faiblesse qui est que l'univers des types dégénérément fibrants n'est pas univalent. Donc pour avoir l'univalence, l'on a toujours besoin de l'univers des types pleinement fibrants, et celui-ci n'est pas définissable de façon interne dans la théorie des types avec un intervalle.

Formalisations

Plusieurs formalisations, soit en Coq soit en Agda, sont présentées dans cette thèse. Elles sont disponibles à l'adresse suivante, chacune dans un dossier séparé :

`https://gitlab.inria.fr/sboulier/thesis-formalizations`

À chaque fois que nous citons un fichier, cela se réfère à ce dépôt. Les noms sont cliquables et sont des liens direct vers ce dépôt.

1. Our ship, Martin Löf Type Theory

This chapter introduces Martin Löf Type Theory: its syntax, some constructions in it, and some of its models. First, we give a formal system, $\lambda\Pi_\omega$, which is a core system for Martin Löf Type Theory. It is quite minimal: it features only dependent products and a hierarchy of universes. Section 1.1 is devoted to its introduction. We then consider several separate extensions of this system: some inductive types, cumulativity, an impredicative universe, ... We present them in a modular way in Section 1.2. And we quickly compare our idealized system with the theories implemented by Coq and Agda in Section 1.3. Throughout this thesis, we will often refer to Homotopy Type Theory and some of its constructions. We introduce them in Section 1.4. Last we will introduce Categories with Families, a framework of models for $\lambda\Pi_\omega$ (Section 1.5) and give an account of a model which we are particularly interested in: the setoid model (Section 1.6).

This chapter is not intended to be an introduction to the presented results, but rather: 1. a place to fix notations (that's unavoidable). 2. a recollection of results which make sense together.

1.1. $\lambda\Pi_\omega$

Let us introduce $\lambda\Pi_\omega$, our minimal type theory featuring only Π types and universes. Its name comes from the $\lambda\Pi$-calculus—the system corresponding to first order logic through the Curry-Howard correspondence—which has only Π types, or *dependent products*, and from the fact it has ω universes. We use the framework of Pure Type Systems (PTS), which is a popular formalism introduced in 80s to study several λ-calculi in a unified framework.
A PTS is a type system parametrized by a set of *sorts* $\mathcal{S}$ and two relations:

- $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ the *axioms* of the system, used to type sorts
- $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ the *rules*, used to type dependent products

In a Pure Type System, terms and types belong to the same syntactic category, their syntax is given by:

$$\begin{aligned} A, B, M, N &::= x \mid s \mid M\ N \mid \lambda x : A.\ M \mid \Pi x : A.\ B && \text{(types and terms)} \\ \Gamma, \Delta &::= \cdot \mid \Gamma, x : A && \text{(contexts)} \end{aligned}$$

where $s \in \mathcal{S}$ is a sort and x is a variable name. We will say *pre*terms, or *raw syntax*, when we want to emphasize the fact that the terms are not necessarily well-typed.

There is a notion of computation on preterms given by the β-reduction. It is the compatible closure of:

$$(\lambda\, x : A.\, M)\; N \to_\beta M\{x \coloneqq N\}$$

The conversion relation $M \simeq_\beta N$ is the least equivalence relation containing $\to_\beta$.
The typing rules of a PTS are standard (see *e.g.* [SU06]), they are given by:

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A : s}{\vdash \Gamma, x : A} \qquad \frac{\vdash \Gamma \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\vdash \Gamma \qquad (s, s') \in \mathcal{A}}{\Gamma \vdash s : s'}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s' \qquad (s, s', s'') \in \mathcal{R}}{\Gamma \vdash \Pi\, x : A.\, B : s''}$$

$$\frac{\Gamma, x : A \vdash M : B \qquad \Gamma \vdash \Pi\, x : A.\, B : s}{\Gamma \vdash \lambda\, x : A.\, M : \Pi\, x : A.\, B} \qquad \frac{\Gamma \vdash M : \Pi\, x : A.\, B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\; N : B\{x \coloneqq N\}}$$

$$\frac{\Gamma \vdash M : B \qquad \Gamma \vdash A : s \qquad A \simeq_\beta B}{\Gamma \vdash M : A}$$

We write $A \to B$ for $\Pi\, x : A.\, B$ when x does not occur free in B.

Now, $\lambda\Pi_\omega$ is the PTS given by the following signature:

$$\begin{aligned}\mathcal{S} &\coloneqq \mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \ldots \ (\mathcal{U}_i \text{ for } i : \mathbb{N})\\ \mathcal{A} &\coloneqq (\mathcal{U}_0 : \mathcal{U}_1), (\mathcal{U}_1 : \mathcal{U}_2), \ldots\\ \mathcal{R} &\coloneqq (\mathcal{U}_i, \mathcal{U}_j, \mathcal{U}_{\max(i,j)})\end{aligned}$$

A few remarks about $\lambda\Pi_\omega$:

- $\lambda\Pi_\omega$ is a *full* PTS: for all $s, s' \in \mathcal{S}$ there exists $s'' \in \mathcal{S}$ such that $(s, s', s'') \in \mathcal{R}$ (*i.e.* all products are well sorted), and every sort has a sort: for all $s \in \mathcal{S}$, there exists $s' \in \mathcal{S}$ such that $(s, s') \in \mathcal{A}$. As such, $\lambda\Pi_\omega$ enjoys a strong form of *type correctness*:

  $$\text{If } \Gamma \vdash M : A \text{ then } \exists\, s, \Gamma \vdash A : s$$

  Also, the right premise in the rule of the λ is unnecessary. See for example [Pol92, Section 4].

- $\lambda\Pi_\omega$ is a *functional* PTS: for all $s \in \mathcal{S}$, there is at most one $s' \in \mathcal{S}$ such that $(s, s') \in \mathcal{A}$ and for all $s, s' \in \mathcal{S}$, there is at most one $s'' \in \mathcal{S}$ such that $(s, s', s'') \in \mathcal{R}$. As such, $\lambda\Pi_\omega$ enjoys *type uniqueness*:

  $$\text{If } \Gamma \vdash M : A \text{ and } \Gamma \vdash M : A', \text{ then } A \simeq_\beta A'.$$

  This property will be weakened when we will introduce cumulativity.

Typed vs untyped conversion In some presentation of type theory, the conversion is typed. There is a judgment:

$$\Gamma \vdash M \simeq_\beta N : A$$

meaning that M and N are terms of type A and are convertible. For Pure Type Systems, it has been shown that the two presentations are equivalent by Herbelin and Siles [SH12] relying on previous works by Adams. This result is far from being trivial because in the untyped setting, conversion can go through ill-typed terms.
However, the typed conversion can be unavoidable when considering some extensions. For instance, the η-rule of unit type:

$$\frac{\Gamma \vdash M, N : \mathbb{1}}{\Gamma \vdash M \simeq_\beta N : \mathbb{1}}$$

Judgment for type well-formedness In some presentations of type theory, there is a judgment to express that a type is well formed:

$$\Gamma \vdash A$$

Often, it comes with the fact that terms and types do not belong to same syntactic category. Such a presentation is heavier (all rules for types have to be duplicated in presence of universes). However, it is simpler to interpret such a type theory in models, because it decorrelates well-formedness (the judgment $\Gamma \vdash A$) and smallness properties (the fact of being in $\mathcal{U}_i$ rather that in $\mathcal{U}_j$). We will make the assumption that such a presentation, with a corresponding embedding, can always be found.

Russell vs Tarski universes There are two ways to introduce universes in Martin Löf Type Theory, either *à la Russell* or *à la Tarski*. In Tarski style, a distinction is made between types and terms inhabiting a universe while they are the same in Russell style.
In Russell style, we thus have:

$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_j}{\Gamma \vdash \prod x : A.\, B : \mathcal{U}_k} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}$$

While in Tarski style, we have:

$$\frac{\Gamma \vdash a : \mathcal{U}_i}{\Gamma \vdash \mathbf{El}_i\ a} \qquad \frac{\Gamma \vdash a : \mathcal{U}_i \qquad \Gamma, x : \mathbf{El}_i\ a \vdash b : \mathcal{U}_j}{\Gamma \vdash \pi_{i,j,k}\,(x : \mathbf{El}_i\ a).\ b : \mathcal{U}_k} \qquad \frac{\Gamma \vdash}{\Gamma \vdash u_i : \mathcal{U}_{i+1}}$$

$$\mathbf{El}_k\ (\pi_{i,j,k}\,(x : \mathbf{El}_i\ a).\ b) \simeq_\beta \prod x : \mathbf{El}_i\ a.\ \mathbf{El}_j\ b \qquad \mathbf{El}_{i+1}\ u_i \simeq_\beta \mathcal{U}_i$$

In Tarski style, elements of a universe $\mathcal{U}$ are called *codes* and the operator $\mathbf{El}$ is the *decoding* function. Pure Type System use Russell style and there is even no judgment to express that a type is well-formed ($\Gamma \vdash A$).

Russell style is often seen as informal style and Tarski style as its formal version. In [Ass14], Assaf proves that they are indeed equivalent. The result still holds in presence of cumulativity but much care has to be taken in the design of coercions (see below).
Type systems often use Russell style (it is the case for $\lambda\Pi_\omega$, Coq and Agda) but models often use Tarski style (Categories with Families for instance, see Section 1.5) that's why the translation of Assaf from Russell to Tarski is very useful.

1.2. Extensions of $\lambda\Pi_\omega$

$\lambda\Pi_\omega$ is a minimal framework, and usable type theories, such as those implemented by proof assistants, always enjoy much more features. We review here some of such extensions.

1.2.1. Core extensions

Cumulativity

Cumulativity is the fact that any type at level i is also a type at level j for j greater than i. With cumulativity, $\mathcal{U}_i$ is not only of type $\mathcal{U}_{i+1}$:

$$\mathcal{U}_i : \mathcal{U}_{i+1}$$

but also included in it:

$$\mathcal{U}_i \subseteq \mathcal{U}_{i+1}$$

Cumulativity is studied in the framework of Cumulative Type Systems introduced by Barras [Bar99, BG05]. In this setting, cumulativity is defined with a subtyping relation. Here we will only consider a weaker notion of cumulativity given by the rule:

$$\frac{\Gamma \vdash A : s}{\Gamma \vdash A : \mathcal{N}(s)}$$

where for each sort s, $\mathcal{N}(s)$ is the unique sort such that $(s, \mathcal{N}(s)) \in \mathcal{A}$. It is weaker because we don't have that a term of type $A \to s$ is also of type $A \to \mathcal{N}(s)$.
We will write $\lambda\Pi_\omega + \subseteq$ to denote our system extended with cumulativity.

Remark. In Tarski style, cumulativity is introduced by *coercion* functions $\uparrow_i : \mathcal{U}_i \to \mathcal{U}_{i+1}$.

$$\frac{\Gamma \vdash a : \mathcal{U}_i}{\Gamma \vdash \uparrow_i a : \mathcal{U}_{i+1}}$$

To preserve the equivalence between Russell and Tarski styles, equations on coercions have to be considered such as:

$$\pi_{1,1,1}\,(x : \mathbf{El}_1\ a).\ \uparrow_1 b \ \simeq_\beta\ \pi_{1,0,1}\,(x : \mathbf{El}_1\ a).\ b$$

See [Ass14] for the complete set of rules.
In Russell style, the cumulativity rule is not syntax-directed. On the contrary, thanks to coercions, it is the case in Tarski style.

η-conversion

The conversion relation $\simeq_\beta$ that we introduced so far only considers β-reduction. However, it is common to consider richer relations. And first of all, the η-reduction:

$$\lambda\, x : A.\ (M\ x) \to_\eta M$$

We write $M \simeq_{\beta\eta} N$ for the least congruence containing $\to_\beta$ and $\to_\eta$.
Unfortunately, η-conversion breaks the Church-Rosser property on preterms. Here is a well-known example due to Nederpelt. Let M be $\lambda\, x : A.\ (\lambda\, y : B.\ y)\ x$, then $M \to_\beta \lambda\, x : A.\ x$ and $M \to_\eta \lambda\, y : B.\ y$. Both terms are in normal form but they are not convertible if A and B are not. Hopefully, Church-Rosser remains true on well-typed terms in strongly normalizing systems such as $\lambda\Pi_\omega$. See [Geu92] for a more detailed account.
The η-conversion also interacts badly with cumulativity: the combination of both breaks subject reduction: if f is of type $\mathcal{U}_1 \to \mathcal{U}_0$, then $\lambda\, x : \mathcal{U}_0.\ f\ x$ is of type $\mathcal{U}_0 \to \mathcal{U}_1$ and η-reduces to f of type $\mathcal{U}_1 \to \mathcal{U}_0$ (example taken from [Her09]).

Remark. The η-conversion makes the type checking harder, that's why many works on the topic use a typed conversion. In Coq, conversion is untyped. To deal with η-conversion, Coq implements the rules:

$$\frac{M \simeq_{\beta\eta} N\ x}{\lambda\, x : A.\ M \simeq_{\beta\eta} N} \qquad \frac{M\ x \simeq_{\beta\eta} N}{M \simeq_{\beta\eta} \lambda\, x : A.\ N}$$

[Gog05] reviews several algorithms to deal with $\simeq_{\beta\eta}$.

There is also a notion of η-reduction for some other type constructors such as dependent sums and unit, we will introduce them later.

Impredicative universe

We introduce a new sort $\mathbb{P}$ called "Prop" the universe of *propositions*:

$$\mathcal{S} \coloneqq \ldots \mid \mathbb{P}$$

This universe has type $\mathcal{U}_0$ (that is why it is sometimes noted $\mathcal{U}_{-1}$)[1]:

$$\mathcal{A} \coloneqq \ldots \mid (\mathbb{P} : \mathcal{U}_0)$$

This universe features *impredicativity*, meaning that, whatever the universe level of A, if B lives in $\mathbb{P}$, so does $\Pi\, x : A.\ B$:

$$\mathcal{R} \coloneqq \ldots \mid (\mathbb{P}, \mathcal{U}_i, \mathcal{U}_i) \mid (s, \mathbb{P}, \mathbb{P}) \text{ for any sort } s$$

The system $\lambda\Pi_\omega + \subseteq + \mathbb{P}$ is called CC_ω, the Calculus of Construction with ω universes, in reference to the Calculus of Constructions of Coquand and Huet [CH86]. There are several variants of CC_ω depending on whether the cumulativity passes under products or not.

[1] In Coq, $\mathbb{P}$ has type $\mathcal{U}_1$ for historical reasons.

There are two features which can be expected for an impredicative universe: proof irrelevance and propositional extensionality. *Proof irrelevance* is the fact that all proofs of a proposition are equal:

$$\mathbf{PI} \coloneqq \prod (A : \mathbb{P})\,(p, q : A).\ p =_A q$$

Propositional extensionality is the fact that logically equivalent propositions are equal:

$$\mathbf{PropExt} \coloneqq \prod A\ B : \mathbb{P}.\ (A \to B) \to (B \to A) \to A =_{\mathbb{P}} B$$

Proposition extensionality implies proof irrelevance because an inhabited proposition is equal to $\top$ (or any other contractible type) which has all its inhabitants equal.
Both axioms are independent of $\lambda\Pi_\omega$. For propositional extensionality, a proof of independence is given in Section 2.2.3.

1.2.2. Inductive types

To do some interesting mathematics, we need some data structures to reason about them. The main way to introduce new data structures in type theory are inductive types. Some others are definitions (in particular with impredicative encodings) and coinductive types. Inductive types are generated by *constructors*, and their *eliminators* allow to reason on them by structural induction. There are some general schemes describing mutual inductive families [Tea], or induction-recursion [Dyb00]. We will not bother with such a generality and only consider some examples.

Natural numbers The type of natural numbers is the archetype of inductive types. $\mathbb{N}$ is a closed type which has two constructors zero, written 0, and successor, written S. It is described by the following typing rules:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbb{N} : \mathcal{U}_i} \qquad \frac{\Gamma \vdash}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash S : \mathbb{N} \to \mathbb{N}}$$

$$\frac{\Gamma \vdash P : \mathbb{N} \to s \qquad \Gamma \vdash t_0 : P\,0 \qquad \Gamma \vdash t_S : \prod n : \mathbb{N}.\ P\,n \to P\,(S\,n)}{\Gamma \vdash \mathbf{nat\_ind}(P, t_0, t_S) : \prod n : \mathbb{N}.\ P\,n}$$

The β-reduction for $\mathbb{N}$ is given by the rules:

$$\mathbf{nat\_ind}(P, t_0, t_S)\,0 \to_\beta t_0 \qquad \mathbf{nat\_ind}(P, t_0, t_S)\,(S\,t) \to_\beta t_S\,(\mathbf{nat\_ind}(P, t_0, t_S)\,t)$$

We chose to have $\mathbb{N}$ of type $\mathcal{U}_i$ and not only $\mathcal{U}_0$—that is to say to have $\mathbb{N}$ *universe polymorphic*—in order to be usable even without cumulativity (as in Agda for instance).

0, 1 and 2 We will need the empty type $\mathbf{0}$, the unit type $\mathbb{1}$ and the type of booleans $\mathbb{B}$. They have respectively zero, one and two elements.
The empty type has no constructor and its eliminator allows to define a term of every type:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{0} : \mathcal{U}_i} \qquad \frac{\Gamma \vdash A : s \qquad \Gamma \vdash t : \mathbf{0}}{\Gamma \vdash !^A t : A}$$

The unit type as only one constructor $\mathtt{tt}$:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbb{1} : \mathcal{U}_i} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathtt{tt} : \mathbb{1}} \qquad \frac{\Gamma \vdash P : \mathbb{1} \to s \qquad \Gamma \vdash t_{\mathtt{tt}} : P\ \mathtt{tt}}{\Gamma \vdash \mathbf{unit\_ind}(P, t_{\mathtt{tt}}) : \prod t : \mathbb{1}.\ P\ t}$$

$$\mathbf{unit\_ind}(P, t_{\mathtt{tt}})\ \mathtt{tt} \to_\beta t_{\mathtt{tt}}$$

The type of booleans has two constructors, its eliminator is written with an if-then-else notation (although dependently typed):

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbb{B} : \mathcal{U}_i} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{true} : \mathbb{B}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{false} : \mathbb{B}}$$

$$\frac{\Gamma \vdash b : \mathbb{B} \qquad \Gamma \vdash P : \mathbb{B} \to s \qquad \Gamma \vdash t_{\mathbf{true}} : P\ \mathbf{true} \qquad \Gamma \vdash t_{\mathbf{false}} : P\ \mathbf{false}}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{ret}\ P\ \mathbf{then}\ t_{\mathbf{true}}\ \mathbf{else}\ t_{\mathbf{false}} : P\ b}$$

$$\mathbf{if}\ \mathbf{true}\ \mathbf{ret}\ P\ \mathbf{then}\ t_{\mathbf{true}}\ \mathbf{else}\ t_{\mathbf{false}} \to_\beta t_{\mathbf{true}} \qquad \mathbf{if}\ \mathbf{false}\ \mathbf{ret}\ P\ \mathbf{then}\ t_{\mathbf{true}}\ \mathbf{else}\ t_{\mathbf{false}} \to_\beta t_{\mathbf{false}}$$

Σ types Σ types, or *dependent sums*, generalize the (non-dependent) cartesian product $A \times B$. They are the first example of parametrized inductive types, *i.e.* inductive type in a non-empty context. Here we introduce *negative* Σ types which have two projections π_1 and π_2. *Positive* Σ types would have only one eliminator given by pattern-matching.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_j}{\Gamma \vdash \Sigma\, x : A.\ B : \mathcal{U}_{\max(i,j)}}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B\{x \coloneqq N\} \qquad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash (M, N) : \Sigma\, x : A.\ B} \qquad \frac{\Gamma \vdash M : \Sigma\, x : A.\ B}{\Gamma \vdash \pi_1(M) : A}$$

$$\frac{\Gamma \vdash M : \Sigma\, x : A.\ B}{\Gamma \vdash \pi_2(M) : B\{x \coloneqq \pi_1(M)\}} \qquad \pi_1((M, N)) \to_\beta M \qquad \pi_2((M, N)) \to_\beta N$$

We write $A \times B$ for $\Sigma\, x : A.\ B$ when x does not occur in B.
At some point, we will also consider the η-rule of Σ types:

$$(\pi_1(M), \pi_2(M)) \to_\eta M$$

This rule is called *surjective pairing*.
The Extended Calculus of Constructions (ECC) of Luo [Luo89] is $\lambda\Pi_\omega + \subseteq + \mathbb{P} + \Sigma$.

1.2.3. Propositional equality

Equality is one of the intricate features of type theory, and it is still heavily investigated. We distinguish *propositional equality* $M = N$ which is the relation induced by a notion of *identity*

types (which are the types of proofs of this equality), from *definitional equality* $M \simeq_\beta N$, the relation induced by conversion, which is much more syntactic. In general, definitional equality implies propositional equality but the converse does not hold, except in Extensional Type Theory where they coincide (see below).
Identity types were first introduced by Martin-Löf and then reformulated as an inductive family by Paulin-Mohring. Using a Coq syntax, this inductive family is given by:

$$\begin{aligned}&\textbf{Inductive}\ \mathsf{eq}\ (A : \mathcal{U})\ (x : A) : A \to \mathcal{U} :=\\ &\mid\ \mathbf{refl}_{=}\ :\ \mathsf{eq}\ A\ x\ x\end{aligned}$$

This can be rephrased by the following rules:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash t, u : A}{\Gamma \vdash t =_A u : \mathcal{U}_i} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathbf{refl}_{=}\, t : t =_A t}$$

$$\frac{\Gamma \vdash p : t =_A u \qquad \Gamma,\, y : A,\, q : t =_A y \vdash P : \mathcal{U}_i \qquad \Gamma \vdash v : P\{y := t,\, q := \mathbf{refl}_{=}\, t\}}{\Gamma \vdash J_{=}(y.q.P,\, p,\, v) : P\{y := u,\, q := p\}}$$

$$J_{=}(y.q.P,\, \mathbf{refl}_{=}\, t,\, v) \to_\beta v$$

The only constructor is $\mathbf{refl}_{=}$ which asserts that two convertible terms are propositionally equal. $J_{=}$ is the elimination principle, it contains the non-dependent elimination principle called, *Leibniz principle*, or *transport*:

$$\mathbf{transport}_A : \Pi\,(P : A \to s)\,(x,\, y : A).\ x =_A y \to P\,x \to P\,y$$

We will write $p\,\#^P\,u := \mathbf{transport}\ P\ \_\ \_\ p\,u$ the transport of u along p in P.
Even if the constructor only allows to equate convertible terms, propositional equality is already more powerful than conversion because of the ability to refer to the type of equalities and to make equality assumptions. For instance we can prove equalities like:

$$\Pi\, n : \mathbb{N}.\ n + 0 =_{\mathbb{N}} n$$

which does not hold definitionally if $+$ is defined by induction on its left argument. The proof goes by induction on the natural number.

While already powerful, we can ask for more. They are two main directions in which propositional equality can be pushed: toward a *strict equality* or toward a *univalent equality*. Strict equality enjoys UIP while the univalent one enjoys univalence. In the rest of this thesis we write $M \equiv_A N$, $\mathbf{refl}_{\equiv}$ and $J_{\equiv}$ for a propositional equality intended to be strict (even it does not satisfy any additional axioms) and $M =_A N$, $\mathbf{refl}_{=}$ and $J_{=}$ for one intended to be univalent. The notation # will refer to transport of either one equality or the other depending on the context.
In Chapter 4 we will introduce Two Level Type Theory, a type theory in which strict and univalent equalities live together.

Strict equality

Strict equality is an equality close to conversion. In the empty context, it holds when the compared terms are syntactically the same.
The typical feature of strict equality is *Uniqueness of Identity Proofs* (UIP), this principle asserts that all proofs of strict equality are equal. It can be stated as the following axiom:

$$\mathbf{UIP}_i \coloneqq \Pi\,(A : \mathcal{U}_i)\,(M,\ N : A)\,(p,\ q : M \equiv_A N).\ p \equiv_{M \equiv N} q$$

This axiom is universe polymorphic as it depends on the universe level i. So, in fact, it is rather an axiom schema, intended to hold for each level. It will be the same for many other axioms.
UIP has been shown to be independent of Martin Löf Type Theory by Hofmann and Streicher [HS98] with the groupoid interpretation of type theory. In this model, each type is interpreted by a groupoid (a category in which every morphism is invertible) and identity types are interpreted by the sets of morphisms. This model negates UIP because there are groupoids with non-equal parallel arrows.
It is worth remarking that UIP looks like proof irrelevance. And indeed, in Coq, the equality is typed in $\mathbb{P}$ (whatever the sort of A). This requires singleton elimination to enable elimination of a proof of equality toward a type which is not in $\mathbb{P}$. See the the Coq Manual [Tea] for details.

Another axiom which can be required for strict equality is *function extensionality*, meaning that two pointwise equal functions are equal (which is the way functions are compared in set theory):

$$\mathbf{FunExt}_{i,j} \coloneqq \Pi\,(A : \mathcal{U}_i)\,(B : A \to \mathcal{U}_j)\,(f,\ g : \Pi\,x : A.\ B\,x).\ (\Pi\,x.\ f\ x \equiv g\ x) \to f \equiv g$$

Extensional Type Theory We emphasized the fact that strict equality is "close to conversion". *Extensional Type Theory* (ETT) pushes all the way this principle and includes strict equality in the conversion by the *reflection rule*:

$$\frac{\Gamma \vdash e : M \equiv_A N}{\Gamma \vdash M \simeq_\beta N : A}$$

The reflection rule can greatly simplify some proof terms. But alas, the reflection rule makes type checking undecidable. That is why few proof assistants try to implement ETT. Conversely, a type theory without the extension rule is called *Intensional Type Theory*.
It has been shown that Extensional Type Theory can be translated toward Intensional Type Theory by Oury [Our05] and Winterhalter *et. al.* [WST18]. In this last work, from a typing derivation $\Gamma \vdash_{\text{ETT}} t : A$, they get Γ', t' and A'—which are decorated versions of the original terms with some transports—such that $\Gamma' \vdash t' : A'$ is derivable in Martin Löf Type Theory with a strict equality enjoying FunExt and UIP. The need for FunExt and UIP in the target theory is unavoidable because both axioms follow from the reflection rule.
In light of this result, conversion can be seen as a subset of strict equality that is taken as big as possible while retaining a decidable type checking. Remark: in between, there is conversion with η on recursive types (*e.g.* $\mathbb{N}$).

Univalent equality

More recently, equality has been pushed in another, totally different, direction: the beautiful world of *Homotopy Type Theory* (HoTT). This time, equality is very semantical and this notion of equality is close to the notion of isomorphism. This intuition is especially valid for equality between types because univalence axiom implies that an equality between two types $A =_{\mathcal{U}} B$ is an equivalence, *i.e.* an "isomorphism of types".
In HoTT, we don't have UIP anymore so we can have two proofs of an equality which are not necessarily equal:

$$p,\, q : x =_A y \qquad p \stackrel{?}{=} q$$

This corresponds to the fact that between two types, there can be several equivalences which are not equal.
The distinctive feature of a univalent equality is, of course, the *univalence* axiom. We will introduce it in section 1.4. Let's remark here that univalence implies function extensionality. FunExt is hence a common feature of strict and univalent equality. Propositional extensionality is a form of univalence for propositions of $\mathbb{P}$.

Different notions of equivalence The different notions of equality implies different notions of equivalence between types. Let $f : A \to B$ be a map such that there exists $g : B \to A$ such that

$$g \circ f \bowtie \mathbf{id}_A \qquad \text{and} \qquad f \circ g \bowtie \mathbf{id}_B$$

We will say that f is:

- a *type isomorphism* if $\bowtie$ is the conversion $\simeq_\beta$.
  We will write $A \cong_\beta B$. *E.g.* $A \times B \cong_\beta B \times A$ (with surjective pairing).
- a *strict equivalence* if $\bowtie$ is the strict equality $\equiv$.
  We will write $A \cong B$. *E.g.* $\mathbb{N} \cong \mathbf{list}\ \mathbb{B}$.
- an *equivalence* if $\bowtie$ is the univalent equality $=$.
  We will write $A \simeq B$. *E.g.* $(A \to \mathcal{U}) \simeq (\Sigma\, B : \mathcal{U}.\, B \to A)$.

Additionally, A and B will be said *logically equivalent*, written $A \leftrightarrow B$, if the two maps $A \to B$ and $B \to A$ exist without further requirements. This is especially meaningful when A and B are propositions or h-propositions.
In fact, we will redefine the *equivalences* with a logically equivalent definition, see Section 1.4.

1.2.4. Streams

Let's introduce streams, which will be our example of coinductive type. Streams of type A are infinite lists of elements of type A. They have two destructors *head* and *tail* and can be built by *corecursion*.

$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash \mathbf{stream}\ A : \mathcal{U}_i} \qquad \frac{\Gamma \vdash M : \mathbf{stream}\ A}{\Gamma \vdash \mathbf{hd}\ M : A} \qquad \frac{\Gamma \vdash M : \mathbf{stream}\ A}{\Gamma \vdash \mathbf{tl}\ M : \mathbf{stream}\ A}$$

$$\frac{\Gamma \vdash S : \mathcal{U}_i \qquad \Gamma \vdash M_0 : S \to A \qquad \Gamma \vdash M_1 : S \to S \qquad \Gamma \vdash N : S}{\Gamma \vdash \mathbf{stream\_corec}\ S\ M_0\ M_1\ N : \mathbf{stream}\ A}$$

$$\mathbf{hd}\ (\mathbf{stream\_corec}\ S\ M_0\ M_1\ N) \to_\beta M_0\ N$$

$$\mathbf{tl}\ (\mathbf{stream\_corec}\ S\ M_0\ M_1\ N) \to_\beta \mathbf{stream\_corec}\ S\ M_0\ M_1 (M_1\ N)$$

Propositional equality often behaves quite poorly on coinductive types, and two streams which are morally the same cannot be proved equal. Therefore, we introduce another notion of equality for streams: *bisimulation*. Two streams are bisimilar when their heads are propositionally equal and their tails are themselves bisimilar.
Bisimulation is thus the coinductive family given by:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash M, N : \mathbf{stream}\ A}{\Gamma \vdash \mathbf{bisim}\ A\ M\ N : \mathcal{U}_i} \qquad \frac{\Gamma \vdash M : \mathbf{bisim}\ A\ N_1\ N_2}{\Gamma \vdash \mathbf{hd_b}\ M : \mathbf{hd}\ N_1 =_A \mathbf{hd}\ N_2}$$

$$\frac{\Gamma \vdash M : \mathbf{bisim}\ A\ N_1\ N_2}{\Gamma \vdash \mathbf{tl_b}\ M : \mathbf{bisim}\ A\ (\mathbf{tl}\ N_1)\ (\mathbf{tl}\ N_2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash S : \mathbf{stream}\ A \to \mathbf{stream}\ A \to \mathcal{U}_i \\ \Gamma \vdash M_0 : \prod s_0\ s_1 : \mathbf{stream}\ A.\ S\ s_0\ s_1 \to \mathbf{hd}\ s_0 =_A \mathbf{hd}\ s_1 \\ \Gamma \vdash M_1 : \prod s_0\ s_1 : \mathbf{stream}\ A.\ S\ s_0\ s_1 \to S\ (\mathbf{tl}\ s_0)\ (\mathbf{tl}\ s_1) \qquad \Gamma \vdash N : S\ P_0\ P_1\end{array}}{\Gamma \vdash \mathbf{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N : \mathbf{bisim}\ A\ P_0\ P_1}$$

$$\mathbf{hd_b}\ (\mathbf{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N) \to_\beta M_0\ P_0\ P_1\ N$$

$$\mathbf{tl_b}\ (\mathbf{bisim\_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N) \to_\beta \mathbf{bisim\_corec}\ S\ M_0\ M_1\ (\mathbf{tl}\ P_0)\ (\mathbf{tl}\ P_1)\ (M_1\ P_0\ P_1\ N)$$

In general, bisimulation does not imply propositional equality, we call *stream extensionality* this fact:

$$\mathbf{StreamExt}_i \coloneqq \prod (A : \mathcal{U}_i)\ (s_1\ s_2 : \mathbf{stream}\ A).\ \mathbf{bisim}\ A\ s_1\ s_2 \to s_1 =_{\mathbf{stream}\ A} s_2$$

We will prove in section 2.2.2 that **StreamExt** is independent of $\lambda\Pi_\omega$.

1.3. Coq and Agda

Coq and Agda are two proof assistants implementing some variants of Martin Löf Type Theory.

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.

Fixpoint plus (n m : nat) : nat
:= match n with
   | O ⇒ m
   | S n ⇒ S (plus n m)
   end.
```

```
data nat : Set where
  O : nat
  S : nat → nat

plus : (n m : nat) → nat
plus O m = m
plus (S n) m = S (plus n m)
```

Figure 1.1.: Coq and Agda definitions of nat and plus

Both have all inductive types, inductive families and even more (coinductive types, inductive-inductive types for Agda, ...). In Coq, elimination of inductive types is given by pattern-matching and fixpoints. In Agda, pattern-matching and recursion are only accessible "at top-level" when defining a function (as in Haskell for instance). We give the example of natural numbers inductive type and plus function in Figure 1.1. Agda pattern-matching is more powerful than the one of Coq because, except in presence of `--witout-K` option, it can proves UIP.

In both proof assistants implicit arguments are written in curly braces { }. It doesn't look like it but implicit arguments and their inference constitute a crucial feature for making a dependently typed language usable. When we write type theory on paper, we will often omit some "implicit arguments" even if we did not declare it.

Agda as an explicit hierarchy of universes:

$$\texttt{Set} : \texttt{Set}_1 : \texttt{Set}_2 \ldots$$

This hierarchy is *not* cumulative. Universe polymorphism has to be introduced by hand with a quantification over `Level`. For instance polymorphic unit type is introduced by:

```
data unit {l : level} : Set l where
tt : unit
```

Agda used to not have a Prop universe but recently[2] Cockx reintroduced it in Agda. This Prop universe enjoys definitional proof irrelevance. We did not have time to try to use it.
In Coq, the hierarchy is cumulative. The universe levels are hidden and handled automatically, this gives "type-in-type in practice", but avoiding Girard paradox. Hence we have

$$\texttt{Type} : \texttt{Type}$$

but in fact the two `Type`s refer to different universes. The universe "$\texttt{Type}_0$" at the bottom of the hierarchy is the only one directly accessible and is written `Set`. There is also an impredicative

[2] Commit 7dda192 on Agda repository, on 2018, May 21
See also https://gist.githubusercontent.com/jespercockx/d7e0885f2078e0c0a54de99117226ac4/raw/da623c143c8922a8459ba136075e6be506b6ba28/PropRezz.agda

universe `Prop`.

We will also use some *records*, both in Agda and Coq. Records are like Σ types but with named fields. There are used to introduce mathematical structures (groups, categories, ...).
Both proof assistants have η-conversion for functions. Agda have η-conversion for all records. In Coq, η-conversion for records is only available for some of them. First, the record has to be declared with `Set Primitive Projections` option. And moreover, as the conversion is untyped in Coq, it is not available for empty and unit records.

To help distinguishing Coq and Agda code we use background colors and .

1.4. Homotopy Type Theory

As we have said, Homotopy Type Theory is the investigation of worlds with higher equalities. We devote it a section because several intuitions of this thesis come from HoTT. Moreover one of the goals of the last chapter of this thesis is to describe the cubical model of Homotopy Type Theory. The unavoidable reference for Homotopy Type Theory is the "HoTT book" [Pro13] which was collectively written.

Dependent elimination gives to identity types a groupoid, and even an ∞-groupoid, structure. Composition and symmetry are written this way:

$$p : x =_A y \vdash p^{-1} : y =_A x$$

$$p : x =_A y,\ q : y =_A z \vdash p \cdot q : x =_A z$$

And they obey groupoidal laws such as "$p \cdot p^{-1} = \mathbf{refl}_{\_}$". There are tons of such groupoidal laws and one difficulty when formalizing theorems in HoTT is to find the right one at each time. Let's emphasize two laws which are particularly useful:

$$\mathbf{ap} : \prod (A,\ B : \mathcal{U}_i)(f : A \to B)(x,\ y : A).\ x =_A y \to f\ x =_B f\ y$$
$$\mathbf{apD} : \prod (A : \mathcal{U}_i)(B : A \to \mathcal{U}_j)(f : \prod x.\ B\ x)(x,\ y : A)(p : x =_A y).\ p\ \#^B\ f\ x =_{B\ y} f\ y$$
$$\mathbf{ap_{10}} : \prod (A,\ B : \mathcal{U}_i)(f,\ g : A \to B)(x : A).\ f =_{A \to B} g \to f\ x =_B g\ x$$

h-Levels A type can have non-trivial higher identity types, to quantify how high they are, HoTT introduces the notion of homotopy level.

– A type A is *contractible*, or *(-2)-truncated*, if it is inhabited and all its elements are equal:

$$\Sigma\, x : A.\ \prod y : A.\ x =_A y$$

We write $\mathbf{Contr}\ A$. Famous examples of contractible types are unit and singleton types:

$$\Sigma\, x : A.\ x =_A x_0 \quad \text{and} \quad \Sigma\, x : A.\ x_0 =_A x$$

This is known as *contractibility of singletons*.

– A type A is an *h-proposition*, or *(-1)- truncated*, if all its elements are equal:

$$\prod x,\ y : A.\ x =_A y$$

– A type A is an *h-set*, or *0-truncated*, if its identity types are h-propositions:

$$\prod (x,\ y : A)\,(p,\ q : x =_A y).\ p =_{x=y} q$$

Types with a decidable equality (such as $\mathbb{N}$, $\mathbb{B}$, ...) are h-sets, this is Hedberg theorem.

The equivalences preserve h-levels: if $A \simeq B$ and if A is n-truncated, then B is n-truncated. The hierarchy is cumulative (if A is n-truncated, then it is $(n+1)$-truncated) and goes on for $n \geq -2$. See Chapter 7 of the HoTT book.

Equivalences We have already defined equivalences in Section 1.2.3 but there was something missing in the definition. A map $f : A \to B$ is said to be an equivalence when the following type is inhabited:

$$\mathbf{IsEquiv}(f) :=$$
$$\Sigma\,(g : B \to A)\,(\eta : \prod x.\ g\ (f\ x) = x)\,(\varepsilon : \prod x.\ f\ (g\ x) = x).\ \prod x.\ \mathbf{ap}\ f\ (\eta\ x) = \varepsilon\ (g\ x)$$

This type is logically equivalent to the previous definition but the additional coherence makes it an h-proposition, which is nice because it makes its inhabitants irrelevant.
An equivalent characterization of equivalences is that they have their fibers contractible:

$$\mathbf{IsEquiv}'(f) := \prod y : B.\ \mathbf{Contr}\ (\Sigma\, x : A.\ f\ x =_B y)$$

We will use this definition in chapter 4.

We say that two types A and B are equivalent, written $A \simeq B$, when there exists a map form A to B which is an equivalence:

$$A \simeq B := \Sigma\, f : A \to B.\ \mathbf{IsEquiv}(f)$$

For every type A the identity map $A \to A$ is an equivalence and thus provides:

$$\mathbf{idequiv}_A : A \simeq A$$

And thus, given a proof of $A =_{\mathcal{U}} B$ we can get an equivalence $A \simeq B$ by transport:

$$\begin{aligned}\mathbf{paths\_to\_equiv}_{A,B} &: A =_{\mathcal{U}} B \to A \simeq B \\ &:= \lambda\, p.\ p\ \#^{\lambda\, B.\ A \simeq B}\ \mathbf{idequiv}_A\end{aligned}$$

At last, we can state univalence which asserts that the only way to build an equality proof between two types is, up to equivalence, to give an equivalence:

$$\mathbf{Univalence}_i := \prod A,\ B : \mathcal{U}_i.\ \mathbf{IsEquiv}\ (\mathbf{paths\_to\_equiv}_{A,B})$$

Univalence is an axiom full of very interesting consequences. Let's only mention one of them:

Proposition 1. Univalence implies function extensionality.

Proof. See [Pro13, Section 4.9]. □

1.4.1. Higher Inductive Types and Quotient Inductive Types

Another important concept of Homotopy Type Theory is Higher Inductive Types. *Higher Inductive Types* (HITs) are inductive types which have constructors not only for elements of the type but also for equalities between elements of the type. *Quotient Inductive Types* (QITs) are their counterpart for strict equality.
The first reference for Higher Inductive Types is still the HoTT book, Chapter 6 this time.
Higher Inductive Types are a very active topic of research. Indeed, both the syntax and the semantics of HITs is still uncertain. They are hence introduced through examples.
One of the simplest examples of HIT is the circle:

$$\begin{aligned}&\textsf{HIT}\ \mathbb{S}^1 : \mathcal{U}_0 \coloneqq \\ &\mid \mathbf{base} : \mathbb{S}^1 \\ &\mid \mathbf{loop} : \mathbf{base} =_{\mathbb{S}^1} \mathbf{base}\end{aligned}$$

The circle $\mathbb{S}^1$ is a closed type. The constant **base** is an element of $\mathbb{S}^1$ and **loop** an equality between **base** and itself. The elimination principle of such an HIT is as follows:

$$\frac{\vdash P : \mathbb{S}^1 \to \mathcal{U} \qquad \vdash \mathrm{base}' : P\ \mathbf{base} \qquad \vdash \mathrm{loop}' : \mathbf{loop}\ \#^P\ \mathrm{base}' \equiv_{P\ \mathbf{base}} \mathrm{base}'}{\vdash \mathbb{S}^1\_\mathbf{ind}(P,\ \mathrm{base}',\ \mathrm{loop}') : \Pi\, w : \mathbb{S}^1.\ P\ w}$$

$$\mathbb{S}^1\_\mathbf{ind}(P,\ \mathrm{base}',\ \mathrm{loop}')\ \mathbf{base} \to_\beta \mathrm{base}' \qquad \mathbf{apD}\ (\mathbb{S}^1\_\mathbf{ind}(P,\ \mathrm{base}',\ \mathrm{loop}'))\ \mathbf{loop} = \mathrm{loop}'$$

Here the computation rule on **loop** is stated with the univalent equality. However when a strict equality is available it is often this one which is used.
Then, there are more complicated HITs. For instance the recursive ones, like propositional truncation for $A : \mathcal{U}_i$:

$$\begin{aligned}&\textsf{HIT}\ \|A\| : \mathcal{U}_i \coloneqq \\ &\mid \iota : A \to \|A\| \\ &\mid \mathbf{eq} : \Pi\, x,\ y : \|A\|.\ x =_{\|A\|} y\end{aligned}$$

The difficulty here is that the quantification in the type of **eq** is on $\|A\|$, the type being defined.
And some introduce higher equalities like the torus:

$$\begin{aligned}&\textsf{HIT}\ \mathbb{T}^2 : \mathcal{U}_0 \coloneqq \\ &\mid \mathbf{b} : \mathbb{T}^2 \\ &\mid \mathbf{p} : \mathbf{b} =_{\mathbb{T}^2} \mathbf{b} \\ &\mid \mathbf{q} : \mathbf{b} =_{\mathbb{T}^2} \mathbf{b} \\ &\mid \mathbf{s} : \mathbf{p} \cdot \mathbf{q} =_{\mathbf{b}=\mathbf{b}} \mathbf{q} \cdot \mathbf{p}\end{aligned}$$

QITs Quotient Inductive Types are the counterpart of HITs for strict equality. They are introduced following the same schema. However, higher equalities are pointless because of

UIP. They satisfy similar elimination principles where univalent equalities are replaced by strict ones.
Particular examples of QIT are *Quotient Types*. Quotient Types were introduced by Hofmann in his thesis [Hof12]. They are as follows. Let $A : \mathcal{U}_i$ be a type and $R : A \to A \to \mathcal{U}_i$ a relation, then A/R is the quotient of A by the equivalence relation generated by R.

$$\begin{aligned}&\textbf{Quotient } A/R : \mathcal{U}_i :=\\ &\mid \iota \ : \ A \to A/R\\ &\mid \textbf{eq} \ : \ \Pi\, x,\, y : A.\ R\, x\, y \to \iota\, x \equiv_{A/R} \iota\, y\end{aligned}$$

A/R is a type of $\mathcal{U}_i$, ι is the projection in the quotient, it is of type $A \to A/R$, and **eq** is of type $\Pi\, x,\, y : A.\ R\, x\, y \to \iota\, x \equiv_{A/R} \iota\, y$ and asserts that the elements of the same equivalence class are equal.
The quotient satisfies the following elimination principle:

$$\frac{\vdash \iota' : \Pi\, x : A.\ P\ (\iota\, x) \qquad \begin{array}{c}\vdash P : A/R \to \mathcal{U}\\ \vdash \mathrm{eq}' : \Pi\, (x,\, y : A)\, (r : R\, x\, y).\ (\textbf{eq}\ r)\ \#\ (\iota'\ x) \equiv \iota'\ y\end{array}}{\vdash \textbf{quo\_ind}(P,\ \iota',\ \mathrm{eq}') : \Pi\, w : A/R.\ P\ w}$$

$$\textbf{quo\_ind}(P,\ \iota',\ \mathrm{eq}')\ (\iota\, x) \ \to_\beta \ \iota'\ x$$

Hence Quotient Types are a simple form of QIT where the relation has to be previously defined. In Section 4.2.2, we will us a QIT which is not a Quotient Type.

Both HITs and QITs break canonicity. This is one of the issues restraining their implementation in proof assistants. Let us notice that Lean proof assistant implements quotient types (at the price of losing canonicity) [AdK15]. However, they are not expected to endanger consistency. For instance, Li [Li15] has shown that the setoid model of Altenkirch [Alt99] supports quotient types.
Let's finish with a remarkable fact. In presence of Quotient Types, FunExt (for $\equiv$) is derivable. And similarly in presence of HITs, FunExt for $=$ is derivable. See for example the HoTT book Section 6.3 for a proof.
A nice overview of the triad HITs / QITs / QTs can be found in Kaposi's thesis [Kap17]. An account of QITs (and even QIITs) has been recently published by Altenkirch *et. al.* [ACD+18].

1.5. Semantic models of $\lambda\Pi_\omega$

In this section we review what is a model of Martin Löf Type Theory, that is to say, what is the structure needed on a space (generally, a category) to interpret a dependent type theory like $\lambda\Pi_\omega$ in it. In particular, a model has to describe how to interpret the conversion and the substitution which are at the heart of dependent type theory.
Display map categories, Categories with attributes, ... there are lots of notions of categorical models for type theory. See [NLaa] for a list, and [ALV17] for a comparison of some of them. In this thesis, we will use one of the closest to the syntax of Martin Löf Type Theory:

Categories with Families. A Category with Families (CwF) is a category which models contexts equipped with more structure to model types, terms and context morphisms. They were introduced by Dybjer in [Dyb95], see also [Hof97] for a good presentation.
Here, we introduce Categories with Families in a very pedestrian way, and let the reader refer to the presentations of Dybjer and Hofmann for a more abstract point of view. The presentation we choose is the one of Altenkirch and Kaposi [AK16b]. In this work, they build an initial CwF, we present it in Section 1.5.2. We heavily rely on Kaposi's formalization [Kap] and the following code is directly borrowed from there. Kaposi initially conducted its formalization in Agda but we replayed some parts in Coq to formalize the Setoid model in a version of Coq with definitional proof irrelevance. So we start in Agda and will switch to Coq in Section 1.6.

A Category with Families is given by four type families:

- `Con` the type of well-formed contexts
- `Ty Γ` the type of well formed types in the context `Γ`
- `Tms Γ Δ` the type of context morphisms between `Γ` and `Δ`
- `Tm Γ A` the type of terms of type `A`

equipped with some operations. The structure of category is then given by `Con` (the objects) and `Tms` (the morphisms).
By *context morphisms*, between `Γ` and `Δ`, we mean a tuple of terms $(t_0, \dots, t_n)$ such that if `Δ` is $x_0 : A_0, \dots, x_n : A_n$ then:

$$\Gamma \vdash t_0 : A_0$$
$$\Gamma \vdash t_1 : A_1\{x_0 := t_0\}$$
$$\dots$$

Categories with Families interpret the conversion by the propositional equality of the metatheory (which here is $\equiv$, the propositional equality of Agda).

Below are the type families and operations that define a CwF, all gathered in a record[3]:

```
record CwF {l} : Set l where
  field
    Con : Set l
    Ty  : Con → Set l
    Tms : Con → Con → Set l
    Tm  : (Γ : Con) → Ty Γ→ Set l

    •     : Con
    _,_   : (Γ : Con) → Ty Γ→ Con

    _[_]T : ∀{Γ Δ} → Ty Δ→ Tms ΓΔ→ Ty Γ

    id    : ∀{Γ} → Tms ΓΓ
```

[3] Maybe "brutish" should be more accurate than "pedestrian" ...

```
_∘_   : ∀{Γ ΔΣ} → Tms ΔΣ → Tms ΓΔ→ Tms ΓΣ
ε     : ∀{Γ} → Tms Γ•
_,s_  : ∀{Γ Δ}(σ : Tms ΓΔ){A : Ty Δ}
        → Tm Γ(A [ σ]T) → Tms Γ(Δ , A)
π1    : ∀{Γ Δ}{A : Ty Δ} → Tms Γ(Δ , A) → Tms ΓΔ

_[_]t : ∀{Γ Δ}{A : Ty Δ} → Tm ΔA → (σ : Tms ΓΔ)
        → Tm Γ(A [ σ]T)
π2    : ∀{Γ Δ}{A : Ty Δ}(σ : Tms Γ(Δ , A))
        → Tm Γ(A [ π1 σ ]T)

[id]T : ∀{Γ}{A : Ty Γ} → A [ id ]T ≡A
[][]T : ∀{Γ ΔΣ}{A : Ty Σ}{σ : Tms ΓΔ}{δ : Tms ΔΣ}
        → (A [ δ]T [ σ]T) ≡(A [ δ∘ σ ]T)

idl   : ∀{Γ Δ}{δ : Tms ΓΔ} → (id ∘δ) ≡δ
idr   : ∀{Γ Δ}{δ : Tms ΓΔ} → (δ ∘id) ≡δ
ass   : ∀{Δ ΓΣΩ}{σ : Tms ΣΩ}{δ : Tms ΓΣ}
        {ν : Tms ΔΓ}
        → ((σ ∘δ) ∘ν) ≡(σ ∘(δ ∘ν))
,∘    : ∀{Γ ΔΣ}{δ : Tms ΓΔ}{σ : Tms ΣΓ}{A : Ty Δ}
         {a : Tm Γ(A [ δ]T)}
        → (δ ,s a) ∘σ
            ≡ (δ ∘ σ) ,s coe (TmΓ= [][]T) (a [ σ]t)
π1β   : ∀{Γ Δ}{A}{δ : Tms ΓΔ}{a : Tm Γ(A [ δ]T)}
        → (π1 (δ ,s a)) ≡δ
πη    : ∀{Γ Δ}{A : Ty Δ}{δ : Tms Γ(Δ , A)}
        → (π1 δ ,s π2 δ) ≡δ
εη    : ∀{Γ}{σ : Tms Γ•}
        → σ≡ ε

π2β   : ∀{Γ Δ}{A}{δ : Tms ΓΔ}{a : Tm Γ(A [ δ]T)}
      → (π2 (δ ,s a))
            ≡[ TmΓ= (ap (λ z → A [ z ]T) π1β) ]≡ a
```

We thus have three levels of objects:

- the four type families (`Con` to `Tm`)
- the constructors (• to π_2) enabling to inhabit the type families
- and the equalities (`[id]T` to $\pi_2\beta$) which are supposed to hold on the terms of the type families.

For the moment, we can only interpret substitution and conversion but no type or term former. This part is called the *substitution calculus*. To interpret richer theories, the CwF needs more structure, which can be assumed in a modular way, in the same spirit that when we considered extensions of $\lambda\Pi_\omega$.

A CwF is said to *support Π types* if it comes with the following additional structure:

```
Π : ∀{Γ}(A : Ty Γ)(B : Ty (Γ , A)) → Ty Γ

Π[] : ∀{Γ Δ}{σ : Tms ΓΔ}{A : Ty Δ}{B : Ty (Δ , A)}
    → ((Π A B) [ σ]T) ≡(Π (A [ σ]T) (B [ σ^ A ]T))

lam : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)} → Tm (Γ , A) B
      → Tm Γ(Π A B)

app : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)} → Tm Γ(Π A B)
      → Tm (Γ , A) B

lam[] : ∀{Γ Δ}{δ : Tms ΓΔ}{A : Ty Δ}{B : Ty (Δ , A)}
        {t : Tm (Δ , A) B}
        → ((lam t) [ δ]t)
            ≡[ TmΓ= Π[] ]≡ (lam (t [ δ^ A ]t))
Πβ  : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)}{t : Tm (Γ , A) B}
      → (app (lam t)) ≡t
Πη  : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)}{t : Tm Γ(Π A B)}
      → (lam (app t)) ≡t
```

A CwF is said to *support a universe* (à la Tarski) if it comes with the following additional structure:

```
U    : ∀{Γ} → Ty Γ
U[]  : ∀{Γ Δ}{σ : Tms ΓΔ} → (U [ σ]T) ≡U
El   : ∀{Γ}(a : Tm ΓU) → Ty Γ
El[] : ∀{Γ Δ}{σ : Tms ΓΔ}{a : Tm ΔU}
    → (El a [ σ]T) ≡(El (coe (TmΓ= U[]) (a [ σ]t)))
```

A CwF is said to *support Σ types* if it comes with the following additional structure:

```
Σ' : ∀{Γ}(A : Ty Γ)(B : Ty (Γ , A)) → Ty Γ

Σ[] : ∀{Γ Δ}{σ : Tms ΓΔ}{A : Ty Δ}{B : Ty (Δ , A)}
    → (Σ' A B) [ σ]T ≡Σ' (A [ σ]T) (B [ σ^ A ]T)

_,,_ : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)}(a : Tm ΓA)
       → Tm Γ(B [ < a > ]T) → Tm Γ(Σ' A B)
π1  : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)}(w : Tm Γ(Σ' A B))
      → Tm ΓA
π2  : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)}(w : Tm Γ(Σ' A B))
      → Tm Γ(B [ < π1w > ]T)
Σβ1 : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)}{a : Tm ΓA}
```

```
            {b : Tm Γ(B [ < a > ]T)}
          → π1 (a ,, b) ≡a
  Σβ2 : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)}{a : Tm ΓA}
        {b : Tm Γ(B [ < a > ]T)}
      → π2 (a ,, b)
           ≡[ TmΓ= (ap (λ z → B [ < z > ]T) Σβ1) ]≡ b
  ,Σ [] : ∀{Γ}{A : Ty Γ}{B : Ty (Γ , A)}{a : Tm ΓA}
          {b : Tm Γ(B [ < a > ]T)}{θ}{σ : Tms θΓ}
        → (a ,, b) [ σ]t ≡[ TmΓ= Σ[] ]≡
            (a [ σ]t) ,, coe (TmΓ= (...)) (b [ σ]t))
```

More extensions can be found in Kaposi's formalization.
Of course, a CwF supporting Π types and a universe is not enough to interpret $\lambda\Pi_\omega$, it requires more universes and more codes in the universes (at least that they are closed under Π types). We will not try to make a complete list of what is required to interpret $\lambda\Pi_\omega$ in this work, we will only use CwFs as a first step towards a full model of $\lambda\Pi_\omega$.

Remark. In [Dyb95], Dybjer makes the distinction between *external* Categories with Families, which are CwFs in a set theoretic setting, and *internal* one, which are CwFs in a type theoretic setting. To formalize the later, he uses setoids, taking inspiration from a formalization of category theory by Huet and Saïbi. Here we don't bother with setoids because we use a strict equality (which is a kind of set-theoretic flavor in a type-theoretic setting).

The metatheory we live in In this whole thesis we use type theory as our metatheory. We won't be very precise on this type theory but we need at least a strict equality to interpret the previous constructions. We will also suppose that we have inductive types, records, ... which means that we are working in Coq or Agda. In Section 1.5.2 we will use a Quotient Inductive-Inductive Type (QIIT) to define an initial Category with Families. In Section 1.6 we will use a definitional impredicative universe $\mathbb{P}$ to define the Setoid model. In Chapter 4 we will use some Quotient Inductive Types to define the presheaf interpretation of the fibrant replacement in Cubical Type Theory.

1.5.1. Example: the standard model

Let's describe the simplest example of model: the standard model. In this model:

- a context is interpreted by a type using Σ types to "pack" the types of the context
- a type is interpreted by a type family over the context
- and a term by a section of this type family.

```
StandardModel : CwF
StandardModel = record
  { Con  = Set1
  ; Ty   = λ Γ → Γ → Set1
  ; Tms  = λ Γ Δ → Γ → Δ
  ; Tm   = λ Γ A → (γ : Γ) → (A γ)
```

```
; •      = Lift 𝟙
; _,_    = λ Γ A → Σ Γ A
; _[_]T  = λ A δ γ → A (δ γ)
; id     = λ γ → γ
; _∘_    = λ δ σ γ → δ (σ γ)
; ε      = λ _ → tt
; _,s_   = λ δ t γ → δ γ ,Σ t γ
; π₁     = λ δ γ → proj₁ (δ γ)
; _[_]t  = λ t δ γ → t (δ γ)
; π₂     = λ δ γ → proj₂ (δ γ)
; [id]T  = refl
; [][]T  = refl
; idl    = refl
; idr    = refl
; ass    = refl
; ,∘     = refl
; π₁β    = refl
; πη     = refl
; εη     = refl
; π₂β    = refl

; U      = λ _ → Set
; U[]    = refl
; El     = λ a γ → Lift (a γ)
; El[]   = refl

; Π      = λ A B γ → (x : A γ) → B (γ ,Σ x)
; Π[]    = refl
; lam    = λ t γ → λ a → t (γ ,Σ a)
; app    = λ t γ → t (proj₁ γ) (proj₂ γ)
; lam[]  = refl
; Πβ     = refl
; Πη     = refl
}
```

`Lift` is an explicit coercion from `Set` to $\texttt{Set}_1$.
All equalities are provided by `refl`, meaning that conversion is in fact interpreted by conversion of the metatheory. This requires the metatheory to have some η-rules:

- η-rules for Π and Σ types (to interpret πη for instance)
- η-rule for 𝟙 to interpret εη. Unfortunately such a rule is available in Agda but not in Coq.

Those η-rules can also be replaced by FunExt but we then lose the fact that all equalities come from the conversion of the metatheory.

Although simple, the standard model is remarkable for two reasons:

- It justifies MLTT with n universes in MLTT with $n+1$ universes.
  Here we interpreted one universe and we needed to use `Set`$_1$ the second universe of our metatheory to interpret contexts and types. We could generalize and add more universes as long as there is one more in the metatheory to type contexts and types. This model gives thus a trivial[4] proof that MLTT with n universe is consistent, in a metatheory with $n+1$ universes.
- It is a model of Extensional Type Theory.
  Indeed, the reflection rule can be interpreted as long as the metatheory has FunExt. It thus provides a trivial[5] model for ETT. This fact was noticed to me by Kaposi.

The formalization of the standard model can be found in Kaposi's formalization [Kap][6]. We also provide a formalization in Coq in the file `CwF-Coq/StandardModel.v`, unfortunately the η-rule on unit is lacking at some point.

1.5.2. The initial model

Altenkirch and Kaposi have defined a type theory which amounts to be an initial Category with Families. Their construction uses a Quotient Inductive-Inductive type (QIIT).
QIITs take the best of Quotient Inductive Types and Inductive-Inductive Types. Inductive-Inductive Types are mutually defined type families in which the second type family can depend on the first one (and so on). A typical example is the type of well-formed contexts and well-formed types:

$$
\begin{aligned}
&\textbf{Inductive } \mathbf{Ctx} : \mathcal{U}_0 :=\\
&\mid \text{empty} : \mathbf{Ctx}\\
&\mid \text{cons} : \Pi\, \Gamma : \mathbf{Ctx}.\ \mathbf{Ty}\ \Gamma \to \mathbf{Ctx}\\
&\textbf{with Inductive } \mathbf{Ty} : \mathbf{Ctx} \to \mathcal{U}_0 :=\\
&\mid \text{nat} : \Pi\, \Gamma : \mathbf{Ctx}.\ \mathbf{Ty}\ \Gamma\\
&\mid \text{Pi} : \Pi\, (\Gamma : \mathbf{Ctx})\, (A : \mathbf{Ty}\ \Gamma)\, (B : \mathbf{Ty}\, (\text{cons}\ \Gamma\ A)).\ \mathbf{Ty}\ \Gamma.
\end{aligned}
$$

See [NF13] for a treatment of Inductive-Inductive Types.
In the type theory of Altenkirch and Kaposi all terms are well-typed: there is no notion of preterm and only well-typed terms can be written down. Additionally, terms are identified up to βη-equivalence, hence there is no notion of conversion.
To achieve this, the syntax is dependently typed and heavily mutually recursive. The idea is to reproduce the definition of CwF with operations as base constructors and equalities as higher constructors. Hence the following types are defined mutually:

- `Con` the type of well-formed contexts
- `Ty` Γ the type of well formed types

[4] Not so trivial in fact … See coming Section 1.5.3 on the interpretation map.
[5] Idem.
[6] https://bitbucket.org/akaposi/tt-in-tt/src/378848c2886fa007579b1a1479bbe1b601440e47/StandardModel/?at=master

- `Tms Γ Δ` the type of context morphisms
- `Tm Γ A` the type of well-typed terms

Here is the construction for the substitution calculus. QIITs are not implemented in Agda so the following code is some "idealized Agda".

```
data Con : Set
data Ty : Con → Set
data Tms : Con → Con → Set
data Tm : (Γ : Con) → Ty Γ→ Set

data Con where
•     : Con
_,_   : (Γ : Con) → Ty Γ→ Con

data Ty where
_[_]T : ∀{Γ Δ} → Ty Δ→ Tms ΓΔ→ Ty Γ

data Tms where
id    : ∀{Γ} → Tms ΓΓ
_∘_   : ∀{Γ ΔΣ} → Tms ΔΣ→ Tms ΓΔ→ Tms ΓΣ
ε     : ∀{Γ} → Tms Γ•
_,s_  : ∀{Γ Δ}(σ : Tms ΓΔ){A : Ty Δ} → Tm Γ(A [ σ]T)
        → Tms Γ(Δ , A)
π1    : ∀{Γ Δ}{A : Ty Δ} → Tms Γ(Δ , A) → Tms ΓΔ

data Tm where
_[_]t : ∀{Γ Δ}{A}(t : Tm ΔA)(σ : Tms ΓΔ)→  Tm Γ(A [ σ]T)
π2    : ∀{Γ Δ}{A}(σ : Tms Γ(Δ , A)) → Tm Γ(A [ π1σ]T)

data Ty where
[id]T : ∀{Γ}{A : Ty Γ} → A [ id ]T ≡A
[][]T : ∀{Γ ΔΣ}{A : Ty Σ}{σ : Tms ΓΔ}{δ : Tms ΔΣ}
        → (A [ δ]T [ σ]T) ≡(A [ δ∘ σ ]T)

data Tms where
idl   : ∀{Γ Δ}{δ : Tms ΓΔ} → (id ∘δ) ≡δ
idr   : ∀{Γ Δ}{δ : Tms ΓΔ} → (δ ∘id) ≡δ
ass   : ∀{Δ ΓΣΩ}{σ : Tms ΣΩ}{δ : Tms ΓΣ}{ν : Tms ΔΓ}
      → ((σ ∘δ) ∘ν) ≡(σ ∘(δ ∘ ν))
,∘    : ∀{Γ ΔΣ}{δ : Tms ΓΔ}{σ : Tms ΣΓ}{A : Ty Δ}
        {a : Tm Γ(A [ δ]T)}
      → (δ ,s a ∘σ)
             ≡(δ ∘ σ) ,s coe (TmΓ= []T) (a [ σ]t)
π1β   : ∀{Γ Δ}{A : Ty Δ}{δ : Tms ΓΔ}{a : Tm Γ(A [ δ]T)}
      → (π1 (δ ,s a)) ≡δ
```

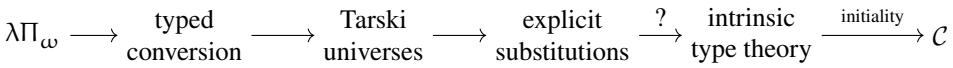


Figure 1.2.: From $\lambda\Pi_\omega$ to a CwF $\mathcal{C}$. The arrow ? is difficult to give.

```
πη    : ∀{Γ Δ}{A : Ty Δ}{δ : Tms Γ(Δ , A)}
      → (π₁ δ,s π₂ δ) ≡δ
εη    : ∀{Γ}{σ : Tms Γ•}
      → σ≡ ε

data Tm where
π₂β   : ∀{Γ Δ}{A : Ty Δ}{δ : Tms ΓΔ}{a : Tm Γ(A [ δ]T)}
      → π₂ (δ ,s a) ≡[ TmΓ= (ap (λ z → A [ z ]T) π₁β) ]≡ a
```

The object that is defined is a type theory: there is a notion of term, of context, etc. but it also organizes as a CwF. The eliminator of the QIIT gives that there is a unique morphism of CwFs into any other CwF, and hence, it is an initial CwF.
There is no notion of reduction for this type theory as convertible terms are identified up to equality. However a metatheory for this type theory still has to be developed. For instance, a notion of normal form (or value) can be defined as a mutual inductive type together with neutral terms. Altenkirch and Kaposi showed that every term has a normal form by normalization by evaluation [AK16a].

1.5.3. On the interpretation map

Well, Categories with Families are models of type theory. Hence, of course, type theory can be interpreted in a CwF. But ... that's not so easy! It is surprising but we don't know how to realize this interpretation map in a type theoretic setting.

Let's try to interpret $\lambda\Pi_\omega$ in a Category with Families. By initiality, we can restrict to the initial CwF. We suppose that we consider CwFs which supports enough universes and which have universes closed under Π types
First, we have to translate $\lambda\Pi_\omega$ to a type theory closer to CwF. Here are the main changes required:

- Move from untyped conversion to a typed conversion:

  $$M \simeq_\beta N \quad \text{vs} \quad \Gamma \vdash M \simeq_\beta N : A$$

  Adams [Ada06], Herbelin and Siles [SH12] have shown that such a translation is possible for all PTS.

- Move from Russell universes to Tarski universes and, if considered, replace the cumulativity by explicit coercions. Assaf developed such a translation in [Ass14].

- Add explicit substitutions.

This suggests a hypothetical compilation chain summed-up in Figure 1.2. But the difficult part is what is remaining. Kaposi calls *extrinsic type theory* the type theory at which we arrived, that is, a type theory with preterms and a typing judgment, and *intrinsic type theory* the initial CwF, that is, an almost identical type theory, but where all terms of the syntax are intrinsically well-typed.
It remains to translate a term with its typing derivation to an intrinsically typed term, we call this the *extrinsic to intrinsic translation*. Of course, this translation is morally the identity. Nevertheless, it is difficult because we have to show that the produced term does not depend on its typing derivation at the same time. At the time of writing, we were not able to complete such a definition.
Streicher gave such an interpretation map in a set theoretic setting [Str91, Def. 3.2], see also [Hof97]. But he uses highly set theoretic features: its interpretation maps are partial and, if we try to give them an intuitive type, even their types are partially defined! This is difficult to reproduce in type theory.

1.6. The setoid model

In this section we take the opportunity of the setting of CwFs to give a formal account of the setoid model of Altenkirch [Alt99], a model for a strict equality enjoying FunExt and UIP in a metatheory without equality. Almost 20 years later, this model remains important as it is a model of the following intuitive idea:

"every type should come with its equality"

As such, it is at the basis of Observational Type Theory [AMS07]. Last, this model can be viewed as a truncated version of the cubical model [CCHM16] in which a type, not only comes with its equality, but also with the higher equalities.

In this model, a context is interpreted by a setoid, that is to say, by a type equipped with an equivalence relation (that we take valued in $\mathbb{P}$). Hence, we will define a CwF whose first component is the type of setoids.

```
SetoidModel : CwF
SetoidModel = record
  { Con  = Setoid
  ; Ty   = SetoidFam
  ; Tms  = SetoidMap
  ; Tm   = FamSection
  ...
```

The setoid model has a strong requirement on the metatheory, it needs a universe of propositions $\mathbb{P}$ for which proof irrelevance holds definitionally:

$$\frac{\Gamma \vdash P : \mathbb{P} \qquad \Gamma \vdash p, q : P}{\Gamma \vdash p \simeq_\beta q : P}$$

Let's remark that such a type theory requires the conversion algorithm to be typed. To experiment with such a metatheory, we use a modified version of Coq by Gilbert [Gil] instead of Agda. This version implement a universe of propositions with definitional proof irrelevance which is written SProp. We also switch to Coq in the presentation here. Kaposi have conducted a similar formalization[7] in Agda but using h-sets instead of proof irrelevance.
We don't give the definitions of all the fields of the CwF but only the most interesting ones. The others can be found in the formalization. The formalization that we present here can be found in the file CwF-Coq/SetoidModel.v. It uses the modified version of Coq by Gilbert[8].

Contexts are interpreted by setoids. A setoid is a type equipped with an equivalence relation valued in SProp:

```
Record Setoid :=
  { carrier : Type;
    eq : carrier → carrier → SProp;
    refl : ∀ x, eq x x;
    sym : ∀ {x y}, eq x y → eq y x;
    trans : ∀ {x y z}, eq x y → eq y z → eq x z }.

Coercion carrier : Setoid ↣ Sort class.
Notation " γ ∼ γ' " := (eq _ γ γ') (at level 20).
```

The coercion makes that, when Γ is a setoid, we can write Γ instead of `carrier Γ` and Coq automatically inserts the coercion.
Thanks to definitional proof irrelevance, the groupoidal laws for the relation hold for free. For instance, we have $p \cdot p^{-1} = 1$ by reflexivity:

```
Check (λ Γ γ γ' p ⇒ eq_refl
 : ∀ (Γ : Setoid) (γ γ' : Γ) (p : γ ∼ γ'), trans p (sym p) = refl γ).
```

Context morphisms are interpreted by morphisms of setoids which are maps between the carriers respecting the relations:

```
Record SetoidMap (Γ Δ : Setoid) :=
  { map : Γ → Δ;
    map_eq : ∀ {γ γ'}, γ ∼ γ' → map γ ∼ map γ' }.

Coercion map : SetoidMap ↣ Funclass.
```

Types are interpreted by families of setoids which are "coherently dependent setoids":

```
Record SetoidFam (Γ : Setoid) :=
  { Fam : Γ → Setoid;
    subst : ∀ {γ γ'}, γ ∼ γ' → SetoidMap (Fam γ) (Fam γ');
    refl* : ∀ {γ} x, subst (refl γ) x ∼ x;
    trans* : ∀ {x y z} (p : x ∼ y) (q : y ∼ z) u,
        subst q (subst p u) ∼ subst (trans p q) u }.
```

[7] https://bitbucket.org/akaposi/tt-in-tt/src/e904d3b257070bc6508a8ae347b1430660bd16ec/Setoid/?at=master
[8] https://github.com/SkySkimmer/coq/tree/sprop

Each type family comes with `subst`, a transport along the equality in its base type. It is how the transport of propositional equality is interpreted in this model: "by induction on P". `refl*` and `trans*` assert that transport preserves reflexivity and transitivity. Thanks to proof irrelevance, they are enough to derive that transport respects symmetry:

```
Definition sym* {Γ} {A: SetoidFam Γ} {γ γ': Γ} (p : γ ~ γ') (x : A γ)
  : subst (sym p) (subst p x) ~ x
  := trans (trans* p (sym p)) (refl* x).
```

Terms are interpreted by sections of setoid families:

```
Record FamSection Γ (A : SetoidFam Γ) :=
  { tm : ∀ γ, Fam A γ;
    resp : ∀ γ γ' (p : γ ~ γ'), subst p (tm γ) ~ tm γ' }.
```

Context extension is interpreted by Σ types together with the Σ of the relations:

```
Definition cons (Γ : Con) (A : Ty Γ) : Con
  := {| carrier := {γ : Γ & A γ};
        eq := λ x y ⇒ { p : x.1 ~ y.1 s& ι p x.2 ~ y.2} ;
        refl := ... ;
        sym := ... ;
        trans := ... |}.
```

where `{ x : A s& B x }` is the Σ type with `A` and `B` in `SProp`.

Functions are interpreted by functions respecting the relations:

$$\Sigma\, f : A \to B.\ \Pi\, x, x' : A.\ x \sim x' \to f\ x \sim f\ x'$$

And two functions f, g of this type are in relation $f \sim g$ if we have:

$$\Pi\, x, x' : A.\ x \sim x' \to \pi_1(f)\ x \sim \pi_1(g)\ x'$$

More formally, the interpretation of Π types is given by:

```
Definition Π {Γ} (A : Ty Γ) (B : Ty (Γ, A)) : Ty Γ
  := {| Fam := λ γ ⇒
        {| carrier := {f : ∀ (x : A γ), B (γ; x) s|
                           ∀ x y (p : x ~ y), ι ... (f x) ~ (f y)};
           eq := λ f g⇒ ∀ x y (p: x ~ y), ι ... (val f x) ~ (val g y);
           refl  := ... ;
           sym   := ... ;
           trans := ... |};
        subst  := ... ;
        refl*  := ... ;
        subst* := ... |}.
```

where `{x : A & B x}` is the usual Σ type (in `Type`) and `{x : A s| B x}` is the Σ type with `A` in `Type` and `B` in `SProp` (subset type).

Thanks to definitional proof irrelevance, all identities of the CwF hold definitionally (as for the standard model). This is quite remarkable.

Let's now have a look at how propositional equality is interpreted:

```
Definition Eq {Γ} (A: Ty Γ) (t u : Tm _ A) : Ty Γ
  := {| Fam := λ γ ⇒
              {| carrier := sBox (t γ ∼ u γ) ;
                 eq := λ _ _ ⇒ sUnit ;
                 refl := λ _ ⇒ stt ;
                 sym := λ _ _ _ ⇒ stt;
                 trans := λ _ _ _ _ _ ⇒ stt |};
              subst  := … ;
              refl*  := λ _ _ ⇒ stt ;
              subst* := λ _ _ _ _ _ _ ⇒ stt |}.
```

`sBox` is the embedding of `SProp` into `Prop`, `sUnit` is the unit type in `SProp` and `stt` is its unique element.
The equality on `A` is interpreted by the relation coming with `A`. The equality of equalities is interpreted by unit and thus validates UIP. Then reflexivity comes from the reflexivity of the relation $\sim$ and transport along a family `P` comes from the `subst` component of `P`. This model validates FunExt by the definition of equality on function types.
A weakness of this model is that transport only weakly computes, that is, `transport refl u` is only propositional equal to `u` and not definitionally. In contrast, as we will see in a moment, some definitional equalities hold for FunExt.

We let the reader refer to the formalization to have all the details of the model. In the following we will highlight a few of its particularities.

1.6.1. SetoidFam and SetoidFam'

The definition of a setoid family `SetoidFam` is the original one of Altenkirch (or, at most, a slight rephrasing). There is another notion of family we could have used, using heterogeneous relations. We show here that the two notions are equivalent. This alternative definition was introduced by Kaposi in his formalization[9].
This time, a setoid family is a "heterogeneous setoid", still with coercion functions:

```
Record SetoidFam' (Γ : Setoid) :=
  { Fam'   : Γ → Type ;
    eq'    : ∀ {γ γ'}, γ ∼ γ' → Fam' γ → Fam' γ' → SProp ;
    coe'   : ∀ {γ γ'}, γ ∼ γ' → Fam' γ → Fam' γ' ;
    coh'   : ∀ {γ γ'} (p : γ ∼ γ') x, eq' p x (coe' p x) ;
    Frefl' : ∀ γ x, eq' (refl γ) x x ;
    Fsym'  : ∀ γ γ' (p : γ ∼ γ') x y, eq' p x y → eq' (sym p) y x ;
    Ftrans': ∀ γ γ' γ'' (p : γ ∼ γ') (q : γ' ∼ γ'') x y z,
```

[9] https://bitbucket.org/akaposi/tt-in-tt/src/e904d3b257070bc6508a8ae347b1430660bd16ec/Setoid/Decl.agda?at=master

```
      eq' p x y → eq' q y z → eq' (trans p q) x z ;
}.
```

We formalized the following:

Proposition 2. For every setoid Γ, `SetoidFam Γ` and `SetoidFam' Γ` are equivalent.

The proof is easy, we let the interested reader refer to the formalization. The proofs of section and retraction require FunExt and propositional extensionality for `SProp`.

1.6.2. FunExt computes

The setoid model is a model of function extensionality, we exhibit here a nice behavior of FunExt in this model: some computation rules hold definitionally.
Let's assume that we have f and g two functions which are pointwise equal:

$$\begin{aligned} f, g &: \prod x : A.\ B\ x \\ e &: \prod x : A.\ f\ x = g\ x \end{aligned}$$

Then, **FunExt**(e) is a proof of $f = g$.
Now, let's assume that x_0 is a point of A, P a predicate over $B\ x_0$ and u a proof of $P\ (f\ x_0)$:

$$\begin{aligned} x_0 &: A \\ P &: B\ x_0 \to \mathcal{U} \\ u &: P\ (f\ x_0) \end{aligned}$$

Then there are two ways to get a proof of $P\ (g\ x_0)$: either by transporting along e or along **FunExt**(e):

$$\begin{aligned} v_1,\ v_2 &: P\ (g\ x_0) \\ v_1 &\coloneqq e\ x_0\ \#^P\ u \\ v_2 &\coloneqq \mathbf{FunExt}(e)\ \#^{\lambda f.\ P\ (f\ x_0)}\ u \end{aligned}$$

Proposition 3. In the setoid model, v_1 and v_2 are convertible.

Proof. The proof is by reflexivity:

```
Check (eq_refl : v1 = v2).
```

See the formalization for the construction of **FunExt**(e), `v1` and `v2`. □

1.6.3. Universe closed under Π types

The last construction in the setoid model that we want to discuss is the universe. In [Alt99], Altenkirch introduced a universe closed only under non-dependent function types. We show here how it is possible to build a universe closed under Π types and lower universes.

Internal universes We first recall what is an internal universe. Internal universes are Inductive-Recursive types which define universes à la Tarski. They are called "internal" because the construction can be carried in the type theory in which we are working (Agda for example) as long as it features induction-recursion. Inductive-Recursive types and their use to define internal universes were introduced by Dybjer in [Dyb00].
Here is a simple example of internal universe closed only under $\mathbb{N}$ and Π types:

$$
\begin{aligned}
&\textbf{Inductive } \mathbf{U} : \mathcal{U}_0 \coloneqq \\
&\mid \text{nat} \ : \mathbf{U} \\
&\mid \text{pi} \ : \Pi\,(a : \mathbf{U})\,(b : \mathbf{El}\,a \to \mathbf{U}).\ \mathbf{U} \\
&\textbf{with } \mathbf{El} \ : \mathbf{U} \to \mathcal{U}_0 \coloneqq \\
&\mathbf{El}\,\text{nat} = \mathbb{N} \\
&\mathbf{El}\,(\text{pi}\ a\ b) = \Pi\, x : \mathbf{El}\,a.\ \mathbf{El}\,(b\ x)
\end{aligned}
$$

Of course it is possible to enrich **U** with more codes.

Internal universe for the setoid model We come back to the setoid model. The idea is to define an internal universe (`U`, `El`) by induction recursion but we also provide enough structure so that `U` is a setoid and so on.
Equality between two codes of `U` is interpreted by syntactic equality between the codes, hence two codes are equal when they are "the same", not when they denote two equivalent setoids.

We give here the interpretation for a first universe $\mathcal{U}_0$, closed under natural numbers and Π types. Afterward, we will sketch the interpretation of a second universe $\mathcal{U}_1$ to show how to handle the hierarchy. For $\mathcal{U}_0$ we get a model for the following rules:

$$
\vdash \mathcal{U}_0 \qquad a : \mathcal{U}_0 \vdash \mathbf{El}\,a \qquad \vdash n : \mathcal{U}_0 \qquad \frac{\vdash a : \mathcal{U}_0 \qquad x : \mathbf{El}\,a \vdash b : \mathcal{U}_0}{\vdash \pi\, x : \mathbf{El}\,a.\, b : \mathcal{U}_0}
$$

$$
\mathbf{El}\,n \simeq_\beta \mathbb{N} \qquad\qquad \mathbf{El}\,(\pi\, x : a.\, b) \simeq_\beta \Pi\, x : \mathbf{El}\,a.\ \mathbf{El}\,b
$$

Let's remark that such a construction does not provide a *complete* universe, meaning that we can't interpret this kind of rules:

$$
\frac{\vdash A \qquad A \text{ small}}{\vdash \lceil A \rceil : \mathcal{U}_0} \qquad\qquad \mathbf{El}(\lceil A \rceil) \simeq_\beta A
$$

Such rules would require to interpret the universe by the type of setoids but this type is not itself a setoid: it is a groupoid. Hence, we don't have a complete universe but still complete enough to interpret all types "coming from the syntax".

The following is the construction of $\mathtt{U}_0$. The construction heavily uses induction-recursion which is not available in Coq so we switch back to Agda. To emulate the definitional proof irrelevance, we use irrelevant arguments [AS12]. Irrelevant arguments are marked with a dot

. in the signature of functions. They are not taken in account when convertibility of applied functions is checked but the result of the function cannot depend on them. See the Agda Manual[10] for more details. The formalization presented in this section is available in the folder SetoidUniverse-Agda.
Let's start by the signatures of our objects. First, we need a type $\mathtt{U}_0$ which is a setoid:

```
data U₀  : Set
_≡ᵤ_     : U₀ → U₀ → Set
≡ᵤrefl   : (a : U₀) → a ≡ᵤa
≡ᵤsym    : {a a' : U₀} → .(a ≡ᵤa') → a' ≡ᵤa
≡ᵤtrans  : {a b c : U₀} → .(a ≡ᵤb) → .(b ≡ᵤc) → a ≡ᵤc
```

Then, we need $\mathtt{El}_0$ to be a family of setoid indexed over $\mathtt{U}_0$:

```
El₀      : U₀ → Set
_~ _     : {a : U₀} → El₀ a → El₀ a → Set
~ refl   : {a : U₀} → (x : El₀ a) → x ~ x
~ sym    : {a : U₀} → {x y : El₀ a} → .(x ~ y) → y ~ x
~ trans  : ∀ {a} {x y z : El₀ a} → .(x ~ y) → .(y ~ z) → x ~ z
```

We need coercion functions (the `subst` field of `SetoidFam`):

```
subst  : {a a' : U₀} → .(p : a ≡ᵤa') → El₀ a → El₀ a'
subst* : ∀ {a a'} → (p : a ≡ᵤa') → {x y : El₀ a} → x ~ y
           → subst p x ~ subst p y
```

And the fact that `subst` preserves reflexivity and transitivity:

```
refl*  : {a : U₀} → (x : El₀ a) → subst (≡ᵤrefl a) x ~ x
trans* : ∀ {a₀ a₁ a₂} (p : a₀ ≡ᵤa₁) (q : a₁ ≡ᵤa₂) (x: El₀ a₀)
           → subst q (subst p x) ~ subst (≡ᵤtrans p q) x
```

As we have seen, `sym*` is always derivable, we will use it in definitions:

```
sym* : {a₀ a₁ : U₀} → (p : a₀ ≡ᵤa₁) → (x : El₀ a₀)
           → subst (≡ᵤsym p) (subst p x) ~ x
sym* p x = ~ trans (trans* p (≡ᵤsym p) x) (refl* x)
```

We now define those 14 objects. This is a lot, but all definitions mutually depend on each other. We are in fact redefining all the setoid model. Let's start with $\mathtt{U}_0$:

```
data U₀ where
  n : U₀
  π : ∀ (a : U₀) (b : El₀ a → U₀)
        (b₁ : ∀ {x y : El₀ a} (p : x ~ y) → b x ≡ᵤb y) → U₀
```

$\mathtt{U}_0$ has a code for $\mathbb{N}$ and one for Π types. The interesting thing is the component $\mathtt{b}_1$ which is a witness that `b` is a setoid map between $\mathtt{El}_0$ `a` and $\mathtt{U}_0$.
Now we can define $\mathtt{El}_0$. The interpretation of Π types is functions preserving the equalities.

[10] http://agda.readthedocs.io/en/latest/language/irrelevance.html

```
El₀ n = ℕ
El₀ (π a b b₁) =
   Σ ((x : El₀ a) → El₀ (b x))
     (λ f → ∀{x y} (p : x ∼ y) → subst (b₁ p) (f x) ∼ f y)
```

Note that the Σ used here is in fact a subset type.
The equality on $\mathtt{U}_0$ is the syntactic equality, defined by pattern-matching:

```
n ≡ᵤ n = ⊤
π a b b₁ ≡ᵤ π a' b' b₁' = Σ (a ≡ᵤa') (λ p → ∀x → b x ≡ᵤb' (subst p x))
_ ≡ᵤ _ = ⊥
```

The equality thus enjoys reflexivity, symmetry and transitivity:

```
≡ᵤrefl n = tt
≡ᵤrefl (π a b b₁) = ≡ᵤrefl a , λx → b₁ (∼ sym (refl* x))
≡ᵤsym = (...)
≡ᵤtrans = (...)
```

The setoid structure on types of the form $\mathtt{El}_0\ \mathtt{a}$ is given by case analysis on a and is guided by the equalities $\mathbf{El}\ n \simeq_\beta \mathbb{N}$ and $\mathbf{El}\ (\pi\, x : a.\, b) \simeq_\beta \prod x : \mathbf{El}\ a.\ \mathbf{El}\ b$:

```
_∼ _ {n} m₁ m₂ = if eq-nat-dec m₁ m₂ then ⊤ else ⊥
_∼ _ {π a b b₁} (f , f₁) (g , g₁) =
      ∀(x y : El₀ a)(p : x ∼ y) → subst (b₁ p) (f x) ∼ g y

∼ refl {n} = eq-nat-refl
∼ refl {π a b b₁} (f , f₁) x y p = f₁ p
∼ sym = (...)
∼ trans = (...)
```

Remain `subst` and the proofs that it respects ∼ , `refl` and `trans`. In every case, this is defined by induction on codes and little choice is left for the definitions. We let the interesting reader dive into the formalization to have all the details.

Let's sketch the interpretation of the next universe. The construction is essentially the same but there are more cases:

```
data U₁ where
  ↑₀  : U₀ → U₁
  u₀  : U₁
  π₀₁ : (a : U₀) → (b : El₀ a → U₁)
      → (b₁ : ∀{x y : El₀ a}(p : x ∼ y) → b x ≡ᵤ1b y) → U₁
  π₁₀ : (a : U₁) → (b : El₁ a → U₀)
      → (b₁ : ∀{x y : El₁ a}(p : x ∼ ₁y) → b x ≡ᵤb y) → U₁
  π₁₁ : (a : U₁) → (b : El₁ a → U₁)
      → (b₁ : ∀{x y : El₁ a}(p : x ∼ ₁y) → b x ≡ᵤ1b y) → U₁
```

$\uparrow_0$ c is the embedding of $\mathtt{U}_0$ in $\mathtt{U}_1$, this coercion give explicit cumulativity à la Assaf [Ass14]. $\mathtt{u}_0$ is the code of $\mathtt{U}_0$ (to interpret $\mathcal{U}_0 : \mathcal{U}_1$). π_{01}, π_{10} and π_{11} are codes for the different

dependent products, this interprets a type theory with annotated products (see Section 2.3 for an example of such annotations).

2. Program translations as syntactic models of $\lambda\Pi_\omega$

In this chapter we introduce *program translations*, which are a method to give a model of Martin Löf Type Theory without making use of Categories with Families. In a first section, we give a general setting for program translations and then we give several examples:

- three "times bool" translations
- the ad-hoc polymorphism translation
- several variations of unary parametricity translation.

In a last section, we compare program translations and more semantical models such as CwFs. We will see that most of good properties of program translations extend to a more general class of *syntactic models* whose description is still not very precise.
This part was largely published in the article *The next 700 syntactical models of type theory* (Boulier, Pédrot, Tabareau) [BPT17] published at CPP 2017. There are some additions: the treatment of several axioms, the fact that weak univalence does not imply full univalence, the presentation of parametricity translations, etc. In general, this chapter owes much to numerous discussions with Pédrot and Tabareau.

2.1. General setting

A program translation is a translation of terms of a source theory $\mathcal{S}$ to terms of a target theory $\mathcal{T}$. For us, $\mathcal{S}$ and $\mathcal{T}$ will always be variations of $\lambda\Pi_\omega$ with more or less expressivity.
There are two different views of a program translation:

- A program translation is a *syntactic model*, that is, it is the interpretation of a type theory $\mathcal{S}$ into a mathematical object $\mathcal{T}$, which turns out to be another type theory. This interpretation can be used to justify some new constructs.
- A program translation is a *compilation phase* from a dependently typed language to another. This compilation phase can be used to implement some constructs of $\mathcal{S}$ in terms of constructs of $\mathcal{T}$.

The program translations that we will present all follow the same scheme. Given a source type theory $\mathcal{S}$ and a target type theory $\mathcal{T}$:

- a term M of $\mathcal{S}$ is translated into a term $[M]$ of $\mathcal{T}$
- a type A of $\mathcal{S}$ is translated into a type $[\![A]\!]$ of $\mathcal{T}$

– and a context Γ of $\mathcal{S}$ is translated into $[\![\Gamma]\!]$ of $\mathcal{T}$

where the translation of types is defined in function of the translation of terms:

$$[\![A]\!] \coloneqq \mathscr{E}\,[A] \quad \text{with } \mathscr{E} \text{ of type } [\![\mathcal{U}]\!] \to \mathcal{U}$$

and where contexts are translated pointwise:

$$\begin{array}{rcl} [\![\cdot]\!] & \coloneqq & \cdot \\ [\![\Gamma, x : A]\!] & \coloneqq & [\![\Gamma]\!], x : [\![A]\!] \end{array}$$

Program translations generalize the PTS morphisms (see for instance [Las12]) which are program translations only changing sorts and not the other term formers.

Remark. The $\mathscr{E}$ map corresponds to the interpretation of the **El** type former when universes are considered à la Tarski:

$$\frac{\Gamma \vdash a : \mathcal{U}_i}{\Gamma \vdash \mathbf{El}\, a}$$

Indeed, $[\![A]\!]$ is the translation when A is considered as a type, and $[A]$ when it is considered as a term (*i.e.* a code of $\mathcal{U}_i$).

Let $\bot_\mathcal{S}$ and $\bot_\mathcal{T}$ represent the absurd in $\mathcal{S}$ and $\mathcal{T}$ (for instance the empty type $\mathbf{0}$, the false proposition $\bot$, its impredicative encoding $\Pi\, X : \mathcal{U}.\, X, \ldots$). For $\lambda\Pi_\omega$, we will take $\Pi\, X : \mathcal{U}_i.\, X$. We will say that a type theory is consistent if its absurd type is not inhabited in the empty context.

The expected properties of a translation are then the following:

typing soundness If $\Gamma \vdash M : A$ then $[\![\Gamma]\!] \vdash [M] : [\![A]\!]$.

consistency preservation If $[\![\bot_\mathcal{S}]\!]$ is inhabited in the empty context then so is $\bot_\mathcal{T}$.

In general, consistency preservation is given by a map of type $[\![\bot_\mathcal{S}]\!] \to \bot_\mathcal{T}$ in $\mathcal{T}$. Under those two conditions the consistency of $\mathcal{T}$ implies the one of $\mathcal{S}$.

Proposition 4. Let $[\_]$ be a translation from $\mathcal{S}$ to $\mathcal{T}$ satisfying typing soundness and consistency preservation. Then, if $\mathcal{T}$ is consistent, so is $\mathcal{S}$.

Proof. The consistency of a type theory is defined by the non-existence of a closed term of type $\bot$. We thus have to show that, given a closed term t of type $\bot_\mathcal{S}$ in $\mathcal{S}$, we can construct a term t' of type $\bot_\mathcal{T}$ in $\mathcal{T}$. By typing soundness, $[t]$ is of type $[\![\bot_\mathcal{S}]\!]$. We thus get t' by applying consistency preservation to $[t]$. □

For a given translation, the proof of typing soundness always relies on the following lemmas:

substitution lemma For all M and N, $[M\{x \coloneqq N\}] \simeq_\beta [M]\{x \coloneqq [N]\}$.

computational soundness If $M \simeq_\beta N$ then $[M] \simeq_\beta [N]$.

The former being needed to prove the later, and the later being needed to handle the case of the conversion rule in the proof of typing soundness.

In some translations, the reduction is preserved (if $M \to N$ then $[M] \to^+ [N]$) which is stronger than computational consistency. This gives the strong normalization for $\mathcal{S}$ if $\mathcal{T}$ enjoys it.

	FunExt	**Univalence**	**weakUnivalence**	**UIP**
times bool on Π types	negates	negates	preserves	preserves
times bool on $\mathcal{U}$	preserves	negates	negates	preserves
cheap parametricity	preserves	preserves	?	preserves
generous parametricity	preserves	negates	negates	preserves

Figure 2.1.: Axioms preservation

Translations and axioms The source theory $\mathcal{S}$ can have more logical principles than $\mathcal{T}$ as long as they have a translation in $\mathcal{T}$. By *logical principle* we mean a set of syntactic constructs together with reduction and typing rules. Among them, we will distinguish:

- an *operator*, which is a constant of closed type with some reduction rules
- an *axiom*, which is a constant of closed type without reduction rules

In practice, $\mathcal{S}$ is often of the form $\mathcal{T} + axiom$ and the translation ensures that the axiom can be assumed without breaking the coherence of the system. In such a case, the trivial translation from $\mathcal{T}$ to $\mathcal{T} + axiom$ (the injection) gives the equiconsistency between the two theories.
Let $[\![\_]\!]$ be a translation from $\mathcal{S}$ to $\mathcal{T}$ and **ax** an axiom of closed typed **Ax** which can be stated both in $\mathcal{S}$ and $\mathcal{T}$. We will say that:

- The translation *proves* or *gives a computational content to* **ax** if $[\![\mathbf{Ax}]\!]$ is inhabited.
  In this case the translation extends to $\mathcal{S} + \mathbf{ax} \to \mathcal{T}$
- The translation *negates* **ax** if $[\![\neg\mathbf{Ax}]\!]$ is inhabited.
  In this case the translation extends to $\mathcal{S} + \neg\mathbf{ax} \to \mathcal{T}$
- The translation *preserves* **ax** if $\mathbf{Ax} \to [\![\mathbf{Ax}]\!]$ is inhabited (in $\mathcal{T}$).
  In this case the translation extends to $\mathcal{S} + \mathbf{ax} \to \mathcal{T} + \mathbf{ax}$

Remark. Correct program translations respect *logical implication*:

$$\text{If } A \to B \text{ then } [\![A]\!] \to [\![B]\!]$$

The term of type $[\![A]\!] \to [\![B]\!]$ is given by $\lambda\, x : [\![A]\!] .\, [t\ x]$.
As a consequence, if **Ax** and **Ax′** are two axioms such that **Ax** implies **Ax′**:

- If $[\![\_]\!]$ proves **Ax**, then it also proves **Ax′**.
- If $[\![\_]\!]$ negates **Ax′**, then it also negates **Ax**.

We recollect in Figure 2.1 some of the preservation results that we will show in this section.

2.2. Times bool

"Times bool" ... by this mysterious name we refer to a technique to negate various extensionality principles. It consists in adding a dummy boolean on the type former for which you want to negate extensionality.

- Add it to Π types and you will negate function extensionality
- Add it to **stream** types and you will negate extensionality of streams
- Add it to all types and you will negate univalence
- Add it to propositions and you will negate propositional extensionality ...

We detail the three first examples in this section. They will be our first examples of translations.

2.2.1. Times bool on Π

The first translation that we consider is a translation from $\lambda\Pi_\omega$ to $\lambda\Pi_\omega + \Sigma + \mathbb{B}$ which negates function extensionality (FunExt). Let's have a look at the translation:

$$\begin{aligned}
[\Pi\, x : A.\ B]_f &\coloneqq (\Pi\, x : [\![A]\!]_f.\ [\![B]\!]_f) \times \mathbb{B} \\
[\lambda\, x : A.\ M]_f &\coloneqq (\lambda\, x : [\![A]\!]_f.\ [M]_f, \mathbf{true}) \\
[M\ N]_f &\coloneqq \pi_1([M]_f)\ [N]_f
\end{aligned}$$

The translation is defined by induction on the syntax of terms. The subscript f is only here to indicate that we are referring to this particular translation. To negate FunExt, a function is translated by a pair of a function and a boolean. The application throws away the boolean (because of the π_1). But as the application is the only way to observe a function, the boolean can never be observed. And thus two identical functions wearing different booleans will be observationally the same although not identical as functions in the target theory.
To complete the definition of the translation, we have to define the translation on variables, sorts and types:

$$\begin{aligned}
[x]_f &\coloneqq x \\
[\mathcal{U}_i]_f &\coloneqq \mathcal{U}_i \\
[\![A]\!]_f &\coloneqq [A]_f
\end{aligned}$$

In this translation, particularly simple, terms and types have the same translation ($[\![A]\!]_f$ is equal to $[A]_f$), and the double brackets are only here to make a "moral" distinction between terms and types. This is only possible because $[\mathcal{U}_i]_f = \mathcal{U}_i$ (and thus $\mathscr{E}$ is the identity).
We first check that the translation enjoys the substitution lemma:

Proposition 5 (formalized)**.** For all terms M, N and variable x we have:

$$[M\{x \coloneqq N\}]_f \ = \ [M]_f\{x \coloneqq [N]_f\}$$

With which we can prove reduction preservation and then typing soundness:

Proposition 6 (formalized)**.** If $M \to_\beta N$ then $[M]_f \to_\beta^+ [N]_f$.

Theorem 7 (formalized)**.** If $\Gamma \vdash M : A$ then $[\![\Gamma]\!]_f \vdash [M]_f : [\![A]\!]_f$.

The proofs of those properties are not difficult: substitution lemma is by induction on M, computational soundness is by induction on the derivation of the conversion judgment and typing soundness is by induction on the typing derivation. However the number of cases is relatively important, that is why we formalized them in Coq. The formalization can be found in the `TimesBoolCorrectness-Coq` folder. We do not present the formalization in this thesis, but such a presentation is available in the CPP article [BPT17].

Last, as $[\![\Pi\, X : \mathcal{U}_i.\ X]\!]_f = (\Pi\, X : \mathcal{U}_i.\ X) \times \mathbb{B}$, we get:

Theorem 8. The translation $[\_]_f$ from $\lambda\Pi_\omega$ to $\lambda\Pi_\omega + \Sigma + \mathbb{B}$ preserves consistency.

This translation is quite simple but yet, it implies strictly more than $\lambda\Pi_\omega$: thanks to the boolean we can discriminate between two functions which are extensionally equal. Let's remark that the boolean worn by a function is always **true** when the function comes from the source theory, but can be arbitrary when the function comes from a quantification over a Π type.
To negate FunExt, we first extends the translation to propositional equality:

$$\begin{aligned}
[M =_A N]_f &\coloneqq [M]_f =_{[\![A]\!]_f} [N]_f \\
[\mathbf{refl}_= M]_f &\coloneqq \mathbf{refl}_= [M]_f \\
[J_=(y.q.P,\ e,\ u)]_f &\coloneqq J_=(y.q.[\![P]\!]_f,\ [e]_f,\ [u]_f)
\end{aligned}$$

This extensions preserves typing soundness, making $[\_]_f$ a translation from $\lambda\Pi_\omega + =$ to $\lambda\Pi_\omega + \Sigma + \mathbb{B} + =$.

Theorem 9. Let's recall that we defined **FunExt** as:

$$\mathbf{FunExt}_{i,j} \coloneqq \Pi\,(A : \mathcal{U}_i)\,(B : A \to \mathcal{U}_j)\,(f,\ g : \Pi\, x : A.\ B\,x).\ (\Pi\, x.\ f\ x = g\ x) \to f = g$$

Then for all levels i, j, there is a closed term of type:

$$[\![\mathbf{FunExt}_{i,j} \to \bot]\!]_f$$

Proof. Let's instantiate A and B by a type X (assumed to be both in $\mathcal{U}_i$ and $\mathcal{U}_j$), and f and g by $(\lambda\, x : X.\ x, \mathbf{true})$ and $(\lambda\, x : X.\ x, \mathbf{false})$. We do have $[\![\Pi\, x : A.\ f\ x = g\ x]\!]_f$ because the application does not look at the boolean. From which we get $(\lambda\, x : X.\ x, \mathbf{true}) = (\lambda\, x : X.\ x, \mathbf{false})$ and thus $\mathbf{true} = \mathbf{false}$. Hence a contradiction. □

Remark. It is quite hard to unfold the whole definition of $[\![\mathbf{FunExt}_{i,j} \to \bot]\!]_f$ on paper. But when this type is automatically computed and injected in a proof assistant, it becomes very easy to inhabit it. In the CPP article, we developed a specific plugin for Coq to realize this operation. In next chapter, we will present a more general plugin working for several translations. In particular, we implemented the translation $[\_]_f$, see Section 3.6. The formalization of the previous proof can thus be found in the file `TemplateCoq/translations/times_bool_fun.v`. We warmly encourage the interested reader to replay the formalization to better understand the previous proof.

This model shows thus that: if $\lambda\Pi_\omega + \Sigma + \mathbb{B} + =$ is consistent, then FunExt is not derivable in $\lambda\Pi_\omega + =$. As FunExt holds in set models of type theory, we conclude that it is independent from $\lambda\Pi_\omega + =$.

Remark. It is possible to define a variant of this translation where the boolean is added inside the Π type instead of outside:

$$[\Pi\, x : A.\, B]_{f'} \quad := \quad \Pi\, x : [\![A]\!]_{f'}.\, ([\![B]\!]_{f'} \times \mathbb{B})$$

The two translations seem to share most of their properties.

Compatibility with extensions of $\lambda\Pi_\omega$

Let's review how this translation interacts with the several extensions of $\lambda\Pi_\omega$.

Inductive types The translation extends easily to inductive types. Indeed most of inductive types are translated by themselves, adding some "$(\_, \mathbf{true})$" where needed. For instance, here is the translation for natural numbers:

$$\begin{aligned} [\mathbb{N}]_f &:= \mathbb{N} \\ [0]_f &:= 0 \\ [S]_f &:= (S, \mathbf{true}) \end{aligned}$$

We implemented the translation of a wide class of inductives in the plugin presented in Section 3.6.

Eta rules The translation negates the η-rule for Π types (even propositionally). The proof goes as for FunExt: take f to be $(\lambda\, x : \mathbb{1}.\, x, \mathbf{false})$, then $\lambda\, x : \mathbb{1}.\, f\ x$ is translated to $(\lambda\, x : \mathbb{1}.\, x, \mathbf{true})$.

Cumulativity The translation trivially preserves cumulativity: it extends to a translation $\lambda\Pi_\omega + \subseteq \longrightarrow\ \lambda\Pi_\omega + \Sigma + \mathbb{B} + \subseteq$.

UIP The translation does not touch equality types, we can thus see that it preserves UIP.

Prop The translation does not validate impredicativity: $(\Pi\, A.\ P) \times \mathbb{B}$ is never of type $\mathbb{P}$. Hence, it does not extend to $\lambda\Pi_\omega + \mathbb{P}$. Let's remark that $\mathbb{P}$ is expected to enjoys proof irrelevance which is incompatible with the negation of FunExt on Π types living in $\mathbb{P}$.

Univalence and Weak Univalence

We conclude this section by a nice result: the translation shows that weak univalence ($A \simeq B \to A = B$) does not imply full univalence, and even not function extensionality.
Let's recall that univalence is defined by:

$$\mathbf{Univalence}_i \ := \ \Pi\, A,\ B : \mathcal{U}_i.\ \mathbf{IsEquiv}\ (\mathbf{paths\_to\_equiv}_{A,B})$$

asserting that a particular map $A = B \to A \simeq B$ is an equivalence.

We define *weak univalence* as the weaker assumption that there only exists a map from $A \simeq B$ to $A = B$:

$$\mathbf{weakUnivalence}_i \coloneqq \Pi\, A,\, B : \mathcal{U}_i.\, A \simeq B \to A = B$$

Of course, univalence implies weak univalence but the converse does not holds. Shulman asked for a model distinguishing them on the HoTT mailing list[1] and the only answer was the one was of Oliveri:

"Why wouldn't a skeletal LCCC be a model of [weak univalence] + UIP?"

This is a beautiful illustration of a case where a syntactic model can give a much simpler proof of independence than a categorical model: "times bool" does the job!
First, we have that the translation preserves weak univalence.

Proposition 10. The translation $[\![\_]\!]_f$ extends to:

$$\lambda\Pi_\omega + = + \mathbf{weakUnivalence} \;\longrightarrow\; \lambda\Pi_\omega + \Sigma + \mathbb{B} + = + \mathbf{weakUnivalence}$$

Proof. The idea is that an element of type $[\![A \simeq B]\!]_f$ is enough to build an equivalence between A and B (we even throw away all the booleans). And we have $A = B \to [\![A = B]\!]_f$.
See the formalization in the file `TemplateCoq/translations/times_bool_fun.v` for the details. □

On the contrary the translation negates full univalence.

Proposition 11. The translation $[\![\_]\!]_f$ extends to:

$$\lambda\Pi_\omega + = + \neg\, \mathbf{Univalence} \;\longrightarrow\; \lambda\Pi_\omega + \Sigma + \mathbb{B} + = + \mathbf{Univalence}$$

Proof. Let's first remark that we can't conclude that the translation negates univalence from the fact that univalence implies FunExt. Indeed, the proof that we know uses η rules which are invalidated by the translation. To circumvent this issue, we could inhabit the translated type $[\![\mathbf{Univalence} \to \mathbf{FunExt}]\!]$.
Here we chose to directly inhabit $[\![\neg\, \mathbf{Univalence}]\!]$. We use **Univalence** in the target but this assumption can probably be weakened.
We consider the equivalence given by univalence on unit:

$$[\![\mathbb{1} \simeq \mathbb{1}]\!]_f \rightleftarrows [\![\mathbb{1} = \mathbb{1}]\!]_f$$

By univalence in the target the type $[\![\mathbb{1} = \mathbb{1}]\!]_f$ is contractible. But $[\![\mathbb{1} \simeq \mathbb{1}]\!]_f$ is not: we can have several elements by tweaking the booleans.
Again, see the formalization for the details. □

Hence the translation $[\![\_]\!]_f$ gives a model of $\mathbf{weakUnivalence} \wedge \neg\, \mathbf{Univalence}$ as long as $\lambda\Pi_\omega + \Sigma + \mathbb{B} + \mathbf{Univalence}$ is consistent, which holds by, *e.g.* the simplicial model.

[1] [HoTT] Weaker forms of univalence, July 20, 2017

2.2.2. Times bool on streams

Let's now focus on streams, the coinductive type introduced in Section 1.2.4. We have seen that there are two notions of equality for streams: the one induced by identity types and the one given by bisimilarity. Stream extensionality (StreamExt) is the fact that these notions of equality coincide:

$$\mathbf{StreamExt}_i \coloneqq \Pi\,(A : \mathcal{U}_i)\,(s_1\ s_2 : \mathbf{stream}\ A).\ \mathbf{bisim}\ A\ s_1\ s_2 \to s_1 =_{\mathbf{stream}\ A} s_2$$

We use here the "times bool technique" to show that stream extensionality is not derivable in Martin Löf Type Theory. The technique is the same that for FunExt: a stream is translated by a pair of a stream and a boolean (always **true** when coming from the source). The boolean is thrown away when accessing the head and threaded when accessing the tail. Bisimilarity only observe streams through **hd**, and thus it cannot discriminate between two identical streams wearing two different booleans.
The translation $[\_]_s$ from $\lambda\Pi_\omega + = + \mathbf{stream}$ to $\lambda\Pi_\omega + = + \mathbf{stream} + \Sigma + \mathbb{B}$ is defined by:

$$\begin{aligned}
[\mathcal{U}_i]_s &\coloneqq \mathcal{U}_i \\
[x]_s &\coloneqq x \\
[\mathbf{stream}\ A]_s &\coloneqq (\mathbf{stream}\ [\![A]\!]_s) \times \mathbb{B} \\
[\mathbf{hd}\ M]_s &\coloneqq \mathbf{hd}\ (\pi_1([M]_s)) \\
[\mathbf{tl}\ M]_s &\coloneqq (\mathbf{tl}\ (\pi_1([M]_s)),\ \pi_2([M]_s)) \\
[\mathbf{stream_corec}\ S\ M_0\ M_1\ N]_s &\coloneqq (\mathbf{stream_corec}\ [S]_s\ [M_0]_s\ [M_1]_s\ [N]_s,\ \mathbf{true}) \\
[\mathbf{bisim}\ A\ M\ N]_s &\coloneqq \mathbf{bisim}\ [\![A]\!]_s\ [M]_s\ [N]_s \\
[\mathbf{hd_b}\ M]_s &\coloneqq \mathbf{hd_b}\ [M]_s \\
[\mathbf{tl_b}\ M]_s &\coloneqq \mathbf{tl_b}\ [M]_s \\
[\mathbf{bisim_corec}\ S\ M_0\ M_1\ P_0\ P_1\ N]_s &\coloneqq \mathbf{bisim_corec}\ \big(\lambda s_0\ s_1.\,[S]_s\ (s_0, (\pi_2\ [P_0]_s))\ (s_1, (\pi_2\ [P_1]_s))\big) \\
&\qquad \big(\lambda s_0\ s_1\ s.\,[M_0]_s\ (s_0, (\pi_2\ [P_0]_s))\ (s_1, (\pi_2\ [P_1]_s))\ s\big) \\
&\qquad \big(\lambda s_0\ s_1\ s.\,[M_1]_s\ (s_0, (\pi_2\ [P_0]_s))\ (s_1, (\pi_2\ [P_1]_s))\ s\big) \\
&\qquad \big(\pi_1\ [P_0]_s\big)\ \big(\pi_1\ [P_1]_s\big)\ [N]_s \\
[\dots]_s &\coloneqq \dots \\
[\![A]\!]_s &\coloneqq [A]_s
\end{aligned}$$

In unspecified cases, the translation commutes with the corresponding syntax constructor. As the previous translation, $[\_]_s$ verifies the substitution lemma, computational soundness and finally typing soundness:

Theorem 12 (formalized)**.** If $\Gamma \vdash M : A$ then $[\![\Gamma]\!]_s \vdash [M]_s : [\![A]\!]_s$.

Remark. This theorem is correct only when *surjective pairing* (η-rule for Σ types) holds. It is needed to show that typing rule of **bisim_corec** is indeed preserved.

Even more directly than for dependent functions, absurd is translated into itself because the translation does not act on dependent products, so that we can derive consistency preservation directly.

Theorem 13. In $\lambda\Pi_\omega + = + \mathbf{stream} + \Sigma + \mathbb{B}$, there is a closed term of type:

$$\vdash [\![\mathbf{StreamExt} \to \bot]\!]_s$$

As a consequence, if $\lambda\Pi_\omega + = + \mathbf{stream} + \Sigma + \mathbb{B}$ is consistent, StreamExt is not derivable in $\lambda\Pi_\omega + = + \mathbf{stream}$.

Proof. Once again, equality observes the boolean in the translated streams but bisimilarity does not. Given a type X and $x : X$ we can define the constant stream $\mathsf{id}_x := (x, x, \dots)$. We then instantiate **StreamExt** with $(\mathsf{id}_x, \mathbf{true})$ and $(\mathsf{id}_x, \mathbf{false})$. It follows $\mathbf{true} = \mathbf{false}$. □

Π types and identity types are not touched by the translation, hence the translation preserves univalence. It thus extends to:

$$\lambda\Pi_\omega + = + \mathbf{stream} + \neg\,\mathbf{StreamExt} + \mathbf{Univalence} \longrightarrow \lambda\Pi_\omega + = + \mathbf{stream} + \Sigma + \mathbb{B} + \mathbf{Univalence}$$

As a consequence we have:

Proposition 14. Univalence does not imply stream extensionality.

This result is not contradictory with the works of Ahrens *et. al.* [ACS15]. In this paper they prove stream extensionality in a univalent setting. However, they are working with defined streams (from natural numbers) and not "primitive" ones. For instance, their streams validate the β-rule only propositionally. In a blog post[2] on `homotopytypetheory.org` Licata suggests a proof of stream extensionality, this one is also only valid for "defined streams". On the contrary, it seems that stream extensionality for primitive streams is derivable in cubical type theory, see the implementation in Cubical-Agda[3].

2.2.3. Times bool on types

Let's use a last time the times bool trick. This time, we add a boolean to all types, this negates both propositional extensionality and univalence (the former being univalence for propositions).
The translation $[\_]_t$ from $\lambda\Pi_\omega + \subseteq + = + \mathbb{P}$ to $\lambda\Pi_\omega + \subseteq + = + \mathbb{P} + \Sigma + \mathbb{B}$ is defined by:

$$\begin{aligned}
[s]_t &:= (s \times \mathbb{B}, \mathbf{true}) \\
[x]_t &:= x \\
[\Pi\, x : A.\, B]_t &:= (\Pi\, x : [\![A]\!]_t.\, [\![B]\!]_t, \mathbf{true}) \\
[M =_A N]_t &:= ([M]_t =_{[\![A]\!]_t} [N]_t, \mathbf{true}) \\
[\dots]_t &:= \dots \\
[\![A]\!]_t &:= \pi_1([A]_t)
\end{aligned}$$

[2] https://homotopytypetheory.org/2014/02/17/another-proof-that-univalence-implies-function-extensionality/
[3] https://github.com/Saizan/cubical-demo/blob/master/examples/Cubical/Examples/Stream.agda

The sort s is either $\mathcal{U}_i$ or $\mathbb{P}$. In unspecified cases, the translation commutes with the corresponding syntax constructor.
Every type is translated by the pair of this type and a boolean (always **true** when coming from the source). When seen as a term, a type wears a boolean, but when seen as a type, the boolean is lost.
For the first time, the universe is not translated by itself. The typing rule $\mathcal{U}_i : \mathcal{U}_{i+1}$ is preserved because:

$$[\mathcal{U}_i]_f \;=\; (\mathcal{U}_i \times \mathbb{B},\, \mathbf{true}) \quad : \quad [\![\mathcal{U}_{i+1}]\!]_f \;=\; \pi_1(\mathcal{U}_{i+1} \times \mathbb{B},\, \mathbf{true}) \simeq_\beta \mathcal{U}_{i+1} \times \mathbb{B}$$

As for previous times bool translations, we have:

Theorem 15 (formalized)**.** The translation $[\_]_t$ verifies the substitution lemma, reduction preservation, typing soundness and consistency preservation.

Consistency preservation comes from the fact that $[\![\Pi\, X : \mathcal{U}_i.\, X]\!]_t \;=\; \Pi\, X : \mathcal{U}_i \times \mathbb{B}.\, \pi_1(X)$ which is logically equivalent to $\Pi\, X : \mathcal{U}_i.\, X$.

Remark. The typing soundness proof uses the following lemmas:

- $\Gamma \vdash t : A \;\Rightarrow\; \exists\, s,\, \Gamma \vdash A : s$ (type correctness)
- $\Gamma \vdash \Pi\, x : A.\, B : s'' \;\Rightarrow\; \exists\, s\, s',\, \Gamma \vdash A : s\,,\;\; \Gamma,\, x : A \vdash B : s'$ and $\mathcal{R}(s, s', s'')$

Although the translation is simple, the proof of typing soundness is very long when carried in details. That's where the formalization becomes interesting, it forces to check all well-formedness conditions of contexts and types which are always said to be "obviously true".

Theorem 16. Let's recall that propositional extensionality is given by:

$$\mathbf{PropExt} \;\coloneqq\; \Pi\, A\, B : \mathbb{P}.\, (A \to B) \to (B \to A) \to A =_{\mathbb{P}} B$$

Then, there is a closed term of type:

$$\vdash [\![\mathbf{PropExt} \to \bot]\!]_s$$

Consequently, **PropExt** is not derivable in $\lambda\Pi_\omega + \subseteq + = + \mathbb{P} + \Sigma + \mathbb{B}$ as long as the type theory $\lambda\Pi_\omega + \subseteq + = + \mathbb{P}$ is consistent.

Proof. As previously, instantiating A and B by $(X, \mathbf{true})$ and $(X, \mathbf{false})$ for any proposition $X : \mathbb{P}$. □

Remark. It is often said that univalence implies propositional extensionality. Without cumulativity, this is only true if univalence is taken directly on the universe $\mathbb{P}$. In general, we can tweak the translation to show that univalence on a sort s does not imply univalence on other sorts. To do this, annotate the products (as in Section 2.3) and add booleans only on other sorts than s. This gives a translation which preserves univalence on s and negates univalence on other sorts. Of course, when cumulativity holds the trick does not work anymore.

2.3. Ad-hoc polymorphism

The main advantage of times bool translations lies in their simplicity, that's why they are a good example to introduce program translations. Unfortunately they negate useful principles (FunExt, StreamExt, ...) instead of inhabiting them. We present here a translation which inhabits something useful: ad-hoc polymorphism. By ad-hoc polymorphism, we refer to an operator, **univ_rec**, allowing to pattern-match on the normal form of a type. With this operator, we can define a function whose behavior depends on the type of its arguments. For instance (with a pattern matching notation):

$$\begin{aligned}\lambda\, A : \mathcal{U}_0.\ &\textbf{match}\ A\ \textbf{with}\\ &\mid \mathbb{B} \Rightarrow \lambda\, b.\ \mathrm{not}\ b\\ &\mid \_ \Rightarrow \lambda\, x.\ x\end{aligned}$$

This function has the type of polymorphic identity $\Pi\, A : \mathcal{U}_0.\ A \to A$, but it discriminates on A to be the negation on bool and the identity otherwise. We yet see that this extension is not compatible with parametricity. The first apparition of such an operator comes back to [NPS90, Section 14.2].
The idea of the translation is to interpret each type by its code in an internal universe **TYPE**, and to translate pattern matching on types to pattern matching on codes. We make here the hypothesis of a "closed world" meaning that no new inductive or coinductive type can be added to the universe on the fly. Contrarily to previous translations, the added principle, **univ_rec**, is not only an axiom but a true operator in the sense that it has some reduction rules.

In this section, the translation depends on the universe level of types. In order to have the translation independent of the typing derivation, we use $\lambda\Pi^a_\omega$, an annotated version of $\lambda\Pi_\omega$. The syntax and the typing rules of $\lambda\Pi^a_\omega$ are defined in Figure 2.2. Annotations are added such that given a typing statement $\Gamma \vdash M : A$ the sort of A is known. When we write $x :_i A$ (in contexts and binders) it means that A is of sort $\mathcal{U}_i$, the j annotation on Π types means that B is of sort j. And last, the judgment $\Gamma \vdash M :_i A$ means that M is of type A and A of sort $\mathcal{U}_i$. We will omit some annotations which are obvious (especially in the $\to$ notation).
In order to be able to write our example of "non identity function" (and to have a non-empty universe), we add booleans to the source theory. We then have to add a universe annotations on $\mathbb{B}$ and on the return type of boolean pattern matching. The typing rules for booleans in $\lambda\Pi^a_\omega$ are given in Appendix B. Of course, the translation extends to richer universes.

Our target theory will be $\lambda\Pi_\omega$ enriched with some internal universes $(\mathbf{TYPE}_i, \mathbf{Elt}_i)_{i:\mathbb{N}}$. Those internal universes are defined by induction-recursion, as in Section 1.6.3. We give here the two first ones $\mathbf{TYPE}_0$ and $\mathbf{TYPE}_1$ in Agda and the formation and introduction rules for general case in Figure 2.3. The typing rules of the eliminator **TYPE_rec** are postponed to Appendix B.
In Agda, $\mathbf{TYPE}_0$ and $\mathbf{TYPE}_1$ are defined as follows. Those definitions can also be found in the attached file `TYPE.agda`.

```
data TYPE₀ : Set
El₀ : TYPE₀ → Set
```

$$A, B, M, N ::= x \mid \mathcal{U}_i \mid M\ N \mid \lambda\, x :_i A.\ M \mid \Pi^j\, x :_i A.\ B$$
$$\Gamma, \Delta ::= \cdot \mid \Gamma, x :_i A$$

$$\frac{}{\vdash \cdot} \qquad \frac{\Gamma \vdash A :_{i+1} \mathcal{U}_i}{\vdash \Gamma, x :_i A} \qquad \frac{\Gamma \vdash \qquad (x :_i A) \in \Gamma}{\Gamma \vdash x :_i A} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{U}_i :_{i+2} \mathcal{U}_{i+1}}$$

$$\frac{\Gamma \vdash A :_{i+1} \mathcal{U}_i \qquad \Gamma, x :_i A \vdash B :_{j+1} \mathcal{U}_j \qquad k = \max(i, j)}{\Gamma \vdash \Pi^j\, x :_i A.\ B :_{k+1} \mathcal{U}_k}$$

$$\frac{\Gamma, x :_i A \vdash M :_j B \qquad k = \max(i, j)}{\Gamma \vdash \lambda\, x :_i A.\ B :_k \Pi^j\, x :_i A.\ B}$$

$$\frac{\Gamma \vdash M :_k \Pi^j\, x :_i A.\ B \qquad \Gamma \vdash N :_i A \qquad k = \max(i, j)}{\Gamma \vdash M\ N :_j B\{x := N\}}$$

$$\frac{\Gamma \vdash M :_i B \qquad \Gamma \vdash A :_{i+1} \mathcal{U}_i \qquad A \simeq_\beta B}{\Gamma \vdash M :_i A} \qquad (\lambda\, x :_i A.\ M)\ N \simeq_\beta M\{x := N\}$$

Figure 2.2.: Typing rules for $\lambda\Pi^a_\omega$ (congruence rules for $\simeq_\beta$ omitted)

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{TYPE}_i : \mathcal{U}_0} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{Elt}_i : \mathbf{TYPE}_i \to \mathcal{U}_0} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{b}_i : \mathbf{TYPE}_i}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{u}_i : \mathbf{TYPE}_{i+1}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \pi_{i,j} : \Pi\, A : \mathbf{TYPE}_i.\ (\mathbf{Elt}_i\ A \to \mathbf{TYPE}_j) \to \mathbf{TYPE}_{\max(i,j)}}$$

Figure 2.3.: Formation and introduction rules for $\mathbf{TYPE}_i$ internal universe

```
data TYPE₀ where
  π₀₀ : (a : TYPE₀) → (b : El₀ a → TYPE₀) → TYPE₀
  b₀ : TYPE₀

El₀ (π₀₀ a b) = (x : El₀ a) → El₀ (b x)
El₀ b₀ = 𝔹


data TYPE₁ : Set
El₁ : TYPE₁ → Set

data TYPE₁ where
  π₀₁ : (a : TYPE₀) → (b : El₀ a → TYPE₁) → TYPE₁
  π₁₀ : (a : TYPE₁) → (b : El₁ a → TYPE₀) → TYPE₁
  π₁₁ : (a : TYPE₁) → (b : El₁ a → TYPE₁) → TYPE₁
  b₁ : TYPE₁
  u₀ : TYPE₁

El₁ (π₀₁ a b) = (x : El₀ a) → El₁ (b x)
El₁ (π₁₀ a b) = (x : El₁ a) → El₀ (b x)
El₁ (π₁₁ a b) = (x : El₁ a) → El₁ (b x)
El₁ b₁ = 𝔹
El₁ u₀ = TYPE₀
```

The translation $[\_]_T$ from $\lambda\Pi^a_\omega + \mathbb{B}$ to $\lambda\Pi_\omega + \mathbb{B} + (\mathrm{TYPE}_i)_{i\in\mathbb{N}}$ is defined by:

$$
\begin{array}{rcl}
[x]_T & \coloneqq & x \\
[\mathcal{U}_i]_T & \coloneqq & \mathbf{u}_i \\
[\Pi^j\, x :_i A.\, B]_T & \coloneqq & \pi_{i,j}\, [A]_T\, (\lambda\, x : [\![A]\!]^i_T.\, [B]_T) \\
[\lambda\, x :_i A.\, M]_T & \coloneqq & \lambda\, x : [\![A]\!]^i_T.\, [M]_T \\
[M\ N]_T & \coloneqq & [M]_T\ [N]_T \\
[\mathbb{B}^i]_T & \coloneqq & \mathbf{b}_i \\
[\mathbf{true}]_T & \coloneqq & \mathbf{true} \\
[\mathbf{false}]_T & \coloneqq & \mathbf{false} \\
[\mathbf{if}\ M\ \mathbf{ret}^j\ P\ \mathbf{then}\ N_1\ \mathbf{else}\ N_2]_T & \coloneqq & \mathbf{if}\ [M]_T\ \mathbf{ret}\ (\mathbf{Elt}_j \circ [P]_T)\ \mathbf{then}\ [N_1]_T\ \mathbf{else}\ [N_2]_T \\
[\![A]\!]^i_T & \coloneqq & \mathbf{Elt}_i\, [A]_T \\
[\![\cdot]\!]_T & \coloneqq & \cdot \\
[\![\Gamma, x :_i A]\!]_T & \coloneqq & [\![\Gamma]\!]_T, x : [\![A]\!]^i_T
\end{array}
$$

Theorem 17. The translation $[\_]_T$ verifies the substitution lemma, computational soundness and typing soundness:

$$\text{If } \Gamma \vdash M :_i A \text{ then } [\![\Gamma]\!]_T \vdash [M]_T : [\![A]\!]^i_T.$$

It is important that $\mathbf{Elt}_{i+1}\ \mathbf{u}_i$ is indeed $\mathbf{TYPE}_i$ and not $\mathcal{U}_i$, otherwise we would not have the fixpoint for universes.

Remark. We did not succeed in defining a unique internal universe **TYPE** indexed over $\mathbb{N}$ and containing all $(\mathbf{TYPE}_i)_{i\in\mathbb{N}}$. However, this does not seem out of reach, it would mean that MLTT with induction-recursion and one universe is more powerful than MLTT with ω universes.

The translation gives access to the family of operators $(\mathbf{univ\_rec}_i^j)_{i,j\in\mathbb{N}}$ where i is the universe level of the type which is pattern-matched and j is the universe level of the return predicate. They are defined in $\lambda\Pi_\omega^a + \mathbb{B}$ by:

$$\frac{\Gamma \vdash P :_{j+1} \mathcal{U}_0 \to \mathcal{U}_j \qquad \Gamma \vdash \mathbf{b}'_0 :_j P\ \mathbb{B}^0 \qquad \Gamma \vdash \pi'_{0,0} : H_{0,0}}{\Gamma \vdash \mathbf{univ\_rec}_0^j(P, \mathbf{b}'_0, \pi'_{0,0}) :_j \Pi\, A : \mathcal{U}_0.\ P\ A}$$

$$\frac{\begin{array}{ccccc} & & \Gamma \vdash P : \mathcal{U}_{i+1} \to \mathcal{U}_j & \Gamma \vdash \mathbf{b}'_{i+1} : P\ \mathbb{B}^{i+1} & \\ \Gamma \vdash \mathbf{u}'_{i+1} : P\ \mathcal{U}_i & \Gamma \vdash \pi'_{0,0} : H_{0,0} & \dots & \Gamma \vdash \pi'_{i+1,i+1} : H_{i+1,i+1} & \end{array}}{\Gamma \vdash \mathbf{univ\_rec}_{i+1}^j(P, \mathbf{b}'_{i+1}, \pi'_{0,0}, \dots, \pi'_{i+1,i+1}, \mathbf{u}'_i) : \Pi\, A : \mathcal{U}_{i+1}.\ P\ A}$$

where H_{i_0,i_1} is:

$\Pi\,(A : \mathcal{U}_i)\,(B : A \to \mathcal{U}_i).\ P\ A \to (\Pi\, x : A.\ P\ (B\ x)) \to P\ (\Pi^i\, x :_i A.\ B\ x)$ when $i_0 = i_1 = i$

$\Pi\,(A : \mathcal{U}_{i_0})\,(B : A \to \mathcal{U}_{i_1}).\ (\Pi\, x : A.\ P\ (B\ x)) \to P\ (\Pi^{i_1}\, x :_{i_0} A.\ B\ x)$ when $i_0 < i_1$

$\Pi\,(A : \mathcal{U}_{i_0})\,(B : A \to \mathcal{U}_{i_1}).\ P\ A \to P\ (\Pi^{i_1}\, x :_{i_0} A.\ B\ x)$ when $i_0 > i_1$

and where in dots of last rule there are all π'_{i_0,i_1} such that $\max(i_0, i_1) = i + 1$.

$$\begin{array}{lll}
\mathbf{univ_rec}_i^j(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ \mathbb{B}^i & \simeq_\beta & \mathbf{b}'_i \\
\mathbf{univ_rec}_{i+1}^j(P, \mathbf{b}'_{i+1}, \pi'_{0,0}, \dots, \mathbf{u}'_i)\ \mathcal{U}_i & \simeq_\beta & \mathbf{u}'_i \\
\mathbf{univ_rec}_i^j(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ (\Pi^i\, x :_i A.\ B) & \simeq_\beta & \pi'_{i,i}\ A\ (\lambda\, x.\ B)\ (\mathbf{univ_rec}_i^j(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ A) \\
 & & \qquad (\lambda\, x.\ \mathbf{univ_rec}_i^j(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ B) \\
\mathbf{univ_rec}_i^j(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ (\Pi^i\, x :_j A.\ B) & \simeq_\beta & \pi'_{j,i}\ A\ (\lambda\, x.\ B)\ (\lambda\, x.\ \mathbf{univ_rec}_i^j(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ B) \\
\mathbf{univ_rec}_i^j(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ (\Pi^j\, x :_i A.\ B) & \simeq_\beta & \pi'_{i,j}\ A\ (\lambda\, x.\ B)\ (\mathbf{univ_rec}_i^j(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ A)
\end{array}$$

where $i > j$ in the two last rules.

The operator $\mathbf{univ\_rec}_i^j$ allows to directly observe the normal form of a type in $\mathcal{U}_i$. The function on $\mathcal{U}_0$ which is the negation on boolean and the identity otherwise is defined by:

$$\begin{array}{l} f : \Pi\, A : \mathcal{U}_0.\ A \to A \\ f = \mathbf{univ\_rec}_0^0(\lambda\, A.\ A \to A,\ \mathrm{neg},\ \mathrm{id}) \end{array}$$

Proposition 18. The translation $[\_]_T$ gives a computational meaning to the family of operators $(\mathbf{univ\_rec}_i^j)_{i,j:\mathbb{N}}$.

Proof. We can extend the translation with:

$$[\mathbf{univ\_rec}_i^j(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)]_T \quad := \quad \mathbf{TYPE\_rec}_i(\mathbf{Elt}_j \circ [P]_T, [\mathbf{b}'_i]_T, [\pi'_{0,0}]_T, \dots)$$

and we check that typing and conversion rules of $\mathbf{univ\_rec}_i^j$ are indeed preserved. □

When the source theory has an empty type $\mathbf{0}$, we can extends the translation by adding a code for $\mathbf{0}$ in $\mathbf{TYPE}_0$, consistency is then preserved because $[\![\mathbf{0}]\!]_T^0 = \mathbf{0}$.
But without an empty type, consistency preservation is less clear. Indeed, $[\![\Pi^i\, A :_{i+1} \mathcal{U}_i.\, A]\!]_T$ is equal to $\Pi\, A : \mathbf{TYPE}_i.\, \mathbf{Elt}_i\, A$ and there is no evident code to feed to this type so that we get an inconsistency. For instance, if we try to give the code of $\Pi\, X : \mathcal{U}.\, X$ we get back the same type.

Proposition 19. The system $\lambda\Pi^a_\omega + \mathbb{B} + \mathbf{0} + (\mathbf{univ\_rec}_i^j)_{i,j}$ is consistent provided that $\lambda\Pi_\omega + \mathbb{B} + \mathbf{0} + (\mathtt{TYPE}_i)_{i\in\mathbb{N}}$ (with codes for $\mathbf{0}$) is so.

We have remarked earlier that ad-hoc polymorphism contradicts parametricity. Let's make this more precise.

Proposition 20. Let **IdParam** be the parametricity statement of $\Pi\, A : \mathcal{U}_0.\, A \to A$:

$$\mathbf{IdParam} := \Pi\,(f : \Pi\, A : \mathcal{U}_0.\, A \to A)\,(A : \mathcal{U}_0)\,(A_t : A \to \mathcal{U}_0)\,(x : A)\,(x_t : A_t\, x).\, A_t\, (f\, x)$$

Then, as long as $\lambda\Pi_\omega + \mathbb{B} + \mathbf{0} + (\mathtt{TYPE}_i)_{i\in\mathbb{N}}$ is consistent, **IdParam** is not derivable in $\lambda\Pi^a_\omega$.

Proof. The function f which is the negation on bool and the identity otherwise allows to construct a closed term of type:

$$\mathbf{IdParam} \to \bot$$

Hence, **IdParam** is not derivable in $\lambda\Pi^a_\omega + \mathbb{B} + \mathbf{0} + (\mathbf{univ\_rec}_i^j)_{i,j}$. □

2.4. Parametricity

Parametricity was introduced by Reynolds [Rey83], and then Wadler [Wad89], for System F as a logical relation. It is a tool to derive *free theorems* for terms of polymorphic types. The Hello World of parametricity is for instance to show that a function f of type

$$\forall A,\ A \to A$$

can, in System F, only be a polymorphic identity function. This is given by the fact that the free theorem for the type $\forall A,\ A \to A$ is

$$\forall (A : \mathcal{U})(A_t : A \to \mathcal{U})(x : A)(x_t : A_t\, x),\ A_t\, (f\, x)$$

and that every closed term f of type $\forall A,\ A \to A$ satisfies it.

Parametricity has then been extended to several systems, and in particular to dependent types and inductive types by Bernardy *et. al.* [BJP10], [BL11]. For System F, a logic has to be introduced to state and prove the free theorems. But for rich systems like $\lambda\Pi_\omega$, the free theorems can be stated and proved in the same system[4]. In those systems parametricity can thus be seen ... as a program translation! And indeed, many recent works on parametricity have been carried in a setting close to ours. With respect to previous translations, the "syntactic model" point of view is less accurate for this translation because it does not add any logical power. It is however an important translation which has many other uses.

unpacked	$[\![\Gamma,\ x : A]\!] = [\![\Gamma]\!],\ x : [A]_0,\ x_t : [A]_1\ x$
packed	$[\![\Gamma,\ x : A]\!] = [\![\Gamma]\!],\ x : \Sigma\, x_0 : [A]_0.\ [A]_1\ x_0$

cheap	$[\Pi\, x : A.\ B]_0 = \Pi\, x : [A]_0.\ [B]_0$
generous	$[\Pi\, x : A.\ B]_0 = \Pi\, (x : [A]_0)\, (x_t : [A]_1\ x).\ [B]_0$

Figure 2.4.: Two variations of the parametricity translation

In this section we review the traditional parametricity translation for $\lambda\Pi_\omega$ and then introduce two variations: the pack / unpack variation, and the cheap / generous one. We focus on *unary* parametricity, also called the *the logical predicate interpretation*. That is, a type is interpreted by a predicate, *i.e.* a unary relation. All results are expected to be generalizable to n-ary parametricity.

2.4.1. Original translation

We present here the original translation introduced for Pure Type Systems in [BJP10], [BL11], [KL12]. As far as we know, all other works relying on the parametricity translation use this presentation (for instance [Las14]) and the two variations that we will introduce after seem little known.

The translation has two components $[\_]_0$ and $[\_]_1$. For a type A, $[A]_1$ is a predicate over $[A]_0$. And for a term M, $[M]_1$ is the proof that $[M]_0$ is parametric, *i.e.* a proof that $[A]_1\ [M]_0$ holds. As usual, the two maps are defined by induction on untyped syntax:

$$
\begin{array}{rcl}
[M]_0 & \coloneqq & M \\
[x]_1 & \coloneqq & x_t \\
[\Pi\, x : A.\ B]_1 & \coloneqq & \lambda\, f : [\Pi\, x : A.\ B]_0.\ \Pi\, (x : [A]_0)\, (x_t : [A]_1\ x).\ [B]_1\ (f\ x) \\
[\lambda\, x : A.\ t]_1 & \coloneqq & \lambda\, (x : [A]_0)\, (x_t : [A]_1\ x).\ [t]_1 \\
[\mathcal{U}_i]_1 & \coloneqq & \lambda\, A : \mathcal{U}_i.\ A \to \mathcal{U}_i \\
[\![\Gamma,\ x : A]\!] & \coloneqq & [\![\Gamma]\!],\ x : [A]_0,\ x_t : [A]_1\ x
\end{array}
$$

[4] See [BJP10] for conditions on a Pure Type System such that it enjoys this property.

In the case of variable, x_t is assumed to always be a fresh name. For Π types, the predicate is the fact that functions are morphisms: if x satisfies the predicate of A then $f\ x$ satisfies the predicate of B. The universe fixpoint is respected because:

$$[\mathcal{U}_i]_1 = \lambda\, A : \mathcal{U}_i.\ A \to \mathcal{U}_i \quad : \quad [\mathcal{U}_{i+1}]_1\ \mathcal{U}_i = \mathcal{U}_i \to \mathcal{U}_{i+1}$$

The first component $[\_]_0$ is the identity in a named setting. In a De-Bruijn setting each index of free variables is doubled because $[t]_0$ is intended to be typed in the context $[\![\Gamma]\!]$. See Section 3.5 for an implementation of this translation.

Theorem 21. The translation verifies the substitution lemma, computational soundness and typing soundness:

$$\text{If } \Gamma \vdash M : A \text{ then } [\![\Gamma]\!] \vdash [M]_0 : [A]_0 \text{ and } [\![\Gamma]\!] \vdash [M]_1 : [A]_1\ [M]_0.$$

2.4.2. Packed translation

Let's introduce a first variation. In the previous translation, there were two translation functions $[\_]_0$ and $[\_]_1$ for terms and one $[\![\_]\!]$ for types. This does not fit very well in the setting of program translations spelled out in Section 2.1 in which there is only one translation function for terms. To recover this setting we can *pack* the two components using Σ types:

$$\begin{array}{rcl} [M] & \coloneqq & ([M]_0,\ [M]_1) \\ [\![A]\!] & \coloneqq & \Sigma\, x : [A]_0.\ [A]_1\ x \end{array}$$

The translation now becomes:

$$\begin{array}{rcl} [M]_0 & \coloneqq & M\left\{x \coloneqq \pi_1(x) \text{ for } x \text{ free in } M\right\} \\ [x]_1 & \coloneqq & \pi_2(x) \\ [\lambda\, x : A.\ M]_1 & \coloneqq & \lambda\, x : [\![A]\!].\ [M]_1 \\ [M\ N]_1 & \coloneqq & [M]_1\ [N] \\ [\Pi\, x : A.\ B]_1 & \coloneqq & \lambda\, f.\ \Pi\, x : [\![A]\!].\ [B]_1\ (f\ \pi_1(x)) \\ [\mathcal{U}_i]_1 & \coloneqq & \lambda\, A : \mathcal{U}_i.\ A \to \mathcal{U}_i \\ [x] & \coloneqq & x \\ [M] & \coloneqq & ([M]_0,\ [M]_1) \text{ in other cases} \\ [\![A]\!] & \coloneqq & \Sigma\, x_0 : [A]_0.\ [A]_1\ x_0 \\ [\![\Gamma,\ x : A]\!] & \coloneqq & [\![\Gamma]\!],\ x : [\![A]\!] \end{array}$$

The two functions $[\_]_0$ and $[\_]_1$ are only auxiliary functions used to define $[\_]$ and $[\![\_]\!]$. The first component of the translation $[\_]_0$ is not the identity anymore because variables come now with their proof of parametricity in the context. We thus have to distinguish between variables bound by the λ and Π and those bound by the context (this mismatch will be fixed in next translation).
We define $[x]$ to be x instead of $([x]_0,\ [x]_1)$ to avoid an unnecessary η-expansion. Let's remark that the application in the definition of $[\![A]\!]$ also makes some unnecessary β-redexes appear.

In the context of CPS translations, such $\beta\eta$-redexes are called *administrative redexes* and it is standard to avoid introducing them, see *e.g.* [DF92, DN05]. A simple improvement in our setting is to define $[\![A]\!]$ as:

$$[\![A]\!] \quad := \quad \Sigma\, x_0 : [A]_0.\, [A]_1 \ @\ x_0$$

where @ is a smart application operator defined on preterms by:

$$\begin{aligned} (\lambda\, x : A.\, t) \ @\ u \quad &:= \quad t\{x := u\} \\ t \ @\ u \quad &:= \quad t\ u \text{ in other cases} \end{aligned}$$

In several works, for instance [BL11], a notation $t : [A]_1$ is used and has the same effect.

This variant still enjoys typing soundness:

Theorem 22. If $\Gamma \vdash M : A$ then $\Gamma \vdash [M]_0 : [A]_0$ and $\Gamma \vdash [M]_1 : [A]_1\ [M]_0$. As a consequence:

$$[\![\Gamma]\!] \vdash [M] : [\![A]\!]$$

Now that we have gone back to our setting, we can investigate about the logical power of this translation. The result is a bit disappointing:

Proposition 23. The packed parametricity translation $[\_]$ is conservative:

If $[\![A]\!]$ is inhabited in the closed context then A was already inhabited.

Proof. The proof if trivial: if M is of type $[\![A]\!]$ then $\pi_1(M)$ is of type $[A]_0$ and $[A]_0$ is A because it has no free variables. □

Of course, this does not mean that parametricity translation is not interesting, only that we can not use it as a syntactic model to extend the logical power of our system.

In [Las14], Lasson says that an axiom of closed type **Ax** is *provably parametric* if the type $\Pi\, h : \mathbf{Ax}.\, [\mathbf{Ax}]_1\ h$ is inhabited. For propositions or h-propositions, this condition is equivalent to the fact that the translation preserves the axiom in sense of Section 2.1. Indeed, axiom preservation is the fact that $\mathbf{Ax} \to \Sigma\, h : \mathbf{Ax}.\, [\mathbf{Ax}]_1\ h$ is inhabited. The preservation of an axiom **Ax** means that the parametricity translation can be extended to $\lambda\Pi_\omega + \mathbf{Ax}$.

Remark. It is not because the translation is conservative ($[\![\mathbf{Ax}]\!] \to \mathbf{Ax}$) that it preserves every axiom ($\mathbf{Ax} \to [\![\mathbf{Ax}]\!]$).

In his article, Lasson proved that FunExt and UIP are provably parametric.

Proposition 24. **FunExt** and **UIP** are provably parametric.

But he left open the question whether univalence is provably parametric. We formalized that it is indeed the case:

Proposition 25. **Univalence** is provably parametric.

It means that parametricity translation can be extended to $\lambda\Pi_\omega + \mathbf{Univalence}$. The proof does not seem very informative, "it just works". We let the reader refer to the formalization for the details. It can be found in the file `TemplateCoq/translations/param_original.v`.

2.4.3. Generous translation

In the previous translation, there was a mismatch between free (or global) and bound (or local) variables. The former come with a parametricity witness, while the later does not. We can solve this mismatch by making the parametricity witness available in the first component of the translation. This gives the following translation:

$$\begin{aligned}
[x]_0 &:= \pi_1(x) \\
[\lambda\, x : A.\, M]_0 &:= \lambda\, x : [\![A]\!].\, [M]_0 \\
[M\ N]_0 &:= [M]_0\, [N] \\
[\Pi\, x : A.\, B]_0 &:= \Pi\, x : [\![A]\!].\, [B]_0 \\
[\mathcal{U}_i]_0 &:= \mathcal{U}_i \\
[_]_1, [_] \text{ and } [\![_]\!] &\text{ are unchanged (but rely on the new } [_]_0)
\end{aligned}$$

We call this variant *generous*, because the proof of parametricity of the argument is available in translation $[\_]_0$ of functions ($x : [\![A]\!]$ instead of $x : [A]_0$). On the contrary, we call *cheap*, the parametricity translation of previous section. This variant still enjoys the same typing soundness theorem.
There is now something quite surprising: although the translation looks very similar to the previous one, it is logically more powerful. Indeed, this translation negates univalence.

Proposition 26. The translation $[\_]$ negates weak univalence.
Let **notUnivalence** be the type:

$$\Sigma\, A\, B : \mathcal{U}.\, (A \simeq B) \times \Sigma\, P : \mathcal{U} \to \mathcal{U}.\, P\, A \times \neg (P\, B)$$

then there exists a closed term of type $\vdash [\![\textbf{notUnivalence}]\!]$.

Proof. As always, the construction of the term is better understood using a proof assistant. Such a formalization is present in file `param_generous_packed.v`.
The intuition is as follows:

- A type is given by a type and a predicate on it. So, take A to be $(\mathbb{B}, \lambda\_.\, \mathbb{1})$. (or, equivalently, $[\![\mathbb{B}]\!]$)
- Take B to be $(\mathbb{1}, \lambda\_.\, \mathbb{B})$.
- They are equivalent because we can use the second component to build the functions ($[\![A \to B]\!] \cong [\![A]\!] \to [\![B]\!]$).
- Now take P to be "having two distinct elements on the first component". It is satisfied by B but not by A.

□

Remark. We could define an "unpacked generous translation" with:

$$\begin{aligned}
[\Pi\, x : A.\, B]_0 &:= \Pi\, (x : [A]_0)\, (x_t : [A]_1\ x).\, [B]_0 \\
[\Pi\, x : A.\, B]_1 &:= \lambda\, f.\, \Pi\, (x : [A]_0)\, (x_t : [A]_1 x).\, [B]_1\ (f\ x\ x_t)
\end{aligned}$$

This translation seems to share most of the properties of the packed generous translation.

2.4.4. N-parametricity

We define here a N-parametricity translation which is a kind of parametricity translation iterated N times for any $N : \mathbb{N}$. In this translation, a type is interpreted by a tower of a type equipped with N successive predicates, which we call a N-cube. Again, we can do a cheap or a generous translation, here we give the generous one.
The logical power of this translation remains to be studied. Our hope is that this translation would inspire an ω-parametricity translation using coinductive types to represent ω-cubes.
All construction of this section are universe polymorphic, we omit universe levels for simplicity.

Cubes We define $\mathbf{Cube}_n : \mathcal{U}$ and $\mathbf{El}_n : \mathbf{Cube}_n \to \mathcal{U}$:

$$\begin{aligned}
\mathbf{Cube}_0 &\coloneqq \mathcal{U} \\
\mathbf{Cube}_{n+1} &\coloneqq \Sigma\, C : \mathbf{Cube}_n.\, \mathbf{El}_n\ C \to \mathcal{U} \\
\mathbf{El}_0\ C &\coloneqq C \\
\mathbf{El}_{n+1}\ C &\coloneqq \Sigma\, x : \mathbf{El}_n\ \pi_1(C).\, \pi_2(C)\ x
\end{aligned}$$

Hence, a n-cube is a type equipped with a tower of predicates over it which are packed using Σ types (associating on left). $\mathbf{El}_n$ gives the type of inhabitants of such a cube: a point in the base type and proofs that the successive predicates hold.
For instance, the first cubes are:

$$\begin{aligned}
\mathbf{Cube}_1 &= \Sigma\, A_0 : \mathcal{U}.\, A_0 \to \mathcal{U} \\
\mathbf{El}_1\ (A_0,\, A_1) &= \Sigma\, x_0 : A_0.\, A_1\ x_0 \\
\mathbf{Cube}_2 &= \Sigma\, (A_0,\, A_1) : (\Sigma\, A_0 : \mathcal{U}.\, A_0 \to \mathcal{U}).\ \ (\Sigma\, x_0 : A_0.\, A_1\ x_0) \to \mathcal{U} \\
\mathbf{El}_2\ ((A_0,\, A_1),\, A_2) &= \Sigma\, x_{01} : (\Sigma\, x_0 : A_0.\, A_1\ x_0).\ \ A_2\ x_{01}
\end{aligned}$$

We now define $\mathbf{Type}_n : \mathbf{Cube}_n$ the n-cube of n-cubes. The definition is made such that we have $\mathbf{El}_n\ \mathbf{Type}_n \simeq_\beta \mathbf{Cube}_n$.

$$\begin{aligned}
\mathbf{Type}_0 &\coloneqq \mathcal{U} \\
\mathbf{Type}_{n+1} &\coloneqq (\mathbf{Type}_n,\, \lambda\, C : \mathbf{Cube}_n.\, \mathbf{El}_n\ C \to \mathcal{U})
\end{aligned}$$

Translation Let N be in $\mathbb{N}$. We can now define the N-parametricity translation. To do this we define $N+1$ translation maps $[\_]_0, [\_]_1, \dots, [\_]_N$ by induction on the term and on n. We will write $[\![A]\!]_n$ for $\mathbf{El}_n\, [A]_n$, $[A]$ for $[A]_N$ and $[\![A]\!]$ for $[\![A]\!]_N$. We will ensure:

$$\forall\, n \le N, \quad \Gamma \vdash M : A \quad \Rightarrow \quad [\![\Gamma]\!] \vdash [M]_n : [\![A]\!]_n$$

where contexts are translated pointwise by $[\![\_]\!]$.
We will also ensure the following invariant:

$$\pi_1([M]_{n+1}) \simeq_\beta [M]_n \qquad (\star)$$

which means that to define the $n+1$ component of the translation from the n one, we only have to give a proof for the last predicate.

Universe is translated by:

$$[\mathcal{U}]_n \quad \coloneqq \quad \mathbf{Type}_n$$

and hence for all $n : \mathbb{N}$, $[\![\mathcal{U}]\!]_n \simeq_\beta \mathbf{Cube}_n$.

Product types are translated by:

$$\begin{aligned}[\Pi\, x : A.\ B]_0 &\coloneqq \Pi\, x : [\![A]\!].\, [B]_0 \\ [\Pi\, x : A.\ B]_{n+1} &\coloneqq ([\Pi\, x : A.\ B]_n, \lambda\, f : [\![\Pi\, x : A.\ B]\!]_n.\, \Pi\, x : [\![A]\!].\, \pi_2([B]_{n+1}\ (\mathbf{ev}_n\ f\ x))\end{aligned}$$

where $\mathbf{ev}_n\ (f_0, \ldots, f_n)\ x = (f_0\ x, \ldots, f_n\ x)$:

$$\begin{aligned}\mathbf{ev}_0\ f\ x &\coloneqq f\ x \\ \mathbf{ev}_{n+1}\ f\ x &\coloneqq (\mathbf{ev}_n\ \ \pi_1(f)\ x,\ \pi_2(f)\ x)\end{aligned}$$

This operator is typed as follows:

$$\text{If } \Gamma \vdash f : [\![\Pi\, x : A.\ B]\!]_n \text{ then } \Gamma,\ x : [\![A]\!] \vdash \mathbf{ev}_n\ f\ x : [\![B]\!]_n$$

The translation is generous because the translation $[\,\_\,]_n$ of Π types involves $[\![A]\!]$ instead of $[A]_n$.

Functions

$$\begin{aligned}[\lambda\, x : A.\ M]_0 &\coloneqq \lambda\, x : [\![A]\!].\, [M]_0 \\ [\lambda\, x : A.\ M]_{n+1} &\coloneqq ([\lambda\, x : A.\ M]_n, \lambda\, x : [\![A]\!].\, \pi_2([M]_{n+1}))\end{aligned}$$

we can check than $\mathbf{ev}_n\ [\lambda\, x : A.\ M]_n\ x \simeq_\beta [M]_n$ (using the invariant ($\star$)).

Application is translated by:

$$[M\ N]_n \quad \coloneqq \quad \mathbf{ev}_n\, [M]_n\, [N]$$

Variables

$$[x]_n \quad \coloneqq \quad p_n(x)$$

where $p_n(M) = \pi_1(\pi_1(\ldots M)) = \pi_1^{N-n}(M) \quad (p_n : \mathbf{Cube}_N \to \mathbf{Cube}_n)$.

The translations maps satisfy the substitution lemma and computational soundness (it requires η-conversion and surjective pairing). Hence we have:

Proposition 27. If $\Gamma \vdash t : A$ then for all $n \leq N$:

$$[\![\Gamma]\!] \vdash [M]_n : [\![A]\!]_n$$

All Models	Category with Families
Syntactic Model	CwF where all equalities are refl Defined on raw syntax
Program translation	A context is translated by a context, a term by a term, ...

Figure 2.5.: Different kind of models for type theory

2.5. The emergence of syntactic models

We want to compare program translations with respect to more traditional ways of giving models of type theory like Categories with Families.

The main advantage of program translations is that they bypass the problem of defining the interpretation map discussed in Section 1.5.3. Indeed, for translations, the interpretation is defined on untyped syntax and type soundness is checked afterward. This is only possible because *the target is untyped syntax*, which is not the case with CwFs.
The program translations also provide relatively simple models. This is in particular due to the fact that many constructions are interpreted transparently: contexts by contexts, universes by universes, etc. Among those constructions there is the conversion rule. The conversion rule is the only rule that is not syntax directed in $\lambda\Pi_\omega$ and program translation does not try to do anything with it: they interpret conversion by conversion.
Program translations also seem to be quite flexible. For instance, for universes, they can account both for Tarski style and Russell style, as long as it is the same style in the source and in the target.
Last, with respect to CwF, program translations require a very weak metatheory because all the requirements are made on the target theory and not on the metatheory. For instance, to define a model of a type theory with many universes we assume that the target theory has many universes but the translation can be defined and proved correct in a very weak metatheory. On the contrary, to define the corresponding CwF in the metatheory we need all those universes in it.
We realized that some models which are not program translations share several of those properties. For instance, for the standard model, it is possible to define an interpretation map on raw syntax. A context is mapped to a type, a type to a type family and a term to a term (Figure 2.6). Type soundness can be checked afterward. We propose to name this kind of models *syntactic models*. For the standard model, let's remark that the translation of a term depends on the context, so with respect to program translations, the valuable property of *locality* is lost.
It seems that we can do the same for the setoid model with definitional $\mathbb{P}$: a raw context is interpreted by a raw term and we show afterward that if the context is well-formed the term is of type **Setoid**. In both models, conversion is interpreted conversion. And indeed, in the CwFs corresponding to those two models all the equalities were given by reflexivity. On the contrary, for the setoid model with h-propositions, all the equalities are not given by reflexivity

$$
\begin{array}{rcl}
[\![\cdot]\!] & \coloneqq & \mathbb{1} \\
[\![\Gamma, x : A]\!] & \coloneqq & \Sigma\, \gamma : [\![\Gamma]\!] .\, [\![A]\!]_\Gamma\, \gamma \\
[\![\Pi\, x : A.\, B]\!]_\Gamma & \coloneqq & \lambda\, \gamma : [\![\Gamma]\!] .\, \Pi\, x : [\![A]\!]_\Gamma\, \gamma.\, [\![B]\!]_{\Gamma, x:A}\, (\gamma, x) \\
[\lambda\, x : A.\, M]_\Gamma & \coloneqq & \lambda\, (\gamma : [\![\Gamma]\!])\, (x : [\![A]\!]_\Gamma\, \gamma).\, [M]_{\Gamma, x:A}\, (\gamma, x) \\
[M\ N]_\Gamma & \coloneqq & \lambda\, \gamma : [\![\Gamma]\!] .\, [M]_\Gamma\, \gamma\, ([N]_\Gamma\, \gamma) \\
[\mathcal{U}_i]_\Gamma & \coloneqq & \lambda\, \_ : [\![\Gamma]\!] .\, \mathcal{U}_i \\
[x_n]_\Gamma & \coloneqq & \lambda\, \gamma : [\![\Gamma]\!] .\, \pi_2(\pi_1^n(\gamma)) \qquad \text{if } x_n \text{ is the } n\text{th variable of } \Gamma
\end{array}
$$

Figure 2.6.: Standard model as a syntactic model

and it does not seem possible to define it on raw syntax.
Categories with Families are used to interpret substitution and conversion (the "substitution calculus" part) in mathematical structure where there are not necessarily present. In the case of syntactic models (including program translations), substitution and conversion are interpreted by themselves so the use of a CwF is superfluous. Let's remark that by postcomposing a program translation by the standard model, we always get a new CwF where all equalities are given by reflexivity.

The precise definition of syntactic models remains to be given but some criteria are emerging:

- they are defined on raw syntax
- they give rise to a CwF where all equalities are given by reflexivity

The definition of syntactic model could be the following. However, this definition does not encompasses program translations (where a precontext is translated to a precontext), hence a more general definition should probably be given.
A syntactic model from source $\mathcal{S}$ to target $\mathcal{T}$ is given by:

- a map $[\![\_]\!]$ from precontexts of $\mathcal{S}$ to preterms of $\mathcal{T}$
- a map $[\![\_]\!]$ from pretypes of $\mathcal{S}$ to preterms of $\mathcal{T}$
- a map $[\ \_\ ]$ from preterms of $\mathcal{S}$ to preterms of $\mathcal{T}$
- a closed type $\mathbf{Ctx}$ in $\mathcal{T}$
- a type family $\mathbf{Ty} : \mathbf{Ctx} \to \mathcal{U}$ in $\mathcal{T}$
- a type family $\mathbf{Tm} : \Pi\, \Gamma.\, \mathbf{Ty}\ \Gamma \to \mathcal{U}$ in $\mathcal{T}$

And the expected typing soundness property is:

$$
\begin{array}{rcl}
\Gamma \vdash & \Rightarrow & \cdot \vdash [\![\Gamma]\!] : \mathbf{Ctx} \\
\Gamma \vdash A & \Rightarrow & \cdot \vdash [\![A]\!] : \mathbf{Ty}\ [\![\Gamma]\!] \\
\Gamma \vdash M : A & \Rightarrow & \cdot \vdash [M] : \mathbf{Tm}\ [\![\Gamma]\!]\ [\![A]\!]
\end{array}
$$

3. Template Coq and the translation plugin

In this chapter we present Template Coq, a reification plugin for Coq which reifies the syntax, the typing and some commands. In the first section we present the reification of the syntax. Template Coq provides a representation of Coq terms as an inductive type, as close as possible as the original implementation, and functions to move from terms to syntax and back. Then, we present the reification of the typing. This one is given as an inductive family over the terms. A type checking algorithm is also provided. Next comes the reification of some commands, with the application of writing Coq plugins directly in Coq. Last, we will focus on a translation plugin that we defined using Template Coq. This plugin allows to define and use program translations, as defined in the previous chapter, directly in Coq.
This chapter is heavily based on the article *Towards Certified Meta-Programming with Typed Template-Coq* (Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau and Nicolas Tabareau) [ABC+18] published at ITP 2018. It is mainly Section 3.4 on the translation plugin which is more detailed.

3.1. Reification of Coq Terms

Reification of syntax. The central piece of Template Coq is the inductive type `term` defined in Figure 3.1 which represents the syntax of Coq terms. This inductive follows directly the `constr` datatype of Coq terms in the OCaml code of Coq, except for the use of OCaml's native arrays and strings. An upcoming extension of Coq [AGST10] with such features should solve this mismatch.
Constructor `tRel` represents variables bound by abstractions (introduced by `tLambda`), dependent products (introduced by `tProd`) and local definitions (introduced by `tLetIn`), the natural number is a De Bruijn index. The `name` is a printing annotation.
Sorts are represented with `tSort`, which takes a `universe` as argument. A universe is the supremum of a (non-empty) list of level expressions, and a level is either `Prop`, `Set`, a global level or a De Bruijn polymorphic level variable.

```
Inductive level := lProp | lSet | Level (_ : string) | Var (_ : ℕ).
Definition universe := list (level * bool). (* level+1 if true *)
```

The application (introduced by `tApp`) is n-ary. The three constructors `tConst`, `tInd` and `tConstruct` represent references to constants (definitions or axioms), inductives, and constructors of an inductive type. The `universe_instances` are non-empty only for universe polymorphic constants. Finally, `tCase` represents pattern-matchings, `tProj` primitive projections, `tFix` fixpoints and `tCoFix` cofixpoints.

```
Inductive term : Set :=
| tRel      : ℕ → term
| tVar      : ident → term
| tEvar     : ℕ → list term → term
| tSort     : universe → term
| tCast     : term → cast_kind → term → term
| tProd     : name → term → term → term
| tLambda   : name → term → term → term
| tLetIn    : name → term → term → term → term
| tApp      : term → list term → term
| tConst    : kername → universe_instance → term
| tInd      : inductive → universe_instance → term
| tConstruct: inductive → ℕ → universe_instance → term
| tCase     : inductive * ℕ → term → term → list (ℕ * term) → term
| tProj     : projection → term → term
| tFix      : mfixpoint term → ℕ → term
| tCoFix    : mfixpoint term → ℕ → term.
```

Figure 3.1.: Representation of the syntax in Template Coq

Quoting and unquoting of terms. Template Coq provides a lifting from concrete syntax to reified syntax (quoting) and the converse (unquoting). It can reify and reflect all kernel Coq terms.
The command `Quote Definition` reifies the syntax of a term. For instance,

```
Quote Definition f := (λ x ⇒ x + 0).
```

generates the term `f` defined as

```
f = tLambda (nNamed "x") (tInd {| inductive_mind := "Coq.Init.Datatypes
    .nat"; inductive_ind := 0 |} []) (tApp (tConst "Coq.Init.Nat.add"
    []) [tRel 0; tConstruct {| inductive_mind := "Coq.Init.Datatypes.
    nat"; inductive_ind := 0 |} 0 []) : term
```

On the converse, the command `Make Definition` constructs a term from its syntax. This example below defines `zero` to be `0` of type ℕ.

```
Make Definition zero := tConstruct (mkInd "Coq.Init.Datatypes.nat" 0)
                                   0 [].
```

where `mkInd na k` is the k^{th} inductive of the mutual block of the name `na`.

Reification of environment. In Coq, the meaning of a term is relative to an environment, which must be reified as well. Environments consist of three parts: (i) a graph of universes (ii) declarations of definitions, axioms and inductives (iii) a local context registering types of De Bruijn indexes.

As we have seen in the syntax of terms, universe levels are not given explicitly in Coq. Instead, there are level variables and constraints between level variables are registered in a *graph of universes*. This is the way typical ambiguity is implemented in Coq. A constraint is given by two levels and a `constraint_type` (`Lt`, `Le` or `Eq`):

```
Definition univ_constraint := Level * constraint_type * Level.
```

Then the graph is given by a set of level variables and a set of constraints. Sets, coming from the Coq standard library, are implemented using lists without duplicates. `LevelSet.t` means the type `t` of the module `LevelSet`.

```
Definition uGraph := LevelSet.t * ConstraintSet.t.
```

Functions to query the graph are provided. For the moment, they rely on a naive implementation of the Bellman-Ford algorithm. `check_leq u1 u2` checks if the graph enforces `u1` $\leq$ `u2` and `no_universe_inconsistency` checks that the graph has no negative cycle.
Constant and inductive declarations are grouped together, properly ordered according to dependencies, in a global context `global_ctx`, which is a list of global declarations `global_decl`.

```
Inductive global_decl :=
| ConstantDecl  : ident → constant_decl   → global_decl
| InductiveDecl : ident → minductive_decl → global_decl.
```

Definitions and axioms just associate a name to a universe context, and two terms for the optional body and type. Inductives are more involved:

```
(* Declaration of one inductive type *)
Record inductive_body := { ind_name : ident;
   ind_type  : term; (* closed arity *)
   ind_kelim : list sort_family; (* allowed elimination sorts *)
   (* names, types, number of arguments of constructors *)
   ind_ctors : list (ident * term * nat);
   ind_projs : list (ident * term) (* names and types of projections *)}.

(* Declaration of a block of mutual inductive types *)
Record minductive_decl := { ind_npars : nat; (* number of parameters *)
   ind_bodies : list inductive_body; (* inductives of the mutual block *)
   ind_universes : universe_context (* universe constraints *) }.
```

In Coq internals, there are in fact two ways of representing a declaration: either as a "declaration" or as an "entry". The kernel takes entries as input, type-check them and elaborate them to declarations. In Template Coq, we provide both, and provide an erasing function `mind_decl_to_entry` from declarations to entries for inductive types.
Finally, local contexts are just lists of local declarations: a type for lambda bindings, and a type and a body for let bindings.

Quoting and unquoting the environment Template Coq provides the command `Quote Recursively Definition` to quote an environment. This command crawls the environment and quotes all declarations needed to typecheck a given term.

The other way, the commands `Make Inductive` allows declaring an inductive type from its entry. For instance, the following redefines a copy of ℕ:

```
Make Inductive (mind_decl_to_entry
    {| ind_npars := 0;  ind_universes := [];
       ind_bodies := [{|
         ind_name := "nat";
         ind_type := tSort [(lSet, false)];
         ind_kelim := [InProp; InSet; InType];
         ind_ctors := [("O", tRel 0, 0);
                       ("S", tProd nAnon (tRel 0) (tRel 1), 1)];
         ind_projs := [] |}] |} ).
```

More examples of use of quoting/unquoting commands can be found in the Template Coq repository in th file `template-coq/demo.v`.

3.2. Type Checking Coq in Coq

In Figure 3.2, we present (an excerpt of) the specification of the typing judgment of the kernel of Coq using the inductive family `typing`. It represents all the typing rules of Coq[1]. This includes the basic dependent lambda calculus with lets, global references to inductives and constants, the `match` construct and primitive projections. Universe polymorphic definitions and the well-formedness judgment for global declarations are dealt with as well.
The only ingredients missing are the guard check for fixpoint and productivity of cofixpoints and the positivity condition of mutual (co-)inductive types. They are work-in-progress.
The typing judgment `typing` is mutually defined with `typing_spine` to account for n-ary applications. Untyped reduction `red1` and cumulativity `cumul` can be defined separately.

Implementation. To test this specification, we have implemented the basic algorithms for type-checking in Coq. In particular, we implemented type inference: given a context and a term, output its type or produce a type error. All the rules of type inference are straightforward except for cumulativity. The cumulativity test is implemented by comparing head normal forms for a fast-path failure and potentially calling itself recursively, unfolding definitions at the head in Coq's kernel in case the heads are equal. We implemented weak-head reduction by mimicking Coq's kernel implementation, which is based on an abstract machine inspired by the KAM. Coq's machine optionally implements a variant of lazy, memoizing evaluation (which can have mixed results, see Coq's PR #555 for example), that feature has not been implemented yet.
The main difference with the OCaml implementation is that all of the functions are required to be shown terminating in Coq. One possibility could be to prove the termination of type-checking separately but this amounts to prove in particular the normalization of CIC which is a complex task. Instead, we simply add a fuel parameter to make them syntactically recursive and make `OutOfFuel` a type error.

[1] We do not treat metavariables which are absent from kernel terms and require a separate environment for their declarations.

```
Inductive cumul (Σ : global_ctx) (Γ : context) : term → term → Prop
:= | cumul_refl t u : leq_term (snd Σ) t u = true → Σ ; Γ ⊢ t ≤ u
| cumul_red_l t u v : red1 Σ Γ t v → Σ ; Γ ⊢ v ≤ u → Σ ; Γ ⊢ t ≤ u
| cumul_red_r t u v : Σ ; Γ ⊢ t ≤ v → red1 Σ Γ u v → Σ ; Γ ⊢ t ≤ u
where " Σ ; Γ ⊢ t ≤ u " := (cumul Σ Γ t u).

Inductive typing (Σ : global_ctx) (Γ : context) : term → term → Set
:= | type_Rel n : ∀ (H : n < List.length Γ),
    Σ ; Γ ⊢ tRel n : lift0 (S n) (safe_nth Γ (exist _ n H)).(decl_type
    )
| type_Sort (l : level) :
    Σ ; Γ ⊢ tSort (Universe.make l) : tSort (Universe.super l)
| type_Prod n t b s1 s2 :
    Σ ; Γ ⊢ t : tSort s1 → Σ ; Γ , vass n t ⊢ b : tSort s2 →
    Σ ; Γ ⊢ tProd n t b : tSort (max_universe s1 s2)
| type_App t l t_ty t' :
    Σ ; Γ ⊢ t : t_ty → typing_spine Σ Γ t_ty l t' →
    Σ ; Γ ⊢ tApp t l : t'
| ...
where " Σ ; Γ ⊢ t : T " := (typing Σ Γ t T)

with typing_spine Σ Γ : term → list term → term → Prop :=
| type_spine_nil ty : typing_spine Σ Γ ty [] ty
| type_spine_const hd tl na A B B' T :
    Σ ; Γ ⊢ T ≤ tProd na A B → Σ ; Γ ⊢ hd : A →
    typing_spine Σ Γ (subst0 hd B) tl B' →
    typing_spine Σ Γ T (cons hd tl) B'
```

Figure 3.2.: Typing judment for `terms`, excerpt

Bootstrapping it. We can extract this checker to OCaml and reuse the setup described in the previous section to connect it with the reifier and easily derive a (partialy verified) alternative checker for Coq's `.vo` object files. Our plugin provides a new command `Template Check` for typechecking definitions using the alternative checker, that can be used as follows:

```
Require Import Template.TemplateCoqChecker List. Import ListNotations.
Definition foo := List.map (λ x ⇒ x + 3) [0; 1].
Template Check foo.
```

3.3. Reification of Coq Commands

Coq plugins need to interact with the environment, for example by repeatedly looking up definitions by name, declaring new constants using fresh names, or performing computations. It is desirable to allow such programs to be written in Coq (Gallina) because of the two following advantages. Plugin-writers no longer need to understand the OCaml implementation of Coq and plugins are no longer sensitive to changes made in the OCaml implementation.
In general, interactions with the environment have side effects—*e.g.* the declaration of new constants—which must be described in Coq's pure setting. To overcome this difficulty, we use the standard "free" monadic setting to represent the operations involved in interacting with the environment, as done for instance in Mtac [ZDK+15].
`TemplateMonad` is an inductive family (Figure 3.3) such that `TemplateMonad A` represents a program which will finally output a term of type `A`. There are special constructors `tmReturn` and `tmBind` to provide (freely) the basic monadic operations. We use the monadic syntactic sugar `x ← t ; u` for `tmBind t (λ x ⇒ u)`.
The other operations of the monad can be classified in two categories:

- the traditional Coq operations (`tmDefinition` to declare a new definition, etc.)
- the quoting and unquoting operations to move between Coq term and their syntax or to work directly on the syntax (`tmMkInductive` to declare a new inductive from its syntax for instance).

An overview of available commands is given in Table 3.1.
A program `prog` of type `TemplateMonad A` can be executed with the command:

```
Run TemplateProgram prog.
```

This command is thus an interpreter for `TemplateMonad` programs, implemented in OCaml as a traditional Coq plugin. The term produced by the program is discarded but, and it is the point, a program can have many side effects like declaring a new definition or a new inductive type, printing something, ...

Let's look at some examples. The following program adds two definitions `foo := 12` and `bar := foo + 1` to the current context.

```
Run TemplateProgram (foo ← tmDefinition "foo" 12 ;
                         tmDefinition "bar" (foo +1)).
```

```
Inductive TemplateMonad : Type → Type :=
(* Monadic operations *)
| tmReturn : ∀ {A}, A → TemplateMonad A
| tmBind   : ∀ {A B},
    TemplateMonad A → (A → TemplateMonad B) → TemplateMonad B

(* General operations *)
| tmPrint      : ∀ {A}, A → TemplateMonad unit
| tmFail       : ∀ {A}, string → TemplateMonad A
| tmEval       : reductionStrategy → ∀ {A}, A → TemplateMonad A
| tmDefinition : ident → ∀ {A}, A → TemplateMonad A
| tmAxiom      : ident → ∀ A, TemplateMonad A
| tmLemma      : ident → ∀ A, TemplateMonad A
| tmFreshName  : ident → TemplateMonad ident
| tmAbout      : ident → TemplateMonad (option global_reference)
| tmCurrentModPath : unit → TemplateMonad string

(* Quoting and unquoting operations *)
| tmQuote          : ∀ {A}, A  → TemplateMonad term
| tmQuoteRec       : ∀ {A}, A  → TemplateMonad program
| tmQuoteInductive: kername → TemplateMonad mutual_inductive_entry
| tmQuoteConstant : kername → bool → TemplateMonad constant_entry
| tmMkDefinition  : ident → term → TemplateMonad unit
| tmMkInductive   : mutual_inductive_entry → TemplateMonad unit
| tmUnquote       : term  → TemplateMonad {A : Type & A}.
| tmUnquoteTyped  : ∀ A, term → TemplateMonad A
```

Figure 3.3.: The monad of commands

Vernacular command	Reified command with its arguments	Description
Eval	tmEval red t	Returns the evaluation of t following the evaluation strategy red (cbv, cbn, hnf, all or lazy)
Definition	tmDefinition id t	Makes the definition id := t and returns the created constant id
Axiom	tmAxiom id A	Adds the axiom id of type A and returns the created constant id
Lemma	tmLemma id A	Generates an obligation of type A, returns the created constant id after all obligations close
About or Locate	tmAbout id	Returns Some gr if id is a constant in the current environment and gr is the corresponding global reference. Returns None otherwise.
	tmQuote t	Returns the syntax of t (of type term)
	tmQuoteRec t	Returns the syntax of t and all the declarations on which it depends
	tmQuoteInductive kn	Returns the declaration of the inductive kn
	tmQuoteConstant kn b	Returns the declaration of the constant kn, if b is true the implementation bypass opacity to get the body of the constant
Make Definition	tmMkDefinition id tm	Adds the definition id := t where t is denoted by tm
Make Inductive	tmMkInductive d	Declares the inductive denoted by the declaration d
	tmUnquote tm	Returns the pair (A;t) where t is the term whose syntax is tm and A it's type
	tmUnquoteTyped A tm	Returns the term whose syntax is tm and checks that it is indeed of type A

Table 3.1.: Main Template Coq commands

The program below asks the user to provide an inhabitant of ℕ (here we provide `3 * 3`), records it in the lemma `foo`, prints its normal form, and records the syntax of its normal form in `foo_nf_syntax` (hence of type `term`). We use Program's obligation mechanism[2] to ask for missing proofs. And the rest of the program is running when the obligation is closed. This allows the implementation of *interactive* plugins.

```
Run TemplateProgram (foo ← tmLemma "foo" ℕ ;
                     nf  ← tmEval all foo ;
                     tmPrint "normal form: " ; tmPrint nf ;
                     nf_ ← tmQuote nf ;
                     tmDefinition "foo_nf_syntax"  nf_).
Next Obligation. exact (3 * 3). Defined.
```

3.3.1. Writing Coq plugins in Coq

The reification of syntax, typing and commands of Coq allow writing a Coq plugin directly inside Coq, without requiring another language like OCaml and an external compilation phase. We develop here a toy example to show the methodology of writing plugins in Template Coq. This example is a plugin which adds a new constructor to an existing inductive type. In next section, we will present a more involved plugin, the translation plugin.

Given an inductive type `I`, we want to declare a new inductive type `I'` which corresponds to `I` plus one more constructor. For instance, let's say that we have a syntax for lambda calculus:

```
Inductive tm : Set :=
    | var : nat → tm  | lam : tm → tm  | app : tm → tm → tm.
```

And that in some part of our development, we want to consider a variation of `tm` with a new constructor, *e.g.* a "let-in" constructor. Then we declare `tm'` with the plugin by:

```
Run TemplateProgram
    (add_constructor tm "letin" (λ tm' ⇒ tm' → tm' → tm')).
```

This command has the same effect as declaring the inductive `tm'` by hand:

```
Inductive tm' : Set :=
    | var' : nat → tm'          | lam' : tm' → tm'
    | app' : tm' → tm' → tm'    | letin : tm' → tm' → tm'.
```

but with the benefit that if `tm` is changed, for instance by adding one new constructor, then `tm'` is automatically changed accordingly. We provide other examples, *e.g.* with mutual inductives, in the file `test-suite/add_constructor.v` of the Template Coq repository.

We will see that it is fairly easy to define this plugin using Template Coq. The main function is `add_constructor` which takes an inductive type `ind` (whose type is not necessarily `Type`

[2] In Coq, a proof obligation is a goal which has to be solved to complete a definition. Obligations were introduced by Sozeau [Soz07] in the Program mode.

if it is an inductive family), a name `idc` for the new constructor and the type `ctor` of the new constructor, abstracted with respect to the new inductive.

```
Definition add_constructor {A} (ind : A) (idc : ident) {B} (ctor : B)
    : TemplateMonad unit
    := tm ← tmQuote ind ;
       match tm with
       | tInd ind0 _ ⇒
         decl ← tmQuoteInductive (inductive_mind ind0) ;
         ctor ← tmQuote ctor ;
         d' ← tmEval lazy (add_ctor decl ind0 idc ctor) ;
         tmMkInductive d'
       | _ ⇒ tmFail "The provided term is not an inductive"
       end.
```

It works in the following way. First the inductive type `ind` is quoted, the obtained term `tm` is expected to be a `tInd` constructor otherwise the function fails. Then the declaration of this inductive is obtained by calling `tmQuoteInductive`, the constructor is reified too, and an auxiliary function is called to add the constructor to the declaration. After evaluation, the new inductive type is added to the current context with `tmMkInductive`.
It remains to define the `add_ctor` auxiliary function to complete the definition of the plugin. It takes a `minductive_decl` which is the declaration of a block of mutual inductive types and returns another `minductive_decl`.

```
Definition add_ctor (mind : minductive_decl) (ind0 : inductive)
                    (idc : ident) (ctor : term) : minductive_decl
  := let i0 := inductive_ind ind0 in
     {| ind_npars := mind.(ind_npars) ;
        ind_bodies := map_i (λ (i : nat) (ind : inductive_body) ⇒
          {| ind_name := tsl_ident ind.(ind_name) ;
             ind_type  := ind.(ind_type) ;
             ind_kelim := ind.(ind_kelim) ;
             ind_ctors :=
               let ctors := map (λ '(id, t, k) ⇒ (tsl_ident id, t, k))
                                ind.(ind_ctors) in
               if Nat.eqb i i0 then
                let n := length mind.(ind_bodies) in
                let typ := try_remove_n_lambdas n ctor in
                ctors ++ [(idc, typ, 0)]
               else ctors;
             ind_projs := ind.(ind_projs) |})
        mind.(ind_bodies) |}.
```

The declaration of the block of mutual inductive types is a record. The field `ind_bodies` contains the list of declarations of each inductive of the block. We see that most of the fields of the records are propagated, except for the names which are translated to add some primes and

`ind_ctors`, the list of types of constructors, for which, in the case of the relevant inductive (i_0 is its number), the new constructor is added.

3.4. The translation plugin

Let's plug program translations of previous chapter and Template Coq together. We present here a plugin that we developed in order to compute the translation of a Coq term given a program translation.
We suppose a translation ($[\![\_]\!]$, $[\![\![\_]\!]\!]$) from Coq to Coq given. The plugin provides four commands `Translate`, `TranslateRec`, `Implement` and `ImplementExisting`.

- `Translate` computes the translation $[\![M]\!]$ of a term M.
- `TranslateRec` computes the translation of a term and of all constant on which it depends.
- `Implement` computes the translation $[\![\mathbf{Ax}]\!]$ of a type **Ax**, then asks the user to inhabit $[\![\mathbf{Ax}]\!]$ in proof mode. Finally, if the user succeeded, it declares an axiom of type **Ax**. If the program translation is correct (*cf.* Section 2.1), it ensures that the axiom does not break consistency.
- `ImplementExisting` is used to provide the translation of some terms "by hand". It can be used to "implement" an existing axiom, but it is mainly useful to experiment with partial translations. Defining complete translation is indeed difficult and this command can be used, for instance, to provide the translation of an inductive type.

To work, the plugin needs a translation. A translation is given by the following record:

```
Class Translation :=
  { tsl_id  : ident → ident ;
    tsl_tm  : tsl_context → term → tsl_result term ;
    tsl_ty  : option (tsl_context → term → tsl_result term) ;
    tsl_ind : tsl_context → string → kername → mutual_inductive_body
              → tsl_result (tsl_table * list mutual_inductive_body) }.
```

This record is a `Class` so that, using type classes inference, when a translation is provided, it is automatically found by Coq.

- `tsl_ident` is how identifiers are translated, it will always be `(λ id ⇒ id ++ "ᵗ")` for us.
- `tsl_tm` is the main translation function implementing $[\![\_]\!]$. It takes a `term` and returns a `term`. The translation context contains the global environment and the previously translated constants, see below. The result is in the monad `tsl_result` which is an error monad:

```
Inductive tsl_error :=
| NotEnoughFuel
| TranslationNotFound (id : ident)
| TranslationNotHandeled
| TypingError (t : type_error).
```

The returned term can be of any type. `tsl_tm` is used by the commands `Translate` and `TranslateRec`.

- `tsl_ty` is the function translating types $[\![\_]\!]$. The returned term is expected to be a type. This function is used by the commands `Implement` and `ImplementExisting`, if it is not provided they are not available. This is the case for models which does not translate a type by a type (for instance, the standard model, the setoid model, ...).
- Last, `tsl_ind` is the function translating inductive types. It returns:
  - an extended translation table with the translations of the inductive type and its constructor
  - a list of inductive declarations which are used in the translation of the inductive type. Generally, an inductive is translated either by itself (in which case the list is empty), or by a new inductive whose constructors are the the original constructors translated (in which case the list is of length one).

  The second argument of `tsl_ind` is technical: it is the path to the module in which the new inductives will be declared.

Translation context In the translation plugin, the constants (definitions, axioms, inductive type or constructor), are translated one by one. They are recorded in a translation table so that the constants are not retranslated each time they appear. The table is the list of the translated constants together with their translation.

```
Definition tsl_table := list (global_reference * term).
```

Thus, the `tConst` case in the `tsl_tm` function is generally implemented by:

```
| tConst s univs ⇒ lookup_tsl_table' E (ConstRef s)
```

and similarly for `tInd` and `tConstruct`.
Some translation that we implemented need to access the global environment in which the considered term makes sense (see Section 3.6). That's why we define a translation context to be a global environment and a translation table:

```
Definition tsl_context := global_context * tsl_table.
```

3.5. Parametricity

Let's describe the use of the plugin for the parametricity translation. It provides an alternative implementation of Keller and Lasson's plugin ParamCoq [LK]. For the moment, only the unary case is implemented.

$$
\begin{aligned}
[M]_0 &= M \\
[x]_1 &= x_t \\
[\prod x : A.\ B]_1 &= \lambda f.\ \prod (x : [A]_0)\,(x_t : [A]_1\ x).\ [B]_1\ (f\ x) \\
[\lambda x : A.\ M]_1 &= \lambda\,(x : [A]_0)\,(x_t : [A]_1\ x).\ [M]_1 \\
[\mathcal{U}_i]_1 &= \lambda\, A : \mathcal{U}_i.\ A \to \mathcal{U}_i \\
[\![\Gamma, x : A]\!] &= [\![\Gamma]\!], x : [A]_0, x_t : [A]_1\ x
\end{aligned}
\qquad\qquad
\frac{\Gamma \vdash M : A}{\begin{array}{c}[\![\Gamma]\!] \vdash [M]_0 : [A]_0 \\ [\![\Gamma]\!] \vdash [M]_1 : [A]_1\ [M]_0\end{array}}
$$

Figure 3.4.: Unary parametricity translation and soundness theorem, excerpt from Section 2.4.1

The parametricity translation that we use here is the "original one", described in Section 2.4.1. It is reminded in Figure 3.4. The two components of the translation $[\_]_0$ and $[\_]_1$ are implemented by two recursive functions `tsl_param`$_0$ and `tsl_param`$_1$.

```
Fixpoint tsl_param0 (n : nat) (t : term) {struct t} : term :=
match t with
| tRel k ⇒ if k >= n then (* global variable *) tRel (2*k-n+1)
                     else (* local  variable *) tRel k
| tProd na A B ⇒ tProd na (tsl_param0 n A) (tsl_param0 (n+1) B)
| _ ⇒ …
end.
```

```
Fixpoint tsl_param1 (E : tsl_table) (t : term) : term :=
match t with
| tRel k  ⇒ tRel (2 * k)
| tSort s ⇒ tLambda (nNamed "A") (tSort s)
                    (tProd nAnon (tRel 0) (tSort s))
| tProd na A B ⇒
    let A0 := tsl_param0 0 A in let A1 := tsl_param1 E A in
    let B0 := tsl_param0 1 B in let B1 := tsl_param1 E B in
    tLambda (nNamed "f") (tProd na A0 B0)
    (tProd na (lift0 1 A0)
           (tProd (tsl_name na) (subst_app (lift0 2 A1) [tRel 0])
                  (subst_app (lift 1 2 B1) [tApp (tRel 2) [tRel 1] ])))
| tConst s univs ⇒ lookup_tsl_table' E (ConstRef s)
| _ ⇒ …
end.
```

On Figure 3.4, the translation is presented in a named setting. As a consequence, the introduction of new variables does not change references to existing ones and that's why $[\_]_0$ is the identity. In the De Bruijn setting of Template Coq, the translation has to take into account the shift induced by the duplication of the context. Therefore, the implementation `tsl_param`$_0$ of $[\_]_0$ is not the identity anymore. The argument `n` of `tsl_param`$_0$ represents the De Bruijn

level from which the variables have to be duplicated. There is no need for such an argument in $\texttt{tsl\_param}_1$, the implementation of $[\_]_1$, because in this function all variables are duplicated. The implemented cases include pattern matching. Fixpoints should be supported soon.
Given those two functions, we can yet translate some terms. For example, the translation of the type of polymorphic identity functions can be gotten by:

```
Definition ID := ∀ A, A → A.
Run TemplateProgram (Translate emptyTC "ID").
```

`emptyTC` is the empty translation context. This defines $\texttt{ID}^t$ to be:

```
λ f: ∀ A, A → A ⇒ ∀ A (A^t : A → Type) (x : A), A^t x → A^t (f A x)
```

We have also implemented the translation of inductives types. For instance, the translation of the equality type `eq` produces the following inductive:

```
Inductive eq^t A (A^t : A → Type) (x : A) (x^t : A^t x)
          : ∀ H, A^t H → x = H → Prop :=
          | eq_refl^t : eq^t A A^t x x^t x x^t eq_refl.
```

Then $[\texttt{eq}]_1$ is given by $\texttt{eq}^t$ and $[\texttt{eq\_refl}]_1$ by $\texttt{eq\_refl}^t$.

Given $\texttt{tsl\_param}_0$ and $\texttt{tsl\_param}_1$ the translation of the declaration of a block of mutual inductive types is not so hard to get. Indeed, such a declaration mainly consists of the arities of the inductives and of the types of constructors. And the aritiy and the types of the translated inductive are produced by translation of the original ones. We let the motivated reader refer to the implementation of the `tsl_mind_body`.

```
Definition tsl_mind_body (E : tsl_table) (mp : string) (kn : kername)
           (mind : mutual_inductive_body) : tsl_table * list
    mutual_inductive_body.
```

All put together, the translation is declared by:

```
Instance param : Translation :=
  {| tsl_id := λ id ⇒ id ++ "^t" ;
     tsl_tm := λ ΣE t ⇒ ret (tsl_param_1 (snd ΣE) t) ;
     tsl_ty := None ;
     tsl_ind := λ ΣE mp kn mind ⇒ ret (tsl_mind_body (snd ΣE) mp kn
    mind) |}.
```

For each constant c of type A, it is $[c]_1$ (of type $[A]_1\ [c]_0$) which is recorded in the translation table. There is no implementation of `tsl_ty` because there is no meaningful function $[\![\_]\!]$ for this presentation of parametricity.

3.5.1. Example

With this translation, the only commands that can be used are `Translate` and `TranslateRec`. Here is an illustration of their use coming from the work of Lasson on the automatic proofs of (ω-)groupoid laws using parametricity [Las14]. We show that all functions which have type $\prod (A : \mathcal{U})\,(x, y : A).\ x = y \to x = y$ are identity functions. Let `IDp` be this type:

$$
\begin{array}{rcl@{\qquad}rcl}
[x]_f & := & x & [\lambda\, x : A.\, M]_f & := & (\lambda\, x : [A]_f.\, [M]_f, \mathbf{true}) \\
[M\ N]_f & := & \pi_1([M]_f)\,[N]_f & [\Pi\, x : A.\, B]_f & := & (\Pi\, x : [A]_f.\, [B]_f) \times \mathbb{B}
\end{array}
$$

Figure 3.5.: Times bool translation on Π types, excerpt from Section 2.2.1

```
Definition IDp := ∀ A x y, x = y → x = y.
```

First we compute the translation of `IDp` using `TranslateRec`.

```
Run TemplateProgram (table ← TranslateRec emptyTC "IDp" ;
                     tmDefinition "table" table).
```

The second line define `table` to be the obtained translation table, so that we can reuse it later. Then we show that every parametric function on `IDp` is pointwise equal to the identity using the predicate $\lambda\, y.\, x = y$.

```
Lemma param_IDp (f : IDp) : IDpᵗ f → ∀ A x y p, f A x y p = p.
Proof.
  intros H A x y p. destruct p.
  destruct (H A (λ y ⇒ x = y) x eq_refl
              x eq_refl eq_refl (eq_reflᵗ _ _)).
  reflexivity.
Qed.
```

Let's define a function `myf` $:= p \mapsto p \cdot p^{-1} \cdot p$ and get its parametricity proof using the plugin.

```
Definition myf: IDp := λ A x y p ⇒ eq_trans (eq_trans p (eq_sym p)) p.
Run TemplateProgram (TranslateRec table "myf").
```

We reuse here `table` in which the translation of equality have been recorded. It is then possible to deduce automatically that $p \cdot p^{-1} \cdot p = p$ for all p:

```
Definition free_thm_myf : ∀ A x y p, myf A x y p = p
  := param_IDp myf myfᵗ.
```

3.6. Times bool on Π

The implementation of times bool translations is a bit trickier. We describe the one for the translation on Π types. It will give an example of the use of the command `Implement`. The translation is reminded in Figure 3.5. See Section 2.2.1 for a complete description.
Even if the translation is very simple, this time, going from the ideal world of $\lambda\Pi_\omega$ to the real world of Coq is not as simple as for parametricity. Indeed, when written in Coq, the translation is no longer fully syntax directed. Contrarily to pairs $(M,\ N)$ of $\lambda\Pi_\omega$, pairs are typed in Coq, meaning that the constructor of Σ types does not only have M and N as arguments but also their types:

```
pair : ∀ (A B : Type), A → B → A × B
```

Hence, in the case of lambdas in the definition of the translation, those types have to be provided:

```
[λ (x:A) ⇒ t] ≔ pair (∀ x:[A]. ?T) bool (λ (x:[A]) ⇒ [t]) true
```

`true` is always of type `bool`, but for the left term, we cannot recover the type `?T` from the source term. There is thus a mismatch between the lambdas which are not fully annotated and the pairs which are. There is a similar issue with applications and projections, but this one can be circumvented using Coq primitive projections which are untyped.
A solution is to use the type inference algorithm of Section 3.2 implemented on Template Coq terms to recover the missing information.

```
[λ (x:A) ⇒ t] ≔ let B ≔ infer Σ (Γ, x:[A]) t in
                  pair (∀ (x:[A]). B) bool (λ (x:[A]) ⇒ [t]) true
```

Here we need to have kept track of the global context Σ and of the local context Γ.
The translation function $[\_]_f$ is thus implemented by:

```
Fixpoint tsl_rec (fuel : nat) (Σ : global_context) (E : tsl_table)
                 (Γ : context) (t : term) {struct fuel}
  : tsl_result term ≔
  match fuel with
  | O ⇒ raise NotEnoughFuel
  | S fuel ⇒
  match t with
  | tRel  n ⇒ ret (tRel n)
  | tSort s ⇒ ret (tSort s)

  | tProd n A B ⇒ A' ← tsl_rec fuel Σ E Γ A ;
                  B' ← tsl_rec fuel Σ E (Γ , vass n A) B ;
                  ret (timesBool (tProd n A' B'))

  | tLambda n A t ⇒ A' ← tsl_rec fuel Σ E Γ A ;
                    t' ← tsl_rec fuel Σ E (Γ , vass n A) t ;
                    match infer Σ (Γ , vass n A) t with
                    | Checked B ⇒
                      B' ← tsl_rec fuel Σ E (Γ , vass n A) B ;
                      ret (pairTrue (tProd n A' B') (tLambda n A' t'))
                    | TypeError t ⇒ raise (TypingError t)
                    end

  ...
  end
  end.
```

We use some fuel because of the non-structural recursive call on `B` in the case of lambdas.

We also implemented the translation of some inductive types. For instance, the translation of the inductive `foo` generates the new inductive $\texttt{foo}^t$:

```
Inductive foo :=
| bar : (nat → foo) → foo.

Inductive foo^t :=
| bar^t : (nat^t → foo^t) × bool → foo^t
```

and the translation is extended by:

```
    [ foo ] = foo^t
    [ bar ] = (bar^t ; true)
```

3.6.1. Example

Let's demonstrate how to use the plugin to negate function extensionality.
The type of the axiom we will add to our theory is:

```
Definition NotFunext :=
  (∀ A B (f g : A → B), (∀ x:A, f x = g x) → f = g) → False.
```

We use `TranslateRec` to get the translation of `eq` and `False` and then we use `Implement` to inhabit the translation of the `NotFunext`:

```
Run TemplateProgram (TC ← TranslateRec emptyTC NotFunext ;
                     Implement TC "notFunext" NotFunext).
Next Obligation.
  tIntro H.
  tSpecialize H unit. tSpecialize H unit.
  tSpecialize H (λ x ⇒ x; true). tSpecialize H (λ x ⇒ x; false).
  tSpecialize H (λ x ⇒ eq_refl^t _ _; true).
  inversion H.
Defined.
```

The `Implement` command generates an obligation whose type is the translation of `NotFunext`, that is:

```
((∀ A, (∀ B, (∀ f : (A → B) × bool, (∀ g : (A → B) × bool,
  ((∀ x : A, eq^t B (π1 f x) (π1 g x)) × bool
       → eq^t ((A → B) × bool) f g)
   × bool) × bool) × bool) × bool) × bool
 → False^t) × bool
```

There is a lot of "`× bool`", that's why it is convenient that this type is automatically computed. We fill the obligation with the tactics `tIntro` and `tSpecialize` which are variants of `intro` and `specialize` dealing with the boolean:

```
Tactic Notation "tSpecialize" ident(H) uconstr(t)
  := apply π1 in H; specialize (H t).
Tactic Notation "tIntro" ident(H)
  := refine (λ H ⇒ _; true).
```

After the obligation being closed (and not before), an axiom `notFunext` of type `NotFunext` is declared in the current environment, as it would have been done by:

```
Axiom notFunext : NotFunext.
```

A constant $\texttt{notFunext}^{\mathrm{t}}$ whose body is the term provided in the obligation is also declared and the mapping $(\texttt{notFunext}, \texttt{notFunext}^{\mathrm{t}})$ is added in the translation table.
If the translation is correct, the consistency of Coq is preserved by the addition of this axiom. Let's remark that here it is not fully the case because Coq, our source theory, has η-conversion, which is incompatible with this translation.

3.7. Other uses

Here is a very quick review of some of our other implementations using the translation plugin. They all can be found in the repository `translations`.

Standard model We implemented the standard model as a "syntactic model" (see Section 2.5).
Let's have a look at a small example.

```
Definition toto := λ (f : ∀ A, A → A) ⇒ f Type.
Run TemplateProgram (Translate emptyTC "toto").
```

This defines $\texttt{toto}^{\mathrm{t}}$ to be the following term:

```
λ (γ : unit)
  (f : ∀ (A : Type) (H : π2 {| π1 := γ; π2 := A |}),
       π2 (π1 {| π1 := {| π1 := γ; π2 := A |}; π2 := H |})) ⇒
π2 {| π1 := γ; π2 := f |} Type
```

and whose type is:

```
unit → (∀ A : Type, A → A) → Type → Type
```

Other parametricity translations We implemented the cheap and the generous translations that we presented in Sections 2.4.2 and 2.4.3 and we used them to check the axiom preservation results.
For instance, we formalized that univalence is provable parametric, meaning that it is preserved by the cheap translation:

```
Theorem Ua_provably_parametric : ∀ h : Univalence, Univalence^t h.
```

Or, on the contrary, that the generous translation negates univalence:

```
Run TemplateProgram (ΣE ← TranslateRec emptyTC equiv ;
                     ΣE ← TranslateRec ΣE ∃ ;
                     Implement ΣE "notUnivalence"
                     (∃ A B, equiv A B × ∃ P, P A × (P B → False))).
```

Let's conclude this section by mentioning that Pujet and Annenkov are currently working on reimplementing the forcing plugin of Pédrot [P´] with the translation plugin. The forcing plugin implement the forcing translation [JLP+16].

4. Adventures in Cubical Type Theories

Up to here, we have encountered two interesting equalities: strict equality and univalent equality. Unfortunately, having both naively in a type theory causes them collapse and is thus inconsistent. However, there exists a way to have them cohabiting: the introduction of *fibrant types*. Fibrant types restrict the types on which the univalent equality can be eliminated and thus avoid the collapsing of the two equalities. Such a type theory with two equalities and a mechanism of fibrant types is called *Two Level Type Theory*. The first to introduce this setting was Voevodsky to reflect what happens in the simplicial model of Homotopy Type Theory.
This chapter is two fold: first, we want to introduce a two level type theory and give a model for it. Second, we take advantage of a type theory that we will discover on the way—*Interval Type Theory*—to prove a categorical result: we give a description of a model structure on the category of cubical sets.
A preliminary version of this chapter was available as a public draft: *Model structure on the universe in a two level type theory* (Simon Boulier, Nicolas Tabareau) [BT17]. It is now outdated and this chapter supersedes it.
This chapter owes much to long and numerous discussions with Huber, A. Lafont, Leena Subramaniam, Pujet and Tabareau.

4.1. Two Level Type Theory

Two Level Type Theory is a type theory with two equalities, a strict one and a univalent one, and a mechanism of fibrant types to make them cohabit. The story of Two Level Type Theory (TLTT) starts with Homotopy Type Theory which is a type theory with one equality enjoying univalence. HoTT turns out to be the fibrant fragment of Two Level Type Theory, meaning that the restriction to the fibrant types and to the univalent equality is HoTT. Thus, the univalent equality of TLTT has to be thought as the equality of HoTT.
The first model of Homotopy Type Theory was the simplicial model of Voevodsky. In this model a type is interpreted by a Kan simplicial set—the category of Kan simplicial sets is a model of well-behaved topological spaces—and univalent equality is interpreted by paths in those topological spaces. But there is another notion of equality in this model: the equality of the metatheory (which, for us, means an equality with FunExt and UIP). This one is reflected by the strict equality of TLTT. Voevodsky was the first to propose a type theory with those two equalities [Voe13] and called this type theory *Homotopy Type System*.
Later Altenkirch *et. al.* gave the first clear presentation of Two Level Type Theory and gave it this name [ACK16, Cap17]. They used it to get some results about Reedy Fibrant Diagrams and Complete Semi-Segal Types. They formalized some of them in Lean. There are a few

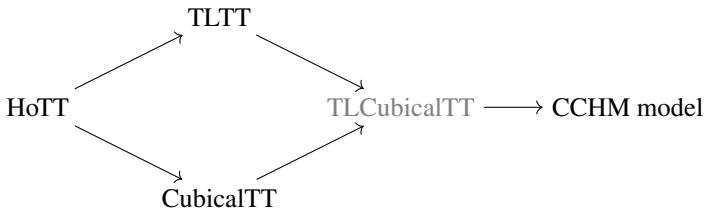


Figure 4.1.: Inclusions of Homotopy Type Theories. TLCubicalTT is hypothetical.

differences between their presentation and Homotopy Type System on which we will come back later. By Two Level Type Theory we will mean the paradigm which encompasses both presentations as well as the presentation that we will give in Section 4.1.1.

In parallel of this, Coquand *et. al.* developed another model for HoTT: the cubical model, in which a type is interpreted by a cubical set. There are two variants of this model, depending on the cube category chosen:

- for cubes without connections, it is called the BCH model (from the initials of its authors) [BCH14, BCH17]
- for cubes with connections, it is called the CCHM model [CCHM16, CHM18]

In those works, they not only developed a model of HoTT but also an intermediate language to carry syntactically some constructions of the model: Cubical Type Theory [Cub]. A distinctive feature of CubicalTT is the introduction of dimensions:

$$\frac{\Gamma \vdash}{\Gamma,\, i : \mathbb{I} \vdash} \qquad i \text{ is a dimension variable}$$

$\mathbb{I}$ is the *interval*, the "type of dimensions", but it is not possible to refer to it in the type theory. A dimension variable i can be specialized in two points: 0 and 1, the *endpoints* of the interval. In this type theory an equality between two points $x,\ y : A$ is a *path* between those two points. That is to say, a map $\mathbb{I} \to A$ whose endpoints are x and y:

$$\frac{\Gamma,\, i : \mathbb{I} \vdash t : A \qquad t\{i \coloneqq 0\} \simeq_\beta x \qquad t\{i \coloneqq 1\} \simeq_\beta y}{\Gamma \vdash \langle i \rangle.\, t : x \sim_A y}$$

The constructs of HoTT can be defined as "macros" in CubicalTT and HoTT can be embedded in CubicalTT.

Although it has not been spelled out, we could perfectly imagine "TLCubicalTT", a two level cubical type theory mixing both approaches. The intended inclusions between the mentioned systems are summed up in Figure 4.1. Those inclusions are "moral": if one would want to carry them in practice, the presentations of the involved type theories should probably be changed a

bit to fit well one with the other.

Last, Orton and Pitts developed a new abstraction layer for the CCHM model [OP17]. As Martin Löf Type Theory can be interpreted in any presheaf category (and even any topos), they identified nine axioms which are valid through the interpretation in the CCHM model and which are enough to carry several constructions of the model (basically all but the universe of fibrant types). So in the end, they use the type theoretic setting to *define* the CCHM model. This is very interesting because it makes the cubical model much more accessible to type theorists (including us!).
Technically, to define the CCHM model, Orton and Pitts define a category with families whose types are defined in their type theory and are afterward interpreted as fibrant presheaves (recovering the CCHM model). In the following we propose to explore an alternative way. We will consider the type theory that they define on its own, we will call it *Interval Type Theory* (𝕀TT), and define enough macros in it so that we can embed a modified version of Two Level Type Theory.

$$\mathbf{MLTT}_2^{\mathbf{F}} \longrightarrow \mathbb{I}\mathrm{TT} \longrightarrow \text{CCHM model}$$

In this type system, which we write $\mathbf{MLTT}_2^{\mathbf{F}}$, the notion of fibrancy does not apply to types anymore but to type families. In this way, we have a finer control on fibrancy. In our interpretation, fibrant type families won't always be interpreted as fully fibrant presheaf families but sometimes as *degenerately fibrant families*.
Our approach is not yet fully working and in particular the universe of fully fibrant types (the universe which is univalent) does not fit very well in our setting. However, we believe that it is worth the while to explore consequences of degenerate fibrancy. We recently got a confirmation of this because degenerate fibrancy and the degenerate fibrant replacement (that we will introduce in a next section) seem to take an important part in the definition in Higher Inductive Types [CHM18].

4.1.1. $\mathbf{MLTT}_2$

Let us introduce Two Level Type Theory. We will write $\mathbf{MLTT}_2$ for our presentation of TLTT. As previously said, in Two Level Type Theory there are both the strict equality $\equiv$ and the univalent one $=$.
As univalence and UIP are contradictory [Pro13, ex 3.1.19], TLTT requires a mechanism to prevent the strict equality and the univalent equality from collapsing. This is achieved by introducing the notion of *fibrant types*. Once again, it reflects what happens in the model: types along which one can transport are interpreted by fibrant simplicial sets (also called Kan simplicial sets).
In the end, $\mathbf{MLTT}_2$ consists of $\lambda\Pi_\omega + \equiv$ (MLTT with a strict equality) enriched with:
1. A new judgment $\Gamma \vdash A$ **Fib** which expresses that a type is *fibrant*.
All usual types are fibrant, except strict equality types. The rules to derive fibrancy are given in Figure 4.2. A non-fibrant type is called a *pretype*.
2. A univalent equality whose elimination is restricted to fibrant types. The typing rules are given in Figure 4.3. The restriction ensures that we have $t \equiv t' \to t = t'$ but never the

$$\frac{}{\Gamma \vdash \mathcal{U}_i \ \mathbf{Fib}} \qquad \frac{\Gamma \vdash A : \mathcal{UF}_i}{\Gamma \vdash A \ \mathbf{Fib}} \qquad \frac{\Gamma \vdash A \ \mathbf{Fib} \qquad \Gamma, x : A \vdash B \ \mathbf{Fib}}{\Gamma \vdash \prod x : A.\, B \ \mathbf{Fib}}$$

$$\frac{\Gamma \vdash A \ \mathbf{Fib} \qquad \Gamma, x : A \vdash B \ \mathbf{Fib}}{\Gamma \vdash \Sigma\, x : A.\, B \ \mathbf{Fib}}$$

Figure 4.2.: Rules for fibrancy in $\mathbf{MLTT_2}$

converse: $t = t' \twoheadrightarrow t \equiv t'$.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash t, t' : A \qquad \Gamma \vdash A \ \mathbf{Fib}}{\Gamma \vdash t =_A t' : \mathcal{U}_i} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \, \mathbf{Fib}}{\Gamma \vdash \mathbf{refl}_= t : t =_A t}$$

$$\frac{\Gamma \vdash A \ \mathbf{Fib} \qquad \Gamma \vdash t, t' : A}{\Gamma \vdash t =_A t' \ \mathbf{Fib}}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \ \mathbf{Fib} \qquad \Gamma \vdash t, t' : A \qquad \Gamma \vdash p : t =_A t' \\ \Gamma, y : A, q : t =_A y \vdash P \ \mathbf{Fib} \qquad \Gamma \vdash u : P\{y \coloneqq t, q \coloneqq \mathbf{refl}_= t\}\end{array}}{\Gamma \vdash J_=(y.q.P, p, u) : P\{y \coloneqq t', q \coloneqq p\}}$$

Figure 4.3.: Typing rules of the fibrant equality in $\mathbf{MLTT_2}$

3. A new hierarchy $\mathcal{UF}_0, \mathcal{UF}_1, \ldots$ of universes of fibrant types is also introduced:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash A \ \mathbf{Fib}}{\Gamma \vdash A : \mathcal{UF}_i} \qquad \frac{\Gamma \vdash A : \mathcal{UF}_i}{\Gamma \vdash A : \mathcal{U}_i} \qquad \frac{}{\Gamma \vdash \mathcal{UF}_i \ \mathbf{Fib}}$$

There are two main differences between the presentation of Voevodsky and the one of Altenkirch *et. al.* We chose to live between them.

1. Altenkirch *et. al.* introduce two versions of each positive inductive type. For instance, $\mathbb{N}^s$ living in the strict fragment and $\mathbb{N}$ living in the fibrant fragment. There is a map $\mathbb{N}^s \to \mathbb{N}$ but elimination in the other direction is restricted. The main advantage of adding this stratification is to get a conservativity result for the fibrant fragment over HoTT [Cap17]. We won't consider such a stratification.
2. Voevodsky considered in its system the reflection rule:

$$\frac{\Gamma \vdash e : t \equiv_A u}{\Gamma \vdash t \simeq_\beta u : A}$$

This is perfectly safe because it reflects what happens in the model: strict equality and conversion are interpreted in the same way (by the equality of the metatheory). However neither Altenkirch *et. al.* nor us will consider it because it makes type-checking undecidable. In light of the ETT to ITT translation (see Section 1.2.3), we know that HTS with the reflection rule is conservative over HTS without it, hence "we don't lose anything".
There is another reason why we do not consider the reflection rule. For the moment the presheaf model of type theory is formulated as a Category with Families (see Section 4.1.5) and it interprets the reflection rule (as the standard model). However, we hope that, at some point, it will be refined as a syntactic model. This time, it would not interpret the reflection rule anymore. A promising work in this direction is the forcing translation [JLP+16] which is a syntactic model and which is very close to a presheaf interpretation.

We found a way to emulate Two Level Type Theory in the Coq proof assistant, we describe it in Appendix A. Altenkirch *et. al.* used a similar technique to formalize their results in Lean.

4.1.2. Interval Type Theory

We now introduce Interval Type Theory, the type theory designed by Orton and Pitts [OP17] and intended to be an internal language for the CCHM model of Cubical Type Theory.
The presheaf model is a standard model of Martin Löf Type Theory (more on this in Section 4.1.5) and we have that the category of presheaves $\widehat{\mathcal{C}}$ over a category $\mathcal{C}$ is a category with families. The CCHM model is based on a presheaf model $\widehat{\square}$ for a particular cube category $\square$. In their article, Orton and Pitts spelled out nine axioms such that, if they are validated by the interpretation in $\widehat{\mathcal{C}}$, then, we can define a subcategory of $\widehat{\mathcal{C}}$ of fibrant presheaves. This subcategory is still a CwF and supports many type formers of Cubical Type Theory. The construction of this CwF is carried internally using Interval Type Theory as a syntactic language. Then, most of Cubical Type Theory can be interpreted in every presheaf category supporting those nine axioms. The only part which cannot be defined within the syntax of Interval Type Theory is the universe of fibrant types, because the type theory is not precise enough to carry this construction. This problem was treated in [LOPS18] but it requires to move to a more complicated setting that we won't detail here.
The nine axioms of Orton and Pitts are formulated in Martin Löf Type Theory enriched with an interval and a universe of cofibrant propositions. In the following we consider this type theory on its own and call it $\mathbb{I}$TT, for Interval Type Theory. In this type theory, we can *define* a univalent equality. It thus can be seen as an intermediate language to interpret $\mathbf{MLTT}_2$. Unfortunately, the universe of fibrant types does not fit in our setting. This is a weakness and it is not clear to us to which extent it invalidates our approach.

Interval Type Theory is Martin Löf Type Theory (let's say $\lambda\Pi_\omega + \eta$) with:

- a strict equality $\equiv$ enjoying FunExt and UIP
- an impredicative universe of propositions $\mathbb{P}$ enjoying Proof Irrelevance and Propositional Extensionality
- usual inductive types (Σ, $\mathbb{N}$, $\mathbb{1}$, ...)

- some quotient inductive types
- an interval $\mathbb{I}$
- a universe of cofibrant propositions **Cof**

The famous nine axioms are related to the interval and to the universe of cofibrant propositions, we detail them below.
As the strict equality enjoys UIP we will suppose that it is typed in $\mathbb{P}$:

$$\frac{\Gamma \vdash x,\ y : A}{\Gamma \vdash x \equiv_A y : \mathbb{P}}$$

In this way we stick to the Coq formalization (in Coq `eq` is typed in `Prop`) and we get UIP from proof irrelevance.
In the rest of this chapter we will often omit the contexts in inference rules. For instance, the above rule will be written:

$$\frac{\vdash x,\ y : A}{\vdash x \equiv_A y : \mathbb{P}}$$

Also, we will not deal with universe levels, we suppose that we have two universes $\mathbb{P}$ and $\mathcal{U}$ with:

$$\mathbb{P} : \mathcal{U} \qquad \mathbb{P} \subseteq \mathcal{U} \qquad \mathcal{U} : \mathcal{U}$$

Our work can be replayed in a type theory with a universe hierarchy.
Following Orton and Pitts approach we formalized our development about Interval Type Theory in Coq, see Section 4.1.3 for a description of the formalization. It can be found in folder `InternalCubical-Coq`. For each paragraph we give a direct link to the corresponding file.

Interval [`InternalCubical-Coq/Interval.v`]
The *interval* $\mathbb{I}$ is the first distinguishing feature of Interval Type Theory. It is a closed type which has two endpoints 0 and 1, and supports min and max operations. It is required to satisfy axioms given in Figure 4.4.

$$\vdash \mathbb{I} : \mathcal{U} \qquad \vdash 0,\ 1 : \mathbb{I} \qquad \vdash \sqcap : \mathbb{I} \to \mathbb{I} \to \mathbb{I} \qquad \vdash \sqcup : \mathbb{I} \to \mathbb{I} \to \mathbb{I}$$

The elements of the interval are called *dimensions* and having a dimension $i : \mathbb{I}$ in the context correspond to going one dimension up. An element of a type $a : A$ represent a point:

$$\bullet\, a$$

A function $p : \mathbb{I} \to A$ represents a line in A, also called a path, between the two points $p\,0$ and $p\,1$:

$$p\,0 \bullet\!\!\longrightarrow\!\!\bullet\ p\,1 \qquad \xrightarrow{\ i\ }$$

$$\mathbf{ax_1} \;:\; \prod P : \mathbb{I} \to \mathbb{P}.\ (\prod i : \mathbb{I}.\ P\ i \vee \neg(P\ i)) \to P\ 0 \equiv_{\mathbb{I}} P\ 1$$

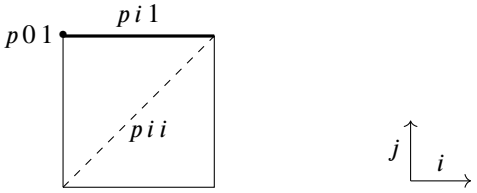
$$\mathbf{ax_2} \;:\; \neg\ 0 \equiv 1$$

$$\mathbf{ax_3} \;:\; \prod i : \mathbb{I}.\ (0 \sqcap i \equiv i \sqcap 0 \equiv 0) \;\wedge\; (1 \sqcap i \equiv i \sqcap 1 \equiv i)$$

$$\mathbf{ax_4} \;:\; \prod i : \mathbb{I}.\ (0 \sqcup i \equiv i \sqcup 0 \equiv i) \;\wedge\; (1 \sqcup i \equiv i \sqcup 1 \equiv 1)$$

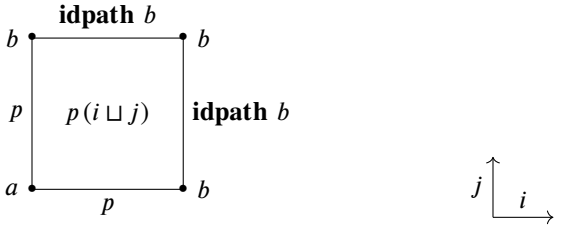
Figure 4.4.: Axioms satisfied by the interval $\mathbb{I}$

The constant function $\lambda\ \_ : \mathbb{I}.\ a$ is the constant path from a to a. It is written $\mathbf{idpath}\ a$.
A two variable function $p : \mathbb{I} \to \mathbb{I} \to A$ represents a square in A:



For instance, the up left point is $p\,0\,1$, the upper side is the line $\lambda\, i.\ p\, i\, 1$ and the upward diagonal is $\lambda\, i.\ p\, i\, i$. We can't represent the other diagonal because we don't have the negation.
A last example. If $p : \mathbb{I} \to A$ is a line between a and b:

Then $\lambda\, i,\ j.\ p\ (i \sqcup j)$ is the following square:



The axioms ensure that the interval is connected ($\mathbf{ax_1}$), that the two endpoints are distinct for strict equality ($\mathbf{ax_2}$) and that min and max give a *path algebra structure* to $\mathbb{I}$ ($\mathbf{ax_3}$ and $\mathbf{ax_4}$). A path algebra structure is like a bounded lattice but without associativity, commutativity and absorption law.
An important difference with the cubical model CCHM is that the interval *does not* have negation $\mathbf{not} : \mathbb{I} \to \mathbb{I}$. We could add it, restricting slightly the presheaf models in which we can interpret our theory. But here we stick to Orton and Pitts approach and do without negation.

$$\mathbf{ax_5} \ : \ \Pi\, i : \mathbb{I}.\ \mathbf{cof}\ (i \equiv 0) \wedge \mathbf{cof}\ (i \equiv 1)$$

$$\mathbf{ax_6} \ : \ \Pi\, \phi, \psi : \mathbb{P}.\ \mathbf{cof}\ \phi \to \mathbf{cof}\ \psi \to \mathbf{cof}\ (\phi \vee \psi)$$

$$\mathbf{ax_7} \ : \ \Pi\, \phi, \psi : \mathbb{P}.\ \mathbf{cof}\ \phi \to (\phi \to \mathbf{cof}\ \psi) \to \mathbf{cof}\ (\phi \wedge \psi)$$

$$\mathbf{ax_8} \ : \ \Pi\, \phi : \mathbb{I} \to \mathbb{P}.\ (\Pi\, i : \mathbb{I}.\ \mathbf{cof}\ (\phi\ i)) \to \mathbf{cof}\ (\Pi\, i : \mathbb{I}.\ \phi\ i)$$

$$\begin{aligned}\mathbf{ax_9} \ : \ & \Pi\, (\phi : \mathbf{Cof})\, (A : \phi \to \mathcal{U})\, (B : \mathcal{U})\, (s : \Pi\, u : \phi.\ A\ u \cong B).\\ & \Sigma\, (B' : \mathcal{U})\, (s' : B' \cong B)\, (e : \Pi\, u : \phi.\ A\ u \equiv B').\ \Pi\, u : \phi.\ e\ u\ \#\ s\ u \equiv s'\end{aligned}$$

$$\begin{aligned}\mathbf{join} \ : \ & \Pi\, (A : \mathcal{U})\, (\phi, \psi : \mathbf{Cof})\, (f : \phi \to A)\, (g : \psi \to A).\\ & (\Pi\, (u : \phi)\, (v : \psi).\ f\ u \equiv g\ v) \to \phi \vee \psi \to A\end{aligned}$$

$$\Pi\, u : \phi.\ \mathbf{join}(f, g)\ (\mathrm{inl}\ u) \equiv f\ u \qquad\qquad \Pi\, v : \psi.\ \mathbf{join}(f, g)\ (\mathrm{inr}\ v) \equiv g\ v$$

Figure 4.5.: Axioms satisfied by **Cof**

Cofibrant propositions [InternalCubical-Coq/Cof.v]
The other important feature of 𝕀TT is the universe of cofibrant propositions **Cof**. It is a subtype of $\mathbb{P}$ defined by a predicate **cof** asserting that a proposition is cofibrant, and it satisfies the axioms of Figure 4.5.

$$\mathbf{cof} \ : \ \mathbb{P} \to \mathbb{P} \qquad\qquad \mathbf{Cof} \ :\equiv \ \Sigma\, P : \mathbb{P}.\ \mathbf{cof}\ P$$

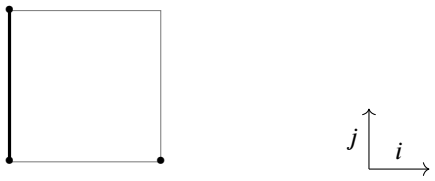
Cofibrant propositions represent formulas restricting the introduced dimensions. A typical cofibrant proposition is for instance:

$$(i \equiv 0) \vee ((i \equiv 1) \wedge (j \equiv 0))$$

We have seen that a function $p : \mathbb{I} \to \mathbb{I} \to A$ is a square. Now a function restricted by this formula

$$p' \ : \ \Pi\, i, j : \mathbb{I}.\ (i \equiv 0) \vee ((i \equiv 1) \wedge (j \equiv 0)) \to A$$

represents the following subsquare:



The axioms assert that **Cof** contains endpoint equality ($\mathbf{ax_5}$), is closed under disjunction, dependent conjunction and universal quantification over 𝕀 ($\mathbf{ax_6}$, $\mathbf{ax_7}$ and $\mathbf{ax_8}$). The last axiom is

a strictness axioms asserting that under some conditions, a type can be replaced by a strictly equivalent type satisfying additional strict equalities. It is a consequence of a stronger axiom, the decidability of **Cof**:

$$\Pi\, \phi : \mathbf{Cof}.\ \phi + \neg\phi$$

Decidability holds in the CCHM model but we don't assume it in 𝕀TT.
Axiom 5 together with propositional extensionality ensures that true and false proposition $\top$, $\bot$ are cofibrant. For ϕ and ψ in **Cof**, we will continue writing $\phi \wedge \psi$ for the element of **Cof** associated to the proposition $\pi_1(\phi) \wedge \pi_1(\psi)$. The same for $\phi \vee \psi$, $i \equiv 0$, $\bot$, … We will also blithely write ϕ instead of $\pi_1(\phi)$.
Last, the universe of cofibrant propositions allows forming the join of two partial elements. A partial element is a map $\phi \to A$ of domain $\phi : \mathbf{Cof}$. Given two cofibrant propositions $\phi, \psi : \mathbf{Cof}$ and two partial elements which agree where there are both defined:

$$f : \phi \to A \qquad g : \psi \to A \qquad \Pi\,(u : \phi)\,(v : \psi).\ f\ u \equiv g\ v$$

The join of f and g is a partial element

$$f \vee g : \phi \vee \psi \to A$$

such that $f \vee g\ (\mathrm{inl}\ u) \equiv f\ u$ and $f \vee g\ (\mathrm{inr}\ v) \equiv g\ v$. The existence of the join is also a consequence of the decidability of **Cof**.

Paths [InternalCubical-Coq/Paths.v]

Now that we have specified what is Interval Type Theory, we can carry the first constructions inside. A fundamental one is the construction of paths. A path between two points is a line with these points at its ends. We write $a \sim_A b$ the type of paths between a and b in A. Reflexivity is given by a constant path.

$$a \sim_A b \ :\equiv\ \Sigma\, p : \mathbb{I} \to A.\ p\ 0 \equiv_A a \wedge p\ 1 \equiv_A b$$

$$\begin{aligned}\mathbf{idpath}_A\ a\ &:\ a \sim_A a\\ &:\equiv\ (\lambda\_ : \mathbb{I}.\ a,\ \mathbf{refl}_{\equiv}\ a,\ \mathbf{refl}_{\equiv}\ a)\end{aligned}$$

Path types are not directly the types by which we will interpret the univalent equality because their transport does not compute strictly. Instead we will use identity types, which are built on top of them.
A type is contractible if it is inhabited and if all its elements are path-equal.

$$\mathbf{Contr}\ A \ :\equiv\ \Sigma\, a : A.\ \Pi\, x : A.\ a \sim_A x$$

The fact that paths are functions has nice consequences. For instance, function extensionality for paths becomes trivial: it is only a change in the order of the arguments of the path seen as a function. If p is a proof of type $\Pi\, x : A.\ f\ x \sim_B g\ x$ then $\lambda\, i.\ \lambda\, x : A.\ p\ x\ i$ is a path between f and g.

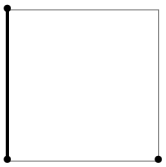
In the same spirit, paths enjoy contractibility of singletons. For every type A and $x : A$, the type of singletons in x is contractible:

$$\mathbf{Contr}\ (\Sigma\, y : A.\ x \sim_A y)$$

The center of contraction is $(x, \mathbf{idpath}_A\ x)$ and the path between $(x, \mathbf{idpath}_A\ x)$ and (y, p) is given by:

$$\lambda\, i.\ (p\ i, \lambda\, j.\ p\ (i \sqcap j))$$

Fibrancy [InternalCubical-Coq/Fibrations.v]
Now that we have Path types, we want to transport along them. Transport is only allowed along fibrant type families so we have to define fibrancy.
Let's first define the type of *partial elements* of a type $A : \mathcal{U}$. A partial element is an element of A only specified on a restricted face of the current context. For instance, if there are two dimensions $i, j : \mathbb{I}$ in the context, a partial element can be defined only on the subsquare given by $(i \equiv 0) \vee ((i \equiv 1) \wedge (j \equiv 0))$:



We write $\square A$ the type of partial elements of A:

$$\square A \coloneqq \Sigma\, \phi : \mathbf{Cof}.\ \phi \to A$$

Let $a : A$ be an element of A and (ϕ, u) be a partial element, we say that a *extends* (ϕ, u) if they coincide where u is defined:

$$(\phi, u) \nearrow a \coloneqq \Pi\, w : \phi.\ u\ w \equiv_A a$$

There is always the empty inhabitant of $\square A$:

$$(\bot, \bot\mathbf{-elim}) : \square A$$

It is the partial element of A which is specified nowhere. Every partial element of A turns out to be path-equal to it, hence the type $\square\, A$ is always contractible:

$$\mathbf{Contr}\ (\square A)$$

A type has an *extension structure* if all its partial elements can be extended:

$$\mathbf{Ext}\ A \coloneqq \Pi\, a : \square A.\ \Sigma\, a_1 : A.\ a \nearrow a_1$$

Fibrancy is a weaker condition than having an extension structure where the partial element is required to be specified in at least one endpoint.

To overcome the lack of negation, Orton and Pitts parametrize their definition of fibrancy by the direction in which the extension is done. To do this we introduce the inductive type $\mathtt{OI}$ which has two elements $\mathtt{O}$ and $\mathtt{I}$.

$$\mathtt{OI} \coloneqq \mathtt{O} \mid \mathtt{I}$$

And we define $\iota : \mathtt{OI} \to \mathbb{I}$ the conversion in the interval to the corresponding endpoint by $\iota\,\mathtt{O} \coloneqq 0$ and $\iota\,\mathtt{I} \coloneqq 1$. In the following, we will suppose that this coercion is always implicitly inserted where needed. We will also use the negation $! : \mathtt{OI} \to \mathtt{OI}$ which is defined by:

$$!\,\mathtt{O} \coloneqq \mathtt{I}$$
$$!\,\mathtt{I} \coloneqq \mathtt{O}$$

A type $A : \mathcal{U}$ is said to be *fibrant*, if the following type is inhabited:

$$\mathbf{Fib}\ A \coloneqq \Pi\,(e : \mathtt{OI})\,(\phi : \mathbf{Cof})\,(a : \Pi\,i : \mathbb{I}.\ \phi \vee i \equiv e \to A).\ \Sigma\,a_1 : A.\ (\phi,\ a\ !e \circ \mathrm{inl}) \nearrow a_1$$

It means that every partial element specified at least in $i \equiv e$ can be extended.
This type is not exactly the one proposed by Orton and Pitts but is strictly equivalent. For reference, they use:

$$\Pi\,(e : \mathtt{OI})\,(\phi : \mathbf{Cof})\,(a : \mathbb{I} \to \phi \to A).\ (\Sigma\,a_0 : A.\ (\phi,\ a\ e) \nearrow a_0) \to (\Sigma\,a_1 : A.\ (\phi,\ a\ !e) \nearrow a_1)$$

Our variant is more compact and will allow defining the fibrant replacement in a simpler way. Fibrancy is weaker than having an extension structure:

$$\mathbf{Ext}\ A \to \mathbf{Fib}\ A$$

In fact, it turns out that a type has an extension structure if and only if it is fibrant and contractible:

$$\mathbf{Ext}\ A \leftrightarrow \mathbf{Fib}\ A \times \mathbf{Contr}\ A$$

It is not a strict equivalence a priori.

Fibrancy generalizes to type families in the following way.

Definition 1. A type family $P : A \to \mathcal{U}$ is said to be *regularly fibrant* if the following type is inhabited:

$$\mathbf{RFib}\ P \coloneqq \Pi\,(e : \mathtt{OI})\,(a : \mathbb{I} \to A)\,(\phi : \mathbf{Cof})\,(p : \Pi\,i : \mathbb{I}.\ \phi \vee i \equiv e \to P\ (a\ i)).$$
$$\Sigma\,p_1 : P\ (a\ !e).\ (\phi,\ p\ !e \circ \mathrm{inl}) \nearrow p_1$$

The operation to which the fibrancy gives access is called *composition*. If $P : A \to \mathcal{U}$ is regularly fibrant we write:

$$\mathbf{comp}_P : \Pi\,(e : \mathtt{OI})\,(a : \mathbb{I} \to A)\,(\phi : \mathbf{Cof})\,(p : \Pi\,i : \mathbb{I}.\ \phi \vee i \equiv e \to P\ (a\ i)).\ P\ (a\ !e)$$

And for all $w : \phi$ we have that:

$$\mathbf{comp}_P\ e\ a\ \phi\ p \equiv p\ !e\ (\mathrm{inl}\ w)$$

A type A amounts to be fibrant if and only if the constant family over unit $\lambda\_ : \mathbb{1}.\ A$ is regularly fibrant.

For a type family $P : A \to \mathcal{U}$, it is **not** equivalent to be regularly fibrant and that for every x, $P\,x$ is fibrant. The later is weaker than the former. We will say that a type family is *degenerately fibrant* or *pointwise fibrant* if we only have the weaker condition:

$$\mathbf{DFib}\ P \coloneqq \Pi\, x : X.\ \mathbf{Fib}\ (P\ x)$$

For a two variables type family $P : \Pi\, x : X.\ A\ x \to \mathcal{U}$ we will write $\mathbf{RFib_2}\ P$ if the type family over the sigma is regularly fibrant:

$$\mathbf{RFib_2}\ P \coloneqq \mathbf{RFib}\ (\lambda\, z : \Sigma\, x : X.\ A\ x.\ P\ \pi_1(z)\ \pi_2(z))$$

A useful remark is that regular fibrancy is stable under precomposition:

$$\mathbf{RFib}\ P \to \mathbf{RFib}\ (P \circ f)$$

Orton and Pitts proved the following propositions, which interpret the fibrancy rules of Two Level Type Theory.

Proposition 28.

$$\mathbf{Fib}\ \mathbb{1} \qquad \text{and} \qquad \mathbf{Fib}\ \mathbb{N}$$

Proposition 29. For $A : X \to \mathcal{U}$ and $B : \Pi\, x : X.\ A\ x \to \mathcal{U}$, if $\mathbf{RFib}\ A$ and $\mathbf{RFib_2}\ B$ then

$$\mathbf{RFib}\ (\lambda\, x : X.\ \Pi\, a : A\ x.\ B\ x\ a)$$

$$\mathbf{RFib}\ (\lambda\, x : X.\ \Sigma\, a : A\ x.\ B\ x\ a)$$

Proposition 30. For $A : X \to \mathcal{U}$, if $\mathbf{RFib}\ A$ then $\_ \sim_A \_ : \Pi\, x : X.\ A\ x \to A\ x \to \mathcal{U}$ is regularly fibrant as a three variables type family. That is to say:

$$\mathbf{RFib\ Paths}_A$$

where $\mathbf{Paths}_A : \Sigma\, x : X.\ A\ x \times A\ x \to \mathcal{U} \coloneqq \lambda\,(x,\ (a,\ b)).\ a \sim_{A\,x} b$

See Orton and Pitts' article for the proofs.
We also have that the universe of pretypes is fibrant. In fact, it even has an extension structure. This is quite surprising because it means that for all types A and B there is a path connecting A and B:

$$A \sim_{\mathcal{U}} B$$

However, this won't be enough to transport along it and get a map $A \to B$ because the type family $\lambda\, X : \mathcal{U}.\ X$ is not regularly fibrant.

Proposition 31. The universe has an extension structure, and hence is fibrant:

$$\mathbf{Ext}\ \mathcal{U}$$

Proof. Let $A : \phi \to \mathcal{U}$ be a partial type (*i.e.* a partial element of $\mathcal{U}$). We have to give $B' : \mathcal{U}$ such that for all $w : \phi$, $A\ w \equiv B'$.
B' is given by the strictness axiom in A. There are two choices for the B appearing in the axiom: either $\Pi\, w : \phi.\ A\ w$ or $\Sigma\, w : \phi.\ A\ w$, both work. □

From the composition operation we can get a richer operation: *Kan filling*.

Proposition 32. If $P : A \to \mathcal{U}$ is regularly fibrant then we can define an operation $\mathbf{fill}_P$

$$\mathbf{fill}_P : \Pi\,(e : \mathbb{O}\mathbb{I})\,(a : \mathbb{I} \to A)\,(\phi : \mathbf{Cof})\,(p : \Pi\, i : \mathbb{I}.\ \phi \vee i \equiv e \to P\ (a\ i)).\ \Pi\, i : \mathbb{I}.\ P\ (a\ i)$$

such that for all e, ϕ, a and p **fill** agrees with p on $\phi \vee i \equiv e$ and with **comp** in e:

$$\Pi\,(i : \mathbb{I})\,(w : \phi \vee i \equiv e).\ \mathbf{fill}_P\ e\ a\ \phi\ p\ i\ \equiv\ p\ i\ w$$

$$\mathbf{fill}_P\ e\ a\ \phi\ p\ !e\ \equiv\ \mathbf{comp}_P\ e\ a\ \phi\ p$$

This is a notable feature of the CCHM model that Kan filling can be defined from composition, it is not the case in BCH for instance. Again, see Orton and Pitts' article for the proof.

Transport [InternalCubical-Coq/Paths.v]

Now that we have defined fibrancy we can at last define transport along a path.

Proposition 33. Let $P : A \to \mathcal{U}$ be a **regularly fibrant** type family, $p : a \sim_A b$ a path between two points of A and u of type $P\ a$. Then u can be transported along p to get an element of type $P\ b$:

$$\mathbf{transport}(P,\ p,\ u)\ :\ P\ b$$

Proof. The transport is given by:

$$\mathbf{comp}_P\ \mathbb{O}\ \bot\ \bot\text{−}\mathbf{elim}\ u$$

modulo rewriting by strict equalities. □

Transport together with contractibility of singletons give the dependent elimination for Path types (still restricted to regularly fibrant families).
However paths have a great defect, the transport does not compute on reflexivity. For all $a : A$ and $u : P\ a$ we have a path

$$\mathbf{transport}(P,\ \mathbf{idpath}_A\ a,\ u)\ \sim_{P\,a}\ u$$

but this equality does **not** hold strictly a priori. To remedy this, we will use identity types.

With the transport and the dependent elimination we can recover all the groupoidal laws satisfied by a univalent equality. First of them are inverse and concatenation:

$$\_^{-1}\ :\ x \sim_A y \to y \sim_A x$$

$$\_ \cdot \_\ :\ x \sim_A y \to y \sim_A z \to x \sim_A z$$

There are some operations which are proven with a transport in HoTT setting but which does not need the fibrancy structure in the cubical setting. They can be defined directly using the cubical definition of paths. For instance:

$$\mathbf{ap}_f\ :\ x \sim_A y \to f\ x \sim_B f\ y$$

$$\mathbf{ap10}_x\ :\ f \sim_{A\to B} g \to f\ x \sim_B g\ x$$

To a path p the first associates the path $\lambda\, i.\ f\ (p\ i)$ and the second the path $\lambda\, i.\ (p\ i)\ x$. It is the same phenomenon that for the definition of function extensionality: as Path type is a richer type than in HoTT, some definitions are simpler.

Identity Types [InternalCubical-Coq/Id.v]
Identity types are a modified version of Path types to recover the strict equality for the transport of reflexivity. They were introduced by Swan [Swa16] and reused in the CCHM model [CCHM16]. An element of an identity type is a path together with a cofibrant proposition indicating "where the path is refl". We will note $x =_A y$ the identity types.

$$x =_A y \coloneqq \Sigma\,(p : x \sim_A y)\,(\phi : \mathbf{Cof}).\ \phi \to \Pi\, i : \mathbb{I}.\ p\ i \equiv_A x$$

Reflexivity is given by the constant path with the proposition "true":

$$\begin{aligned}\mathbf{refl}_{=}\ x\ &:\ x =_A x\\ &\coloneqq (\mathbf{idpath}_A\ x,\ \top,\ \lambda\, w\, i.\ \mathbf{refl}_{\equiv}\ x)\end{aligned}$$

Identity types and paths are logically equivalent. From identity types to paths there is a "forget" map, and in the other direction we use the proposition "false".

$$\begin{aligned}\mathbf{id2paths}\ &:\ x =_A y\ \to\ x \sim_A y\\ &\coloneqq \lambda\,(p,\ \_,\ \_).\ p\\ \mathbf{paths2id}\ &:\ x \sim_A y\ \to\ x =_A y\\ &\coloneqq \lambda\, p.\ (p,\ \bot,\ \bot\text{-}\mathbf{elim})\end{aligned}$$

id2paths sends "refl to refl" (*i.e.* $\mathbf{refl}_{=}$ to **idpath**), but **paths2id** does **not** send **idpath** to $\mathbf{refl}_{=}$.
Given the extra proposition ϕ asserting where the path is refl, it is possible to define a dependent eliminator for identity types which "computes" strictly on $\mathbf{refl}_{=}$.

$$\frac{\begin{array}{c}\vdash P : \Pi\, y : A.\ t =_A y \to \mathcal{U}\\ \vdash H : \mathbf{RFib_2}\ P \qquad \vdash p : t =_A t' \qquad \vdash u : P\ t\ (\mathbf{refl}_{=}\ t)\end{array}}{\vdash J_{=}(P,\ H,\ p,\ u) : P\ t'\ u}$$

$$J_{=}(P,\ H,\ \mathbf{refl}_{=}\ t,\ u) \equiv u$$

To conclude this section, let us remark that the CCHM model justifies some rules which are a bit stronger than what we gave for $\mathbf{MLTT_2}$. In the rules of identity types Figure 4.3 (formation, introduction and elimination rules), there is no need for the fibrancy condition on A and only the condition on P in the elimination rule is enough. A priori they can be needed for other models such as the simplicial model or the BCH model.

4.1.3. Formalization in Coq

Interval Type Theory can be simulated in Coq. We use:

- The equality `eq` of Coq for the strict equality. It is typed in `Prop`.
- The `Prop` universe for $\mathbb{P}$. Proof irrelevance and propositional extensionality are postulated as axioms.

```
Axioms (I : Type)
       (zero one : I)
       (zero_one : ~ zero = one)
       (inf sup : I → I → I)
       (connected : ∀ (P : I → Prop), (∀ (i : I), P i \/ ~(P i))
                                    → P zero = P one).
Notation "0" := zero.
Notation "1" := one.
Axioms (zero_inf : ∀ {i}, inf 0 i = 0)
       (one_inf : ∀ {i}, inf 1 i = i)
       (inf_zero : ∀ {i}, inf i 0 = 0)
       ...
```

Figure 4.6.: Begining of the definition of the interval in Coq

- The usual inductive types of Coq.
- Quotient inductive types are implemented with a private inductive type. This is the way Higher Inductive Types are usually implemented in Coq.
- The interval and the universe of cofibrant propositions are postulated as axioms. And the nine axioms are postulated too. See Figure 4.6.

We essentially replayed Orton and Pitts' Agda formalization [OP18] in Coq. Hence, it will be interesting mainly for the next section about the model structure on the universe.
We formalized all the constructions of the previous parts except for the fibrancy rules of Π, Σ and Id types (which were formalized by Orton and Pitts). A small example of what our formalization looks like is given in Figure 4.7.
The formalization in Coq goes pretty well, the only painful thing is to rewrite all strict equalities by hand. The advantage of Coq over Agda is that we can use the tactics. On the other hand, Orton and Pitts used some "rewrite rules" to add the equalities of the path algebra structure (like "$i \vee 0 \equiv i$" ...) as definitional. This simplifies a bit the proofs by avoiding some strict rewritings.

4.1.4. Modified Two Level Type Theory

We propose to interpret $\mathbf{MLTT}_2^{\mathbf{F}}$, a modified version of Two Level Type Theory, in Interval Type Theory. The constructions defined above (fibrancy and identity types) are used to interpret fibrancy rules and the univalent equality.
We do not detail $\mathbf{MLTT}_2^{\mathbf{F}}$ here and only sketch what it would look like. In this type theory, the primitive notion of fibrancy would not be attached to type anymore but to type families. Hence, there would be a judgment

$$\Gamma \vdash x : A.\ P\ \mathbf{RFib}$$

```
Definition Itransport {A} (P : A → Type) {H : RFib P} {x y}
                      (p : Id A x y) (u : P x) : P y.
Proof.
  destruct p as [p φ].
  unshelve refine (let XX := H 0' (λ i ⇒ p i) φ.1 _ _ in _).
  - intros i f; cbn. refine (_ E# u).
    symmetry; now apply φ.2.
  - unshelve econstructor. refine (_ E# u).
    exact p.2.1^E.
    intro w; cbn in *.
    eapply Eap10, Eap, uip.
  - cbn in XX.
    refine (_ E# XX.1).
    exact p.2.2.
Defined.
```

Figure 4.7.: Excerpt of the formalization: transport for identity types

asserting that a type family is regularly fibrant.
$\vdash A$ **Fib** would be a notation for $\vdash \_ : \mathbb{1}.\ A$ **RFib**.
The fibrancy rules would look like:

$$\frac{\Gamma \vdash x : X.\ A\ \textbf{RFib} \qquad \Gamma \vdash (x,\ a) : \Sigma\, x : X.\ A.\ B\ \textbf{RFib}}{\Gamma \vdash x : X.\ \Pi\, a : A.\ B\ \textbf{RFib}}$$

and the elimination rule of univalent equality would be guarded by a regular fibrancy condition $\Gamma \vdash x : A.\ P$ **RFib**. In $\mathbf{MLTT}_2^{\mathbf{F}}$, a degenerate fibrant replacement is admissible as we will see in the next section.

However our interpretation suffers from several defects.
1. The major defect of our system is that the univalent universe of the cubical model ($\mathcal{UF}$ in $\mathbf{MLTT}_2$) does not fit very well in our setting. We can add it with the following rules:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathcal{UF}} \qquad \frac{\Gamma \vdash A : \mathcal{UF}}{\Gamma \vdash A : \mathcal{U}} \qquad \frac{\Gamma,\ x : A \vdash P : \mathcal{UF}}{\Gamma \vdash x : A.\ P\ \textbf{RFib}} \qquad \frac{\cdot \vdash x : A.\ P\ \textbf{RFib}}{x : A \vdash P : \mathcal{UF}}$$

But the rules do not seem very natural. In particular the right one, which makes appear an empty context.
2. In our setting, fibrancy is a structure and not a property: its inhabitants are meaningful. In the formalization, we managed to avoid all situations where we have two different derivations of the same fibrancy judgment but we wonder if it can always be the case. In the current state, it is misleading to "hide" the proof of fibrancy which has to be in the arguments of transport and $J_=$.
3. Last, if we don't consider the reflection rule, we get very few computational rules. Even if all desired equalities hold for the strict equality, this is not as computational as it should be.

4.1.5. The presheaf model

To complete the presentation of Interval Type Theory, we discuss a bit its intended model (CCHM) which is a presheaf model over a cube category.
The presheaf model of dependent type theory is a refinement of the standard model. Instead of being interpreted by a type, a context is interpreted by a presheaf which is a type parametrized by the objects of a category $\mathcal{C}$. Presheaves are used to model notions such as "variation over time", "resource availability" and, in our case, "dimension".
The first account of this model goes back to Hofmann [Hof97] although it seems to have been known before. A nice and recent presentation is also given in Huber's thesis [Hub]. Traditionally, and it is the case in those two references, the presheaf model is given in a set theoretic setting: definitions of categories, functors, etc. are given in set theory. However, nothing prevents us from presenting the presheaf model in type theory with a strict equality. And indeed one can show that $\widehat{\mathcal{C}}$, the category of presheaves over a category $\mathcal{C}$, is a category with families and hence a model of dependent type theory (see Section 1.5). Kaposi formalized in Agda that $\widehat{\mathcal{C}}$ forms a CwF supporting Π types [Kap][1].
The CCHM model is thus a presheaf model over the cube category $\square$. The goal of this section is mainly to see how degenerate fibrancy is interpreted in the category of cubical sets. We will only give the basic definitions of the presheaf model and of the interpretation. In particular, we won't redefine the cube category $\square$, nor the interpretation of the interval $\mathbb{I}$ or the face lattice $\mathbb{F}$. We let the reader refer to [CCHM16] for them. Also, for simplicity, we will suppose that the universe of cofibrant propositions **Cof** is interpreted by the face lattice $\mathbb{F}$, instead of being defined from a predicate "being cofibrant".

Let's recall the rudiments of presheaf models, they start with categories. Defining the right notion of category in HoTT with a relevant equality is quite intricate as several choices can be made to tame higher coherences. There is no such shilly-shallying in a type theory where the equality is irrelevant, as already noticed in [ACK16]. In fact, everything takes place exactly as in set theory.

Definition 2. A *category* consists of:

- a type $\mathcal{C}$ of objects,
- for all $I, J : \mathcal{C}$, a type $\mathrm{Hom}(I, J)$ of arrows
- for all $I : \mathcal{C}$, an identity arrow $\mathbf{id}_A : \mathrm{Hom}(I, I)$
- for all $I, J, K : \mathcal{C}$, a composition function
  $\_\circ\_ : \mathrm{Hom}(J, K) \to \mathrm{Hom}(I, J) \to \mathrm{Hom}(I, K)$
- for all $f : \mathrm{Hom}(I, J)$, a proof of $f \circ \mathbf{id}_I \equiv f$ and $\mathbf{id}_J \circ f \equiv f$
- for all $f : \mathrm{Hom}(I, J)$, $g : \mathrm{Hom}(J, K), h : \mathrm{Hom}(K, L)$, a proof of
  $h \circ (g \circ f) \equiv (h \circ g) \circ f$

We often write $\mathcal{C}$ both for the category and the type of its objects. In the following the composition of arrows $g \circ f$ is simply written gf.

[1] https://bitbucket.org/akaposi/tt-in-tt/src/378848c2886fa007579b1a1479bbe1b601440e47/PSh/?at=master

With this definition, each universe $\mathcal{U}$ forms a category where $\mathrm{Hom}(A, B)$ is given by $A \to B$. Identity and composition are those of functions and the laws are given by $\beta\eta$-conversion. We write **Type** this category, it plays the same role as the category **Set** in set theory.

A presheaf over $\mathcal{C}$ is a functor from $\mathcal{C}^{\mathrm{op}}$ to **Type**. They interpret contexts.

Definition 3. A *presheaf* over the category $\mathcal{C}$ is given by:
- $\Gamma : \mathcal{C} \to \mathcal{U}$ a type family over the objects of $\mathcal{C}$
- some restriction maps $\Pi\, I,\, J : \mathcal{C}.\, \mathrm{Hom}(J, I) \to \Gamma(I) \to \Gamma(J)$
- respecting identity $\Pi\,(I : \mathcal{C})\,(\rho : \Gamma(I)).\, \Gamma_{\mathbf{id}_I}\ \rho \equiv \rho$
- and respecting composition:
  $\Pi\,(f : \mathrm{Hom}(J, I))\,(g : \mathrm{Hom}(K, J))\,(\rho : \Gamma(I)).\, \Gamma_{f \circ g}\ \rho \equiv \Gamma_g\ (\Gamma_f\ \rho)$

We write Γ both for the presheaf and the associated type family. We write Γ_f the restriction map induced by an arrow f, and ρf its action on an element $\rho : \Gamma(I)$. Last we write $\Gamma \mathrel{\hat{\vdash}}$ to indicate that Γ is a presheaf.

The presheaf families are "dependent" presheaves, varying over another presheaf Γ. They interpret type families.

Definition 4. Let $\Gamma \mathrel{\hat{\vdash}}$ be a presheaf. A *presheaf family* over Γ is given by:
- a type family $A : \Pi\,(I : \mathcal{C})\,(\rho : \Gamma(I)).\, \mathcal{U}$
- some restriction maps $\Pi\,(I,\, J : \mathcal{C})\,(f : \mathrm{Hom}(J, I))\,(\rho : \Gamma(I)).\, A\rho \to A(\rho f)$
- respecting identity $\Pi\,(I : \mathcal{C})\,(\rho : \Gamma(I))\,(x : A\rho).\, A_{\mathbf{id}_I}\ x \equiv x$
- and respecting composition:
  $\Pi\,(f : \mathrm{Hom}(J, I))\,(g : \mathrm{Hom}(K, J))\,(\rho : \Gamma(I))\,x.\, A_{f \circ g}\ x \equiv A_g\ (A_f\ x)$

We write A both for the type family and the presheaf family. We write $A\rho$ the type family taken in $(I,\, \rho)$ and A_f the restriction map induced by an arrow f. Its action on an element $x : A\rho$ is again written xf. We write $\Gamma \mathrel{\hat{\vdash}} A$ to indicate that A is a presheaf family over Γ.

Last, we define the sections of a presheaf family, they interpret terms.

Definition 5. Let $\Gamma \mathrel{\hat{\vdash}} A$ be a presheaf family. A *section* of A consists of:
- a section of the type family $t : \Pi\,(I : \mathcal{C})\,(\rho : \Gamma(I)).\, A\rho$
- which is invariant by restrictions $\Pi\,(f : \mathrm{Hom}(J, I))\,(\rho : \Gamma(I)).\, (t\rho)f \equiv t(\rho f)$

We write t both for the section of the type family and the section of the presheaf family. We write $t\rho$ the term taken in $(I,\, \rho)$. Last we write $\Gamma \mathrel{\hat{\vdash}} t : A$ to indicate that t is a section of the presheaf family $\Gamma \mathrel{\hat{\vdash}} A$.

The presheaves over $\mathcal{C}$ form a category written $\widehat{\mathcal{C}}$. This category can be made into a category with families using presheaf families and sections. The empty context is interpreted by $\mathbb{1}$ the constant presheaf equal to unit, and context extension is interpreted by Σ types. Then, a

presheaf family in the empty context is then the same thing as a presheaf[2]. *C.f.* the mentioned references for further details.
The presheaf model supports Π types and Σ types [Hof97]. It also supports universes. This construction is due to Hofmann and Streicher [HS97]. Let $\mathcal{U}_i$ be a universe included in the universe $\mathcal{U}$ of the definition of presheaves (in the sense that we have $\mathcal{U}_i : \mathcal{U}$). Then $\mathcal{U}_i$ can be interpreted in $\widehat{\mathcal{C}}$ as the following presheaf (it is a closed type):

$$\begin{aligned}\mathcal{U}_i(I) \coloneqq\ & \Sigma(A : \Pi(J : \mathcal{C})(f : \mathrm{Hom}(J, I)).\, \mathcal{U}_i) \\ & (\alpha : \Pi(f : \mathrm{Hom}(J, I))(g : \mathrm{Hom}(K, J)).\, A_f \to A_{fg}) \\ & \Pi I,\, x.\, x\,\mathbf{id}_I \equiv x \times \Pi f,\, g,\, h,\, x.\, xgh \equiv (xg)h\end{aligned}$$

where the action of $\alpha\, f\, g$ on $x : A_f$ is once again written xg. The restriction maps are given by postcomposition.
Hence, if the metatheory has $n + 1$ universes, n universes can be built in the presheaf model (exactly as for the standard model). Let's remark that the interpretation of $\mathbb{P}$ seen as a universe is:

$$\begin{aligned}\mathbb{P}(I) \coloneqq\ & \Sigma(A : \Pi(J : \mathcal{C})(f : \mathrm{Hom}(J, I)).\, \mathbb{P}) \\ & (\alpha : \Pi(f : \mathrm{Hom}(J, I))(g : \mathrm{Hom}(K, J)).\, A_f \to A_{fg}) \\ & \ldots \text{ (always true by proof irrelevance) } \ldots\end{aligned}$$

This presheaf amounts to be the presheaf of *sieves* which is the subobject classifier in a presheaf category (see [MM12]).
To interpret Interval Type Theory it remains to check that the presheaf model preserves Function Extensionality, Propositional Extensionality, Proof Irrelevance and Quotient Inductive Types. It is true for the three axioms. It should also be the case for QITs but we did not do the formal verification.

Remark. The requirement of having Propositional Extensionality in the metatheory is very strong. It seems that the use of this axiom could be avoided by working with the face lattice $\mathbb{F}$ of CCHM instead of the cofibrancy predicate **cof** (the definition $\mathbf{Cof} \coloneqq \Sigma P : \mathbb{P}.\, \mathbf{cof}\ P$ has really a set theoretic flavor). As we said, it is in fact the assumption that we make in this section.

𝕀TT as an internal language

Interval Type Theory can be used as a subset of the internal language of $\widehat{\mathcal{C}}$, meaning that the syntax of 𝕀TT can be used to carry constructions in $\widehat{\mathcal{C}}$. Indeed, in the presheaf model, no one interpretation of rule makes the assumption that the premises of the rule are "coming from the syntax". For instance, the interpretation of the formation rule of Π types can be carried for arbitrary presheaf families $\Gamma \mathrel{\hat{\vdash}} A$ and $\Gamma, A \mathrel{\hat{\vdash}} B$, even if they are not the interpretation of a type of 𝕀TT :

$$\frac{\Gamma \mathrel{\hat{\vdash}} A \qquad \Gamma, A \mathrel{\hat{\vdash}} B}{\Gamma \mathrel{\hat{\vdash}} \Pi\, A\, B}$$

[2] The two types are isomorphic provided η on unit.

Hence, for any arbitrary presheaf family $\Gamma \hat{\vdash} A$, we can form the presheaf family $\Gamma \hat{\vdash} \mathbf{Fib}\ A$ by unfolding the definition of the **Fib** type former. Taking the "proposition-as-types" point of view, we say that the proposition **Fib** A *holds* if the presheaf family $\Gamma \hat{\vdash} \mathbf{Fib}\ A$ has a section.

Degenerate Fibrancy

With this "internal language" point of view, **Fib** induces a notion of fibrancy on all presheaf families. In this paragraph, we compare this notion of fibrancy with the one introduced in the CCHM model.
Even if we don't want to detail here the definition of the cube category $\square$, we need to know a little about it. Its objects are written $I, J, \ldots$ and are morally finite sets of dimensions. For each object I there exists a fresh dimension not in I written $i \# I$ giving rise to an object I, i which is "one dimension up". There are two arrows $I \to I, i$ written $(i = 0), (i = 1)$ and called *face maps*. And in the other direction, there is an arrow $s_I : I, i \to I$ called a *degeneracy*. We have $s_I(i = 0) \equiv s_I(i = 1) \equiv \mathbf{id}_I$.

Let us recall the definition of fibrancy introduced in CCHM model [CCHM16]—we only rephrased it to use Orton and Pitts' trick to avoid negation[3].

Definition 6 (CCHM fibrancy)**.** Let $\Gamma \hat{\vdash} A$ be a presheaf family. Then A is fibrant if it is equipped with a *composition structure* meaning that, for each $e : \mathtt{OI}$, $I : \square$, $i \# I$, $\rho : \Gamma(I, i)$, $\phi : \mathbb{F}(I)$, u partial element in $A\rho$ of extent ϕ, $a_0 : A\rho(i = e)$ such that for all $f : J \to I$ with $\phi f \equiv 1_{\mathbb{F}}$, $a_0 f \equiv u_{(i=e)f}$, we have an element

$$\mathbf{comp}(e, I, i, \rho, \phi, u, a_0) \ :\ A\rho(i = !e)$$

such that for any $f : J \to I$ and $j \# J$,

$$(\mathbf{comp}(e, I, i, \rho, \phi, u, a_0))f \equiv \mathbf{comp}(e, J, j, \rho(f, i = j), \phi f, u(f, i = j), a_0 f)$$

and $\mathbf{comp}(e, I, i, \rho, 1_{\mathbb{F}}, u, a_0) \equiv u_{(i=!e)}$.

Now if we unfold our definition of fibrancy we get the following.

Proposition 34. Let $\Gamma \hat{\vdash} A$ and $\Gamma, A \hat{\vdash} P$ be presheaf families. Then there is a section of $\Gamma \hat{\vdash} \mathbf{RFib}\ P$ if and only if, for each $e : \mathtt{OI}$, $I : \square$, $i \# I$, $\rho : \Gamma(I)$, $x : A(\rho s_I)$, $\phi : \mathbb{F}(I)$, u partial element in $P(\rho s_I, x)$ of extent ϕ, $p_0 : P(\rho, x(i = e))$ such that for all $f : J \to I$ with $\phi f \equiv 1_{\mathbb{F}}$, $p_0 f \equiv u_{(i=e)f}$, we have an element

$$\mathbf{comp}(e, I, i, \rho, x, \phi, u, p_0) \ :\ P(\rho, x(i = !e))$$

such that for any $f : J \to I$ and $j \# J$,

$$(\mathbf{comp}(e, I, i, \rho, x, \phi, u, p_0))f \equiv \mathbf{comp}(e, J, j, \rho f, x(f, i = j), \phi f, u(f, i = j), p_0 f)$$

and $\mathbf{comp}(e, I, i, \rho, x, 1_{\mathbb{F}}, u, p_0) \equiv u_{(i=!e)}$.

[3] and in type theory of course!

This unfolding makes appear a *degenerate part* (the ρ) and a *regular part* (the x). Thus, we will say that P is regularly fibrant with respect to A and degenerately fibrant with respect to Γ, or simply that P is *regularly fibrant*.
For a presheaf family, we thus get several notions of fibrancy depending on where we put the limit between the degenerate and the regular part.

- If all the dependency is put in the regular part, we recover the CCHM notion of fibrancy. It corresponds to the fact that, in the empty context, $\cdot \hat{\vdash} \mathbf{RFib}\ P$ has a section where P is considered as a family over Γ, A in the empty context. We will say that P is *fully fibrant*.
- If all the dependency is put in the degenerate part, we get something weaker which corresponds to the fact that $\Gamma, A \hat{\vdash} \mathbf{Fib}\ P$ has a section, or equivalently that $\Gamma \hat{\vdash} \mathbf{DFib}\ P$ has a section. We will say that P is *degenerately fibrant*.

We can remark that the fibrancy hypothesis in the transport rule

$$\frac{\Gamma \hat{\vdash} t, t' : A \qquad \Gamma \hat{\vdash} p : t =_A t' \qquad \Gamma, y : A \hat{\vdash} h : \mathbf{Fib}\ P \qquad \Gamma \hat{\vdash} u : P\{y \coloneqq t\}}{\Gamma \hat{\vdash} \mathbf{transport}(P, p, u) : P\{y \coloneqq t'\}}$$

is minimal in the sense that only what is needed to interpret the transport is required to be in the regular part.

The notion of regular fibrancy has been generalized to *damped horn inclusions* by Nuyts in [Nuy18]. It is called *contextual fibrancy* there. In particular, Nuyts has shown that there is always a fibrant replacement for contextual fibrancy. We will demonstrate this for our particular case in Section 4.2.2.

4.2. Model category on cSet

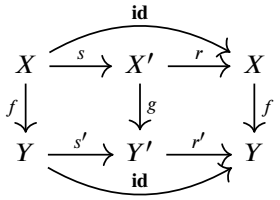
In homotopy theory, model categories are a standard setting of "place to do homotopy theory". A model category is a category equipped with three classes of arrows—the fibrations, the cofibrations and the weak equivalences—enjoying several properties. In particular, they have to give rise to two weak factorization systems. Fibrations can be seen as "nice surjections", cofibrations as "nice injections" and weak equivalence as "homotopy equivalences". Typical examples of model categories are **Top**, the category of topological spaces, **sSet**, the category of simplicial sets, an **cSet**, the category of cubical sets. Model categories are of great importance to compare those different "places to do homotopy" with the notion of Quillen equivalence.
In this section, we use $\mathbb{I}$TT as an internal language of the presheaf category $\widehat{\mathcal{C}}$ in which it is interpreted and show that this one is a model category. In particular it gives a description of a model structure on **cSet** the category of cubical sets with connections (the one of the CCHM model).
The construction goes in two steps: first we show internally in $\mathbb{I}$TT that there is a model structure on $\mathcal{U}$ the universe. Then we remark that the interpretation of this result in $\widehat{\mathcal{C}}$ gives the desired model structure.

It has already been shown in Homotopy Type Theory that there is a pre-model structure on the universe. The first factorization system was given by Gambino and Garner [GG08] using homotopy fibers and the second one by Lumsdaine [Lum11] using mapping cylinders. As HoTT is the fibrant fragment of Two Level Type Theory, this result has to be understood as a pre-model structure on the universe of fibrant types $\mathcal{UF}$ (and not $\mathcal{U}$). Our work is thus a generalization to the universe of arbitrary types. It makes appear the fibrant replacement, which was unneeded in the fibrant case as all types were fibrant.
A description of a model structure on the category of cubical sets has recently been given by Sattler [Sat17] and it should be interesting to compare its model structure with the one given by the interpretation of our internal model structure in 𝕀TT in the cubical sets model.

4.2.1. Model category

Here we present what is a model category in a type theoretic setting, see [Hir09, Hov07] (which are standard references) for a set theoretic account. In the following we fix $\mathcal{C}$ a category (always defined with the strict equality).

Definition 7. Let $f : \mathrm{Hom}(X, Y)$ and $g : \mathrm{Hom}(X', Y')$ be arrows of $\mathcal{C}$. We say that f is a *retract* of g if there exist[4] arrows s, r, s' and r' such that the following diagram commutes:



By a *class* of arrows of $\mathcal{C}$, we simply mean a predicate $P : \Pi X, Y : A.\ \mathrm{Hom}(X, Y) \to \mathcal{U}$. We write $f \in P$ for all function such that $P\ f$ is inhabited. If Q is another class, we write $P \leftrightarrow Q$ if we have $\Pi X, Y.\ \Pi f : \mathrm{Hom}(X, Y).\ P\ f \leftrightarrow Q\ f$; and $P \cap Q$ for the conjunction of the two classes.

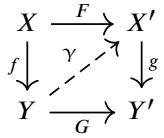
Definition 8. A class P of arrows of $\mathcal{C}$ satisfies the *2-out-of-3* property if, for all arrows $X \xrightarrow{f} Y \xrightarrow{g} Z$ such that two of f, g and $g \circ f$ belong to P, so does the third. More precisely, it means that we have three functions:

- $\Pi f, g.\ P\ f \to P\ g \to P\ (g \circ f)$
- $\Pi f, g.\ P\ (g \circ f) \to P\ f \to P\ g$
- $\Pi f, g.\ P\ g \to P\ (g \circ f) \to P\ f$

Definition 9. Let $f : \mathrm{Hom}(X, Y)$ and $g : \mathrm{Hom}(X', Y')$ be arrows of $\mathcal{C}$. It is said that f has the *left lifting property* (LLP) with respect to g (and that g has the *right lifting property* (RLP) with respect to f) if, for all arrows $F : \mathrm{Hom}(X, X')$ and $G : \mathrm{Hom}(Y, Y')$ such that the square below commutes, there exists an arrow $\gamma : \mathrm{Hom}(Y, X')$ making the two triangle

[4] "exist" is always understood in the constructive sense, *i.e.* as a sigma type and not as the squashed sigma type.

commute:



We then say that an arrow f has the LLP (resp. the RLP) with respect to a class of arrows P if it has it with respect to all arrows of P. We write $\mathbf{LLP}(P)$ (resp. $\mathbf{RLP}(P)$) the class of such arrows.

Definition 10. A *weak factorization system* (wfs) on $\mathcal{C}$ consists of two classes of arrows L and R such that:

1. every arrow f of $\mathcal{C}$ can be factorized as $f \equiv r \circ l$ with $l \in L$ and $r \in R$
2. L is exactly the class of arrows of $\mathcal{C}$ which have the LLP with respect to R:

$$L \leftrightarrow \mathbf{LLP}(R)$$

3. R is exactly the class of arrows of $\mathcal{C}$ which have the RLP with respect to L:

$$R \leftrightarrow \mathbf{RLP}(L)$$

The classes L and R of a weak factorization system enjoy several good properties: they contain all isomorphisms, they are closed under retract, L is closed under pushouts, R is closed under pullbacks, ...

We can now define what is model category.

Definition 11. A *pre-model structure* on $\mathcal{C}$ consists of three classes of arrows F, C and W, the *fibrations*, the *cofibrations* and the *weak equivalences*, such that:

1. W satisfies the 2-out-of-3 property
2. (AC, F) and (C, AF) are two weak factorization systems,
   where $AC \coloneqq C \cap W$ and $AF \coloneqq F \cap W$.

The arrows of AC (resp. AF) are called the *acyclic cofibrations* (resp. *acyclic fibrations*).

If $\mathcal{C}$ has a terminal object $\mathbb{1}$, we say that an object X is *fibrant* if the map $X \to \mathbb{1}$ is a fibration.

Definition 12. A *model category* is a category equipped with a pre-model structure which is complete (it has all small limits) and cocomplete (it has all small colimits).

In the following, we will show that there is a pre-model structure on the category $\mathcal{U}$. For this we first need two constructions: the fibrant replacement and the mapping cylinders.

4.2.2. Fibrant Replacement

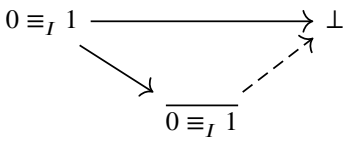
Fibrant Replacement for full fibrancy As fibrancy is a predicate on types in Two Level Type Theory, it would be satisfying to have a fibrant replacement for it acting like a modality. It would have the following rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash \overline{A}} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \overline{A}\ \mathbf{Fib}} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \eta : A \to \overline{A}} \qquad \frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash B\ \mathbf{Fib}}{\Gamma \vdash \mathbf{repl\_rec}(f) : \overline{A} \to B}$$

$$\mathbf{repl\_rec}(f)\ (\eta\ x) \simeq_\beta f\ x$$

However, it has been remarked by several authors that adding such a fibrant replacement is inconsistent with the existence of types which are not h-sets [Cap17] or with an interval [NLab]. We repeat here the proof of the nLab for reference.

Let's suppose I that is a type with (at least) two elements $0, 1 : I$, such that $0 =_I 1$ and $\neg(0 \equiv_I 1)$. For all $x : I$ we have that $\overline{0 \equiv_I x}$ is fibrant. Hence, by transporting $\eta\ \mathbf{refl}_\equiv$ along the equality $0 =_I 1$ we get an element of type $\overline{0 \equiv_I 1}$.
But $\bot$ is fibrant so we can factorize $0 \equiv_I 1 \to \bot$ as:



Thus, from the element of type $\overline{0 \equiv_I 1}$ we get a proof of False.

Degenerate Fibrant Replacement [InternalCubical-Coq/FibRepl.v]
Things go nicer in our setting because we use degenerate fibrancy. It turns out that a "degenerate fibrant replacement" is admissible. In fact, it can be defined in Interval Type Theory, using a quotient inductive type.
The fibrant replacement is the (strict) quotient inductive type defined as follows:

Quotient $\overline{A} :=$
| η : $A \to \overline{A}$
| **hcomp** : $\Pi\,(e : \mathbb{OI})\,(\phi : \mathbf{Cof})\,(a : \Pi\, i : \mathbb{I}.\ \phi \vee i \equiv e \to \overline{A}).\ \overline{A}$
| **eq** : $\Pi\, e,\ \phi,\ (a : \Pi\, i : \mathbb{I}.\ \phi \vee i \equiv e \to \overline{A})\,(w : \phi).\ a\ !e\ (\mathrm{inl}\ w) \equiv \mathbf{hcomp}\ e\ \phi\ a$

η is the embedding of A into $\overline{A}$, **hcomp** freely adds the compositions needed so that $\overline{A}$ is (degenerately) fibrant, and **eq** assert than **hcomp** $e\ \phi\ a$ extends $(\phi, a\ !e \circ \mathrm{inl})$.

It enjoys the following elimination principle:

$$\frac{\begin{array}{c}\vdash P : \overline{A} \to \mathcal{U} \qquad \vdash \eta' : \prod a : A.\ P\ (\eta\ a) \\ \vdash \mathrm{hcomp}' : \prod e,\ \phi,\ (a : \prod i : \mathbb{I}.\ \phi \vee i \equiv e \to \overline{A})\ (a' : \prod i,\ w.\ P\ (a\ i\ w)).\ P\ (\mathbf{hcomp}\ e\ \phi\ a) \\ \vdash \mathrm{eq}' : \prod e,\ \phi,\ a,\ a',\ w.\ \mathbf{eq}\ e\ \phi\ a\ w\ \#^P\ a'\ !e\ (\mathrm{inl}\ w) \equiv \mathrm{hcomp}'\ e\ \phi\ a\ a'\end{array}}{\vdash \mathbf{repl\_ind}(P,\ \eta',\ \mathrm{hcomp}',\ \mathrm{eq}') : \prod z : \overline{A}.\ P\ z}$$

$$\mathbf{repl\_ind}(P,\ \eta',\ \mathrm{hcomp}',\ \mathrm{eq}')\ (\eta\ a) \simeq_\beta \eta'\ a$$

$$\begin{aligned}&\mathbf{repl\_ind}(P,\ \eta',\ \mathrm{hcomp}',\ \mathrm{eq}')\ (\mathbf{hcomp}\ e\ \phi\ a) \\ &\qquad \simeq_\beta \mathrm{hcomp}'\ e\ \phi\ a\ (\lambda\, i,\ w.\ \mathbf{repl\_ind}(P,\ \eta',\ \mathrm{hcomp}',\ \mathrm{eq}')(a\ i\ w))\end{aligned}$$

From this "raw elimination principle" generated by the quotient, we can prove that the fibrant replacement has another elimination principle, which the one expected for a fibrant replacement.

Proposition 35. The degenerate fibrant replacement satisfies the following elimination principle:

$$\frac{\vdash P : \overline{A} \to \mathcal{U} \qquad \vdash \eta' : \prod a : A.\ P\ (\eta a) \qquad \vdash H : \mathbf{RFib}\ P}{\vdash \mathbf{repl\_ind}'(P,\ \eta',\ H) : \prod z : \overline{A}.\ P\ z}$$

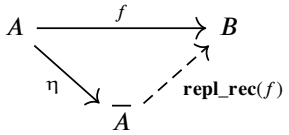
$$\mathbf{repl\_ind}'(P,\ \eta',\ H)\ (\eta\ a) \simeq_\beta \eta'\ a$$

Proof. $\mathbf{repl\_ind}'(P,\ \eta',\ H)$ is defined by induction on the quotient (using $\mathbf{repl\_ind}$). In the case of $\mathbf{hcomp}\ e\ \phi\ a$, modulo strict rewritings, the provided term is given by a composition in P:

$$\mathbf{comp}_P\ e\ a_0\ \phi\ p_0$$

with $a_0 \coloneqq \lambda\, i.\ \mathbf{hcomp}\ e\ (\phi \vee (i \equiv e))\ (\lambda\, j\ w.\ a\ (i \sqcap_e j)\ (\ldots\ \#\ w))$
and $p_0 \coloneqq \lambda\, i\ w.\ (\ldots)\ \#\ w$
where $\sqcap_0 \coloneqq \wedge$ and $\sqcap_{\mathbb{I}} \coloneqq \vee$. □

Instantiated with a constant type family $\lambda\_ : A.\ B$ we get the non-dependent elimination principle asserting for every map $f : A \to B$, if B is fibrant, then f can be factorized as:



(**Fib** B)

and the triangle commutes definitionally.
With this non-dependent elimination principle we can lift every map $f : A \to B$ to a map $\overline{f} : \overline{A} \to \overline{B}$ and we have the following equalities:

$$\overline{f}\ (\eta\ x) \simeq_\beta \eta\ (f\ x) \qquad \overline{g \circ f} \equiv \overline{g} \circ \overline{f} \qquad \overline{\mathbf{id}_A} \equiv \mathbf{id}_{\overline{A}}$$

The two strict equalities are proven by induction on the quotient, using the first elimination principle **repl_ind**.
Last, as $\mathcal{U}$ is fibrant (it even has an extension structure), every type family $P : A \to \mathcal{U}$ can be lifted to a type family over $\overline{A}$:

$$\begin{aligned} \widetilde{P} \ &: \ \overline{A} \to \mathcal{U} \\ &:= \ \mathbf{repl\_rec}(P) \end{aligned}$$

For the following, we will need the property the if P is regularly fibrant then $\widetilde{P}$ is regularly fibrant too. We call the property the *extension rule*. Although we got the intuition that it holds, we didn't succeed in proving it for the moment.

Conjecture 1 (extension rule)**.** *For all type family $P : A \to \mathcal{U}$ we have:*

$$\mathbf{RFib}\ P \to \mathbf{RFib}\ \widetilde{P}$$

The extension rule has a very strong consequence, it allows transporting along an equality $\eta\ x = \eta\ y$ to get a map $P\ x \to P\ y$ if P is regularly fibrant. Indeed, in this case, $\widetilde{P}$ is also fibrant. Hence transport provides:

$$\eta\ x = \eta\ y \to \widetilde{P}\ (\eta\ x) \to \widetilde{P}\ (\eta\ y)$$

And we have $\widetilde{P}\ (\eta\ x) \simeq_\beta P\ x$ and $\widetilde{P}\ (\eta\ y) \simeq_\beta P\ y$.

One of the reasons making us believe that the extension rule holds is that it is valid in the empty context. The proof uses the univalent universe $\mathcal{UF}$ of "fully fibrant families".
Let $P : A \to \mathcal{U}$ be a regularly fibrant family in the empty context. Since the context is empty, P is in fact fully fibrant, hence it can be lifted to a map

$$P' : A \to \mathcal{UF}$$

But the universe $\mathcal{UF}$ is itself fibrant, hence we have the map:

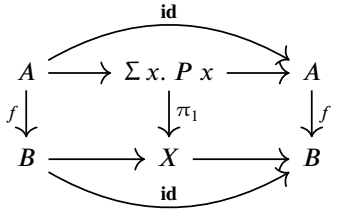
$$\mathbf{repl\_rec}(P') : \overline{A} \to \mathcal{UF}$$

By the fibrancy rule of $\mathcal{UF}$, we have that $\mathbf{repl\_rec}(P') : \overline{A} \to \mathcal{U}$ is fully fibrant, and hence regularly fibrant. We check that this map is indeed $\widetilde{P}$.

4.2.3. (AC, F)-WFS [InternalCubical-Coq/Model_structure.v]

The fibrant replacement allows us to define the first factorization system, given by homotopy fibers, and weak equivalences. Our design choices are guided by the following definition of fibrations.

Definition 13. A function $f : A \to B$ is said to be a *fibration* if there exists a **regularly fibrant** type family $P : X \to \mathcal{U}$ such that f is a retract of $\pi_1 : \Sigma\, x : X.\, P\, x \to X$.

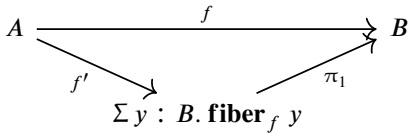


We write F the class of fibrations. The class of acyclic cofibrations is defined to be $\mathbf{LLP}(F)$ for the moment, we will check that it coincides with $C \cap W$ afterward.

For a map $f : A \to B$, let's define its homotopy fiber in $y : B$ to be:

$$\mathbf{fiber}_f\ y \coloneqq \Sigma\, x : \overline{A}.\ \overline{f}\ x = \eta\ y$$

In Homotopy Type Theory (the fibrant fragment of Two Level Type Theory), the fiber is defined as $\Sigma\, x : A.\ f\ x =_B y$. We add the fibrant replacement so that for every map f the type family $\lambda\, y : B.\ \mathbf{fiber}_f\ y$ is always regularly fibrant (because the identity type is taken in $\overline{B}$).
Then every function $f : A \to B$ factorizes through its fibers:



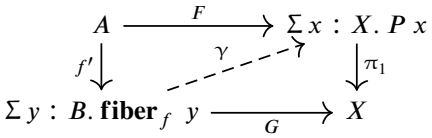
with $f' \coloneqq \lambda\, x : A.\ (f\ x,\ \eta\ x,\ \mathbf{refl}_{=}\ (\eta\ (f\ x)))$.

Proposition 36. $(\mathbf{LLP}(F),\ F)$ is a weak factorization system on $\mathcal{U}$.

Proof. We have to check that:

- for all f, $f' \in \mathbf{LLP}(F)$
- $\mathbf{RLP}(\mathbf{LLP}(F)) \subseteq F$

We only give the proof of the first point. All other proofs of this section are similar and can be found in the formalization.
As the lifting property is stable under retracts, to show that $f' \in \mathbf{LLP}(F)$ we only have to solve the following lifting problem:



with P a regularly fibrant family.
The map γ is defined as the composition:

$$\Sigma\, y : B.\ \mathbf{fiber}_f\ y \xrightarrow{(\mathrm{id},\,\alpha)} \Sigma\, y : B.\ P\ (G\ y) \xrightarrow{\beta} \Sigma\, x : X.\ P\ x$$

where $\beta(w, z) \coloneqq (G\ w, z)$. To provide α, we need a map of type:

$$\Pi\, (y : B)\, (x' : \overline{A})\, (p : \overline{f}\ x' = \eta\ y).\ P\ (G\ (y, x', p))$$

By induction on the fibrant replacement (P is regularly fibrant) we need:

$$\Pi\, (y : B)\, (x : A)\, (p : \eta\ (f\ x) = \eta\ y).\ P\ (G\ (y, \eta\ x, p))$$

Then by using the path-induction principle given by the extension rule:

$$\Pi\, x : A.\ P\ (G\ (f\ x, \eta\ x, \mathbf{refl}_{=}))$$

which is given by F.
We see here that the extension rule is crucial to have the lifting. □

4.2.4. Weak equivalences [InternalCubical-Coq/Equivalences.v]

To define weak equivalences we use HoTT equivalences. Let's recall that they are defined by:

$$\begin{aligned}&\mathbf{IsEquiv}(f)\\ &\coloneqq \Sigma\, (g : B \to A)\, (\eta : \Pi\, x.\ g\ (f\ x) = x)\, (\varepsilon : \Pi\, x.\ f\ (g\ x) = x).\ \Pi\, x.\ \mathbf{ap}\ f\ (\eta\ x) = \varepsilon\ (g\ x)\end{aligned}$$

In [OP17], Orton and Pitts use Path types to define equivalences. Here we choose to use identity types because it implies less conversions between paths and identity types in the following. Of course, the two types are logically equivalent but we do not know which one is more meaningful.

Definition 14. A map $f : A \to B$ is a *weak equivalence* if $\overline{f} : \overline{A} \to \overline{B}$ is an equivalence:

$$W(f) \coloneqq \mathbf{IsEquiv}\ \overline{f}$$

Here the fibrant replacement seems to play a role similar to geometric realization. In homotopy theory, the usual model structure on cubical sets has for weak equivalences the maps whose geometric realization in Top are weak equivalences.

Remark. If we look at the characterization of equivalences with contractibility of fibers for $\overline{f}$ we get:

$$\Pi\, y' : \overline{B}.\ \mathbf{Contr}\ (\Sigma\, x' : \overline{A}.\ \overline{f}\ x' = y')$$

But by induction on the fibrant replacement (the predicate is regularly fibrant), this is equivalent to:

$$\Pi\, y : B.\ \mathbf{Contr}\ (\Sigma\, x' : \overline{A}.\ \overline{f}\ x' = \eta\ y)$$

It means that we recover the contractibility of fibers of f for our definition of $\mathbf{fiber}_f$.

4.2.5. Cylinder [InternalCubical-Coq/Cylinder.v]

For the second factorization system, we need cylinders, which are the dual of fibers. We define cylinders from homotopy pushouts, so let's first define homotopy pushouts.

Pushout

In Homotopy Type Theory, homotopy pushouts are an instance of Higher Inductive Types (HIT). Recently, Coquand *et. al.* gave the interpretation of several HIT including pushouts in [CHM18]. We replay here their construction in light of the fibrant replacement.
Let's fix the following span of which we want to construct the pushout:

$$\begin{array}{ccc} A & \xrightarrow{g} & C \\ {\scriptstyle f}\downarrow & & \\ B & & \end{array}$$

The pushout of f and g is the following HIT:

$$\begin{aligned}
&\textsf{HIT}\ B \sqcup_A C \coloneqq \\
&\mid \mathbf{pol} \ : \ B \to B \sqcup_A C \\
&\mid \mathbf{por} \ : \ C \to B \sqcup_A C \\
&\mid \mathbf{poq} \ : \ \prod x : A.\ \mathbf{pol}\ (f\ x) =_{B \sqcup_A C} \mathbf{por}\ (g\ x)
\end{aligned}$$

Remark. The homotopy pushout should not be confused with the strict pushout given by a strict quotient.

Now we want to *define* this HIT in 𝕀TT. The construction goes in two steps: first we define a "naive" strict quotient and then we take the fibrant replacement. Let $\mathbf{PO_0}$ to be the following strict quotient:

$$\begin{aligned}
&\textsf{Quotient}\ \mathbf{PO_0} \coloneqq \\
&\mid \mathbf{pol_0} \ : \ B \to \mathbf{PO_0} \\
&\mid \mathbf{por_0} \ : \ C \to \mathbf{PO_0} \\
&\mid \mathbf{poq_0} \ : \ A \to \mathbb{I} \to \mathbf{PO_0} \\
&\mid \mathbf{poq_l} \ : \ \prod x : A.\ \mathbf{poq_0}\ x\ 0 \equiv_{\mathbf{PO_0}} \mathbf{pol_0}\ (f\ x) \\
&\mid \mathbf{poq_r} \ : \ \prod x : A.\ \mathbf{poq_0}\ x\ 1 \equiv_{\mathbf{PO_0}} \mathbf{por_0}\ (g\ x)
\end{aligned}$$

$\mathbf{pol_0}$ and $\mathbf{por_0}$ interpret **pol** and **por**, and $\mathbf{poq_0}\ x$ is a path from $\mathbf{pol_0}\ (f\ x)$ to $\mathbf{por_0}\ (g\ x)$. The endpoints agree thanks to the strict equalities by which we quotient. We will write $\mathbf{PO_0}$**_ind** the eliminator generated by the strict quotient.
Unfortunately this type is never fibrant and hence we take the fibrant replacement:

$$\begin{aligned}
B \sqcup_A C \quad &\coloneqq \quad \overline{\mathbf{PO_0}} \\
\mathbf{pol} \ : \ B \to B \sqcup_A C \quad &\coloneqq \quad \eta \circ \mathbf{pol_0}
\end{aligned}$$

$$\mathbf{por} \ : \ C \to B \sqcup_A C \quad \coloneqq \quad \eta \circ \mathbf{por_0}$$

$$\mathbf{poq} \ : \ \prod x : A.\, \mathbf{pol}\ (f\ x) \sim_{B \sqcup_A C} \mathbf{por}\ (g\ x) \quad \coloneqq \quad \mathbf{ap}_\eta\ (\mathbf{poq_0}, \mathbf{poq_l}, \mathbf{poq_r})$$

Let's remark that **poq** is a path and not an inhabitant of an identity type. Hence, if we want to recover an identity type we have to apply **paths2id**. In general, the interaction between HIT and identity types remains to be studied.
As in the case of the fibrant replacement, we can define the expected elimination principle from the "raw elimination principle":

Proposition 37. $B \sqcup_A C$ satisfies the following elimination principle.

$$\frac{\begin{array}{c}\vdash P : B \sqcup_A C \to \mathcal{U} \qquad \vdash H : \mathbf{RFib}\ P \\ \vdash \mathrm{pol}' : \prod b : B.\ P\ (\mathbf{pol}\ b) \qquad \vdash \mathrm{por}' : \prod c : C.\ P\ (\mathbf{por}\ c) \\ \vdash \mathrm{poq}' : \prod x : A.\ \mathbf{transport}(P,\ \mathbf{poq}\ x,\ \mathrm{pol}'\ (f\ x)) \sim_{P\ (\mathbf{por}\ (g\ x))} \mathrm{por}'\ (g\ x)\end{array}}{\vdash \mathbf{PO\_ind}(P,\ H,\ \mathrm{pol}',\ \mathrm{por}',\ \mathrm{poq}') : \prod w : B \sqcup_A C.\ P\ w}$$

$$\mathbf{PO\_ind}(P,\ H,\ \mathrm{pol}',\ \mathrm{por}',\ \mathrm{poq}')\ (\mathbf{pol}\ b) \simeq_\beta \mathrm{pol}'\ b$$

$$\mathbf{PO\_ind}(P,\ H,\ \mathrm{pol}',\ \mathrm{por}',\ \mathrm{poq}')\ (\mathbf{por}\ c) \simeq_\beta \mathrm{por}'\ c$$

The transport in poq' is a transport along a *path*.

Proof. The proof goes by induction on the fibrant replacement using **repl_ind**' (P is regularly fibrant) and then by induction on the quotient with $\mathbf{PO_0\_ind}$. A two-dimensional Kan filler (in P) is needed to complete the case of the $\mathbf{poq_0}$. See the formalization. □

Remark. For the non-dependent eliminator we have a strict computational rule on **poq**:

$$\mathbf{ap}_{\mathbf{PO\_rec}(P,\ H,\ \mathrm{pol}',\ \mathrm{por}',\ \mathrm{poq}')}\ (\mathbf{poq}\ x) \equiv \mathrm{poq}'\ x$$

Even if we expect it, we don't know, at the time of writing, if a similar strict equality holds for the dependent eliminator.

Pushout satisfies the following fibrancy rule.

Proposition 38. If $A, B, C : X \to \mathcal{U}$ are regularly fibrant type families over X. Then for all maps $f : \prod x : X.\ A\ x \to B\ x$ and $g : \prod x : X.\ A\ x \to C\ x$, the type family given by the pushout $\lambda x : X.\ B\ x \sqcup_{A\ x} C\ x$ is regularly fibrant.

$$\frac{\vdash A\ \mathbf{RFib} \qquad \vdash B\ \mathbf{RFib} \qquad \vdash C\ \mathbf{RFib}}{\vdash \lambda x : X.\ B\ x \sqcup_{A\ x} C\ x\ \mathbf{RFib}}$$

This property is quite remarkable because, in the general case, the pointwise fibrant replacement $\lambda x.\ \overline{A\ x}$ has no hope to be regularly fibrant. However, it is the case for this particular family $\lambda x.\ \overline{\mathbf{PO_0}\ x}$. The secret lies in a decomposition of regular fibrancy into degenerate fibrancy and transport structure. This decomposition was discovered by Coquand *et. al.* [CHM18].

Definition 15. A *transport structure* on a type family $P : A \to \mathcal{U}$ is an inhabitant of the following type:

$$\mathbf{Trans}\ P \coloneqq \Pi\,(e : \mathbb{OI})\,(a : \mathbb{I} \to A)\,(\phi : \mathbf{Cof})\,(\mathrm{cst} : \phi \to \Pi\, i : \mathbb{I}.\ a\ e \equiv a\ i)(p_0 : P\ (a\ e)).$$
$$\Sigma\, p_1 : P\ (a\ !e).\ \Pi\, w : \phi.\ \mathrm{cst}\ \ w\ !e\ \#\ p_0 \equiv p_1$$

It means that there is a composition operation along all paths a which are constant on the face ϕ considered.

Proposition 39. A type family P is regularly fibrant if and only if it is degenerately fibrant and has a transport structure:

$$\mathbf{RFib}\ P \quad \leftrightarrow \quad \mathbf{DFib}\ P \times \mathbf{Trans}\ P$$

Proof. See [CHM18]. □

Proposition 40. The fibrant replacement preserves the transport structure:

$$\mathbf{Trans}\ P \quad \to \quad \mathbf{Trans}\ (\lambda\, x.\ \overline{P\ x})$$

Proof. The following argument comes from Cavallo and Orton[5].
This property is true for any endofunctor[6] on $\mathcal{U}$, that is to say, for any operator $R : \mathcal{U} \to \mathcal{U}$ such that there is an operator

$$F : \Pi\, A,\ B : \mathcal{U}.\ (A \to B) \to (RA \to RB)$$

with $F\ \mathrm{id}_A \equiv \mathrm{id}_{RA}$ and $F\ (g \circ f) \equiv F\ g \circ F\ f$. We have seen that such equalities hold for the fibrant replacement.
The transport operation on P gives a map $P\ (a\ e) \to P\ (a\ !e)$. By applying F we get a map $R\ (P\ (a\ e)) \to R\ (P\ (a\ !e))$. The fact that F preserves identity maps is enough to show that it actually gives a transport structure on $R \circ P$. □

The pushout is degenerately fibrant (because of the fibrant replacement), hence it remains to show that it has a transport structure.

Proposition 41. $\lambda\, x.\ \overline{\mathbf{PO_0}\ x}$ has a transport structure.

Proof. See [CHM18] or the formalization in `InternalCubical-Coq/Cylinder.v`. □

Cone

For any type A, the cone of A is the following homotopy pushout:

$$\begin{array}{ccc} A & \longrightarrow & \mathbb{1} \\ {\scriptstyle \mathrm{id}}\downarrow & & \downarrow \\ A & \longrightarrow & \mathbf{cone}\ A \end{array}$$

[5] https://github.com/IanOrton/cubical-topos-experiments/blob/master/src/hcomp-trans.agda
[6] In fact, we don't even need that fact that the functor preserves composition, only that it preserves identity maps.

So we define $\mathbf{cone}\ A \coloneqq A \sqcup_A \mathbb{1}$. It can also be seen directly as an HIT:

$$\begin{aligned}&\mathsf{HIT}\ \mathbf{cone}\ A \coloneqq\\ &\mid \mathbf{inc} : A \to \mathbf{cone}\ A\\ &\mid \mathbf{point} : \mathbf{cone}\ A\\ &\mid \mathbf{eq} : \prod x : A.\, \mathbf{inc}\ x =_{\mathbf{cone}\ A} \mathbf{point}\end{aligned}$$

Cones satisfy the following elimination principle:

$$\frac{\begin{array}{c}\vdash P : \mathbf{cone}\ A \to \mathcal{U} \qquad \vdash H : \mathbf{RFib}\ P \qquad \vdash \mathrm{inc}' : \prod x : A.\, P\ (\mathbf{inc}\ x)\\ \vdash \mathrm{point}' : P\ \mathbf{point} \qquad \vdash \mathrm{eq}' : \prod x : A.\, \mathbf{transport}(P, \mathbf{eq}\ x, \mathrm{inc}'\ x) \sim \mathrm{point}'\end{array}}{\vdash \mathbf{cone\_ind}(P, H, \mathrm{inc}', \mathrm{point}', \mathrm{eq}') : \prod w : \mathbf{cone}\ A.\, P\ w}$$

The cone enjoys the expected fibrancy rule $\mathbf{RFib}\ A \to \mathbf{RFib}\,(\lambda x.\, \mathbf{cone}\,(A\,x))$.

Proposition 42. A cone is always contractible. For any type A we have:

$$\mathbf{Contr}\ (\mathbf{cone}\ A)$$

Proof. By induction with **cone_ind**. □

Cylinder

At last, we define the *mapping cylinder*.
Let $f : A \to B$ be a map and $y : B$. The mapping cylinder of f in y is:

$$\mathbf{Cyl}_f\ y \coloneqq \mathbf{cone}\,(\Sigma\, x : A.\, f\ x = y)$$

Then, the cylinder satisfies the following introduction and elimination rules:

$$\begin{aligned}\mathbf{top} &: \prod x : A.\, \mathbf{Cyl}_f\,(f\ x)\\ &\coloneqq \lambda x : A.\, \mathbf{inc}\,(x, \mathbf{refl}_{=}\,(f\ x))\\ \mathbf{base} &: \prod y : B.\, \mathbf{Cyl}_f\ y\\ &\coloneqq \lambda y : B.\, \mathbf{point}\\ \mathbf{cyleq} &: \prod x : A.\, \mathbf{top}\ x \sim_{\mathbf{Cyl}\,(f\ x)} \mathbf{base}\,(f\ x)\\ &\coloneqq \lambda x : A.\, \mathbf{eq}\,(x, \mathbf{refl}_{=}\,(f\ x))\end{aligned}$$

$$\frac{\begin{array}{c}\vdash P : \prod y : B.\, \mathbf{Cyl}_f\ y \to \mathcal{U} \qquad \vdash H : \mathbf{RFib}_2\ P\\ \vdash \mathrm{top}' : \prod x : A.\, P\,(f\ x)\,(\mathbf{top}\ x) \qquad \vdash \mathrm{base}' : \prod y : B.\, P\ y\,(\mathbf{base}\ y)\\ \vdash \mathrm{cyleq}' : \prod x : A.\, \mathbf{transport}(P, \mathbf{cyleq}\ x, \mathrm{top}'\ x) \sim \mathrm{base}'\,(f\ x)\end{array}}{\vdash \mathbf{cyl\_ind}(P, H, \mathrm{top}', \mathrm{base}', \mathrm{cyleq}') : \prod (y : B)\,(w : \mathbf{Cyl}_f\ y).\, P\ y\ w}$$

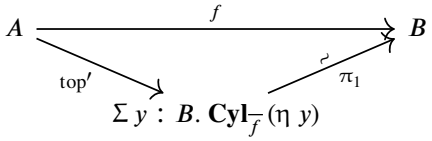
Proposition 43. Let A and B be regularly fibrant families $X \to \mathcal{U}$. Then **Cyl** is regularly fibrant as a three variables type family:

$$\mathbf{RFib}\,(\lambda\,(x, f, y) : \Sigma\,(x : X)\,(f : A\,x \to B\,x)\,(y : B\,x).\ \mathbf{Cyl}_f\ y.)$$

Last, as a cylinder is cone, all cylinders $\mathbf{Cyl}_f\ y$ are contractible.

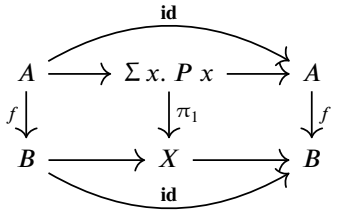
4.2.6. (C, AF)-WFS [InternalCubical-Coq/Model_structure.v]

We can now describe the second factorization system. Every function f factorizes as:



where $\mathrm{top}' \coloneqq \lambda\, x.\, (f\ x, \mathbf{top}_f\ (\eta\ x))$.
With respect to the factorization that Lumsdaine gave for the fibrant fragment [Lum11], the fibrant replacement is added. It is added so that the family $\lambda\, y.\, \mathbf{Cyl}_f\ (\eta\ y)$ is regularly fibrant. The need for the contractibility of cylinders comes from the following characterization of acyclic fibrations.

Proposition 44. A map $f : A \to B$ is an acyclic fibration (*i.e.* both a fibration and a weak equivalence) if and only if there exists a **regularly fibrant** type family $P : X \to \mathcal{U}$ with **contractible** fibers ($\Pi\, x : X.\, \mathbf{Contr}\ (P\ x)$) such that f is a retract of $\pi_1 : \Sigma\, x : X.\, P\ x \to X$.



We write AF the class of acyclic fibrations. The class of cofibrations is defined as $\mathbf{LLP}(AF)$.

We can check that we have indeed a weak factorization system.

Proposition 45. $(\mathbf{LLP}(AF),\ AF)$ is a weak factorization system on $\mathcal{U}$.

4.2.7. Model structure on $\mathcal{U}$

We check that we have taken a correct characterization of acyclic fibrations.

Proposition 46. A map $f : A \to B$ is in $\mathbf{LLP}(F)$ if and only if it is both a weak equivalence and in $\mathbf{LLP}(AF)$.

As a consequence, if we plug the two weak factorization systems together we have a pre-model structure.

Theorem 47. Under the assumption that the extension rule is correct, there is a pre-model structure on $\mathcal{U}$ with the weak equivalences, fibrations and cofibrations as previously defined. Σ types and strict equality give all small limits and strict quotients give small colimits. Hence $\mathcal{U}$ is a model category.

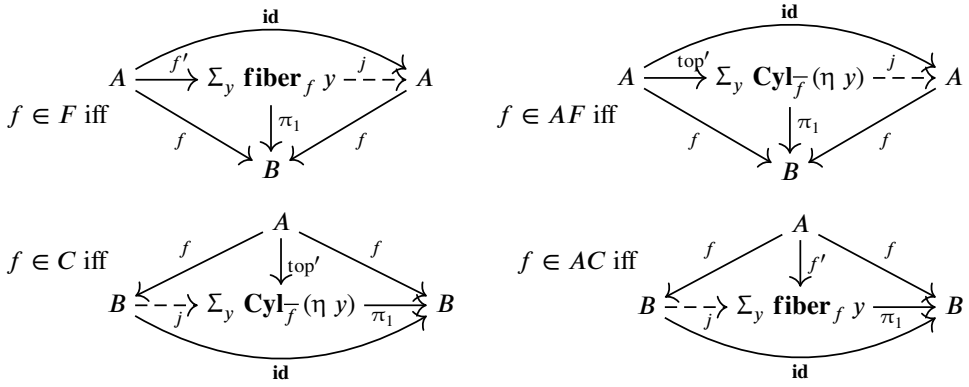


Figure 4.8.: Diagrammatic illustration of Proposition 49

With the internal language point of view (Section 4.1.5), the model structure on the universe is exactly translated as a model structure on the presheaf category. Hence we can have the following interpretation of the theorem:

Corollary 48. Let $\square$ be a cube category such that $\widehat{\square}$ enjoys the nine axioms of Figures 4.4 and 4.5 and such that the extension rule holds. Then there is a model structure on the category of cubical sets $\widehat{\square}$ with the classes as previously defined.

Remark. As remarked by Coquand, the proof of the theorem 47 only uses the extension rule in the empty context which, in this case, follows from the fibrancy of $\mathcal{U}_F$.

4.2.8. Characterization of the classes

On the way of proving the previous theorem we (re)discovered some characterizations of the four classes of maps F, C, AF and AC.
The two factorization systems give prototypical examples of maps which are (acyclic) fibrations and (acyclic) cofibrations. We proved that, in fact, all (acyclic) fibrations and (acyclic) cofibrations arise as retracts, in a canonical sense, of such maps.

Proposition 49. Let A and B be arbitrary types and $f : A \to B$ a map. Then we have:

- f is a fibration if and only if there exists $j : \Sigma\, y.\ \mathbf{fiber}_f\ y \to A$ such that $f \circ j \equiv \pi_1$ and $j \circ f' \equiv \mathbf{id}_A$
- f is an acyclic fibration if and only if there exists $j : \Sigma\, y.\ \mathbf{Cyl}_{\overline{f}}(\eta\ y) \to A$ such that $f \circ j \equiv \pi_1$ and $j \circ \mathrm{top}' \equiv \mathbf{id}_A$
- f is a cofibration if and only if there exists $j : B \to \Sigma\, y.\ \mathbf{Cyl}_{\overline{f}}(\eta\ y)$ such that $j \circ f \equiv \mathrm{top}'$ and $\pi_1 \circ j \equiv \mathbf{id}_B$

- f is an acyclic cofibration if and only if there exists $j : B \to \Sigma\, y.\ \mathbf{fiber}_f\ y$ such that $j \circ f \equiv f'$ and $\pi_1 \circ j \equiv \mathbf{id}_B$

where f' is $\lambda\, x.\ (f\ x, \eta\ x, \mathbf{refl}_{=}\ (\eta\ (f\ x)))$ and top' is $\lambda\, x.\ (f\ x, \mathbf{top}_f\ (\eta\ x))$.

Gambino and Garner already gave the characterization of fibrations (in the fibrant case) and call such maps *type-theoretic normal isofibration* [GG08].
Acyclic cofibrations have an even better description. As already noticed by Gambino and Garner, they are the injective equivalences. We generalize their definition to encompass non-fibrant types.

Proposition 50. Let A and B be arbitrary types and $f : A \to B$ a map. Then f is an acyclic cofibration if and only if it is an *injective equivalence*, *i.e.* if and only if there exists $r : B \to \overline{A}$ and

- $\theta : \prod x : A.\ r\ (f\ x) \equiv \eta_A\ x$
- $\varepsilon : \prod y : B.\ \overline{f}\ (r\ y) = \eta_B\ y$
- $\alpha : \prod x : A.\ \mathbf{strict2id}\ (\mathbf{ap}_{\equiv}\ \overline{f}\ (\theta\ x)) = \varepsilon\ (f\ x)$.

where **strict2id** is the map $x \equiv y \to x = y$.
The last condition means that $\varepsilon\ (f\ x)$ is "identity-equal" to $\mathbf{refl}_{=}$ modulo strict equality rewritings.

4.2.9. Formalization

All the results of this section have been formalized in Coq using the implementation described in Section 4.1.3.
As a small example, fibrations are defined by the following record:

```
Record IsFib {A B} (f : A → B) :=
  { fib_A : Type ;
    fib_P : fib_A → Type ;
    fib_Fib : RFib fib_P;
    fib_H : f RetractOf (π1 fib_P) }.
```

The statement of the main theorem is very short:

```
Theorem type_model_structure : model_structure TYPE.
```

And the statements of the characterizations look like this:

```
Definition F_caract {A B} {f: A → B} :
IsFib f
    ↔ ∃ (g : sigT (fiber f) → A), f o g == pr1 × g o (f' f) == idmap.
```

The axioms on which the formalization is relying can be get with the `Print Assumptions` command. This command crawls recursively the environment and looks for the axioms or

admitted lemmas on which a definition is relying. Its output for the main theorem and the four characterizations is given in Figures 4.9 and 4.10. The first group of Figure 4.9 is the set of assumptions on the equality of the metatheory (Propext, Proof Irrelevance and FunExt). The second group is the set of axioms used to model Interval Type Theory and the last group is the set of axioms used to implement strict quotient types (for the fibrant replacement and homotopy pushouts). On Figure 4.10, the first and second groups are admitted lemmas. The two first are the standard properties of equivalences *on fibrant types*. The three others are the fibrancy rules, they were already formalized by Orton and Pitts in Agda. The last assumption is the only true hypothesis of our development, it is the extension rule (Conjecture 1).

```
propext : ∀ A B : Prop, A ↔ B → A = B
proof_irr : ∀ (P : Prop) (x y : P), x = y
funext : ∀ A (P : A → Type) (f g : ∀ x : A, P x), f == g → f = g


I : Type
zero : I
one : I
sup : I → I → I
inf : I → I → I
zero_one : 0 ≠ 1
zero_sup : ∀ i : I, sup 0 i = i
zero_inf : ∀ i : I, inf 0 i = 0
sup_zero : ∀ i : I, sup i 0 = i
sup_one : ∀ i : I, sup i 1 = 1
one_sup : ∀ i : I, sup 1 i = 1
one_inf : ∀ i : I, inf 1 i = i
inf_zero : ∀ i : I, inf i 0 = 0
inf_one : ∀ i : I, inf i 1 = i
cof_or : ∀ P Q : Prop, cof P → cof Q → cof (P ∨ Q)
cof : Prop → Prop
cof1 : ∀ i : I, cof (i = 1)
cof0 : ∀ i : I, cof (i = 0)
strictness : ∀ (φ : Cof) (A : φ → Type) B (s : ∀ u, SEquiv (A u) B),
  ∃ B' (s' : SEquiv B' B) (e : ∀ u, A u = B'),
    ∀ u, Etransport (λ X, SEquiv X B) (e u) (s u) = s'
join : ∀ A φ φ', cof φ → cof φ' → ∀ (f : φ → A) (g : φ' → A),
  (∀ (u : φ) (v : φ'), f u = g v) → φ ∨ φ' → A
join_r : ∀ A φ φ' (Hφ : cof φ) (Hφ' : cof φ') f g H (v : φ'),
  join f g H (or_intror v) = g v
join_l : ∀ A φ φ' (Hφ : cof φ) (Hφ' : cof φ') f g H (v : φ'),
  join f g H (or_introl u) = f u


qq : ∀ A (e : OI) (φ : Cof) (a : ∀ i : I, φ ∨ i = e → repl A),
  extends (φ; λ x, a !e (or_introl x)) (hcomp e φ a)
push_l : ∀ A B C (f : A → B) (g : A → C) (x : A),
  push x 0 = pol (f x)
push_r : ∀ A B C (f : A → B) (g : A → C) (x : A),
  push x 1 = por (g x)
```

Figure 4.9.: Assumptions on which the formalization rely (1/2)

```
two_out_of_three_IsEquiv
  : two_out_of_three (λ (A B : TYPEH) (f : A ⟶ B), IsEquiv f)
isequiv_adjointify : ∀ A B, Fib A → Fib B →
  ∀ (f : A → B) (g : B → A), f o g I~ ~ idmap → g o f I~ ~ idmap
  → IsEquiv f


UFib_forall : ∀ A (B : A → Type) (C : ∀ x : A, B x → Type),
  UFib B → UFib2 C → UFib (λ x : A, ∀ y : B x, C x y)
UFib_sigma : ∀ A (B : A → Type) (C : ∀ x : A, B x → Type),
  UFib B → UFib2 C → UFib (λ x : A, ∃ y, C x y)
UFib_paths : ∀ A (B : A → Type), UFib B
  → UFib (λ x : ∃ x : A, B x × B x, fst x.2 ~ snd x.2)
UFib_Id : ∀ A (B : A → Type), UFib B
  → UFib (λ x : ∃ x : A, B x × B x, fst x.2 I~ snd x.2)


extension_rule : ∀ (A : Type) (P : A → Type), UFib P → UFib (lift P)
```

Figure 4.10.: Assumptions on which the formalization rely (2/2)

A. Emulating Homotopy Type System in Coq

We describe here a way to emulate Two Level Type Theory in the Coq proof assistant using typeclass inference. This development can be found in the folder HTS-Coq.

First we define a typeclass `Fibrant` to keep track of fibrant types. The same technique has already been used in the HoTT library [BGL+16] to keep track of the use of functional extensionality and univalence axioms.

```
Axiom Fibrant : Type → Type.
Existing Class Fibrant.
```

As `Fibrant` is declared as an axiom, the only way to inhabit this class is to use postulated fibrancy rules. For instance, the rule for the dependent product and universe is:

```
Axiom fibrant_∀ : ∀ (A : Type) (B : A → Type),
    Fibrant A → (∀ x, Fibrant (B x)) → Fibrant (∀ x, B x).
Axiom fibrant_Type : Fibrant Type.
```

They are then declared as instances so that fibrancy is always inferred automatically:

```
Existing Instance fibrant_∀.
Existing Instance fibrant_Type.
```

And each time a new inductive type is declared, an axiom corresponding to its fibrancy rule has to be added.
Then we define the identity types as a private inductive type. The private inductive types forbid to use pattern matching for this inductive type outside of the module where it is defined. It gives an additional layer of abstraction. They have been introduced in Coq to define Higher Inductive Types which behave correctly. Here we use them to forbid the use of the elimination principle of univalent equality when the predicate and the type are not fibrant.

```
Module Export PathsDefinition.
  Private Inductive paths {A : Type} (x : A) : A → Type :=
  | idpath : paths x x.

  Definition paths_ind {A} {x : A} (P : ∀ y : A, paths x y → Type)
     {HP : ∀ y p, Fibrant (P y p)} (u : P x idpath) {y} (p : paths x y)
  : P y p
  := match p with
  | idpath ⇒ u
```

```
  end.
End PathsDefinition.
```

Then we can for instance define transport outside of the module `PathsDefinition`.

```
Definition transport {A} (P : A → Type) {HP : ∀ y, Fibrant (P y)}
     {x y} (p : paths x y) (u : P x) : P y
  := paths_ind (λ y _ ⇒ P y) u p.
```

The fibrancy conditions are then checked automatically each time that we use `transport` (or any other lemma) by typeclass inference.
A nice point with private inductive types is that we retain the computational rule of the inductive type:

$$\texttt{paths\_ind P u idpath} \simeq_\beta \texttt{u}$$

The universe of fibrant types is defined using a record and we include it into `Type` using a coercion:

```
Record TypeF := { TypeF_T : Type ; TypeF_F : Fibrant TypeF_T}.
Coercion TypeF_T : TypeF ↣ Sortclass.
```

The only drawback with this implementation is that the use of a private inductive type breaks some Coq tactics to reason on equality, especially `destruct` and `rewrite`.
With the help of Pédrot, we wrote a small plugin to enforce that `rewrite` uses the lemmas defined from `paths_ind` (*cf.* the file `Overture.v` of the formalization). And we defined a tactic `destruct_path` which tries to emulate `destruct`.

Remark. Contrarily to $\mathbf{MLTT}_2$, here we do not enforce `A` to be fibrant in the formation / introduction / elimination rules of paths but, of course, this additional requirement it can be added.

Remark. We can use the same technique to implement $\mathbf{MLTT}_2^{\mathbf{F}}$. This time, the fibrancy typeclass does not refer to types but to type families:

```
Axiom RFibrant : ∀ {A : Type}, (A → Type) → Type.
Existing Class RFibrant.
```

B. Typing rules for the ad-hoc polymorphism translation

B.1. Booleans in $\lambda\Pi^a_\omega$

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbb{B}^i :_{i+1} \mathcal{U}_i} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{true} :_i \mathbb{B}^i} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{false} :_i \mathbb{B}^i}$$

$$\frac{\Gamma \vdash b :_i \mathbb{B}^i \qquad \Gamma \vdash P : \mathbb{B}^i \to \mathcal{U}_j \qquad \Gamma \vdash t_{\mathbf{true}} :_j P\ \mathbf{true} \qquad \Gamma \vdash t_{\mathbf{false}} :_j P\ \mathbf{false}}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{ret}^j\ P\ \mathbf{then}\ t_{\mathbf{true}}\ \mathbf{else}\ t_{\mathbf{false}} :_j P\ b}$$

B.2. Definition of $\mathbf{TYPE}_i$ internal universes

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{TYPE}_i : \mathcal{U}_0} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{Elt}_i : \mathbf{TYPE}_i \to \mathcal{U}_0}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{b}_i : \mathbf{TYPE}_i} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbf{u}_i : \mathbf{TYPE}_{i+1}}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \pi_{i,j} : \Pi\, A : \mathbf{TYPE}_i.\ (\mathbf{Elt}_i\ A \to \mathbf{TYPE}_j) \to \mathbf{TYPE}_{\max(i,j)}}$$

$$\frac{\Gamma \vdash P : \mathbf{TYPE}_0 \to \mathcal{U}_j \qquad \Gamma \vdash \mathbf{b}'_0 : P\ \mathbf{b}_0 \qquad \Gamma \vdash \pi'_{0,0} : H_{0,0}}{\Gamma \vdash \mathbf{TYPE\_rec}_0(P, \mathbf{b}'_0, \pi'_{0,0}) : \Pi\, a : \mathbf{TYPE}_0.\ P\ a}$$

$$\frac{\begin{array}{c}\Gamma \vdash P : \mathbf{TYPE}_{i+1} \to \mathcal{U}_j \qquad \Gamma \vdash \mathbf{b}'_{i+1} : P\ \mathbf{b}_{i+1} \\ \Gamma \vdash \mathbf{u}'_i : P\ \mathbf{u}_i \qquad \Gamma \vdash \pi'_{0,0} : H_{0,0} \qquad \dots \qquad \Gamma \vdash \pi'_{i+1,i+1} : H_{i+1,i+1}\end{array}}{\Gamma \vdash \mathbf{TYPE\_rec}_{i+1}(P, \mathbf{b}'_{i+1}, \pi'_{0,0}, \dots, \pi'_{i+1,i+1}, \mathbf{u}'_i) : \Pi\, a : \mathbf{TYPE}_{i+1}.\ P\ a}$$

where H_{i_0,i_1} is:

$\Pi\,(a : \mathbf{TYPE}_i)\,(b : \mathbf{Elt}_i\ a \to \mathbf{TYPE}_i).\ P\ a \to (\Pi\, x : \mathbf{Elt}_i\ a.\ P\ (b\ x)) \to P\ (\pi_{i,i}\ a\ b)$ when $i_0 = i_1 = i$

$\Pi\,(a : \mathbf{TYPE}_{i_0})\,(b : \mathbf{Elt}_{i_0}\ a \to \mathbf{TYPE}_{i_1}).\ \Pi\, x : (\mathbf{Elt}_{i_0}\ a.\ P\ (b\ x)) \to P\ (\pi_{i_0,i_1}\ a\ b)$ when $i_0 < i_1$

$\Pi\,(a : \mathbf{TYPE}_{i_0})\,(b : \mathbf{Elt}_{i_0}\ a \to \mathbf{TYPE}_{i_1}).\ P\ a \to P\ (\pi_{i_0,i_1}\ a\ b)$ when $i_0 > i_1$

and where in dots of last rule there are all π'_{i_0,i_1} such that $\max(i_0, i_1) = i + 1$.

$$
\begin{array}{ll}
\mathbf{Elt}_i\ \mathbf{b}_i & \simeq_\beta\ \mathbb{B} \\
\mathbf{Elt}_{i+1}\ \mathbf{u}_i & \simeq_\beta\ \mathbf{TYPE}_i \\
\mathbf{Elt}_k\ (\pi_{i,j}\ a\ b) & \simeq_\beta\ \prod x : \mathbf{Elt}_i\ a.\ \mathbf{Elt}_j\ (b\ x) \\
\mathbf{TYPE\_rec}_i(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ \mathbf{b}_i & \simeq_\beta\ \mathbf{b}'_i \\
\mathbf{TYPE\_rec}_{i+1}(P, \mathbf{b}'_{i+1}, \pi'_{0,0}, \dots, \mathbf{u}'_i)\ \mathbf{u}_i & \simeq_\beta\ \mathbf{u}'_i \\
\mathbf{TYPE\_rec}_i(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ (\pi_{i,i}\ a\ b) & \simeq_\beta\ \pi'_{i,i}\ a\ b\ (\mathbf{TYPE\_rec}_i(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ a) \\
 & \qquad (\lambda x.\ \mathbf{TYPE\_rec}_i(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ (b\ x)) \\
\mathbf{TYPE\_rec}_i(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ (\pi_{j,i}\ a\ b) & \simeq_\beta\ \pi'_{j,i}\ a\ b\ (\lambda x.\ \mathbf{TYPE\_rec}_i(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ (b\ x)) \\
\mathbf{TYPE\_rec}_i(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ (\pi_{i,j}\ a\ b) & \simeq_\beta\ \pi'_{i,j}\ a\ b\ (\mathbf{TYPE\_rec}_i(P, \mathbf{b}'_i, \pi'_{0,0}, \dots)\ a)
\end{array}
$$

where $i > j$ in the two last rules.

Bibliography

[ABC+18] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In *Interactive Theorem Proving*, Lecture Notes in Computer Science, pages 20–39. Springer, Cham, July 2018.

[ACD+18] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In *International Conference on Foundations of Software Science and Computation Structures*, pages 293–310. Springer, 2018.

[ACK16] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending Homotopy Type Theory with Strict Equality. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France*, volume 62 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[ACS15] Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in Homotopy Type Theory. *arXiv:1504.02949 [cs, math]*, April 2015.

[Ada06] Robin Adams. Pure type systems with judgemental equality. *Journal of Functional Programming*, 16(2):219–246, January 2006.

[AdK15] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. Microsoft Research, https://leanprover. github. io/tutorial/tutorial. pdf, 2015.

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with Imperative Features and its Application to SAT Verification. In *Interactive Theorem Proving*, July 2010.

[AK16a] Thorsten Altenkirch and Ambrus Kaposi. Normalisation by evaluation for dependent types. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 52. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[AK16b] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 18–29. ACM, 2016.

[Alt99] Thorsten Altenkirch. Extensional Equality in Intensional Type Theory. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 412–420. IEEE Computer Society, 1999.

[ALV17] Benedikt Ahrens, Peter LeFanu Lumsdaine, and Vladimir Voevodsky. Categorical structures for type theory in univalent foundations. *arXiv:1705.04310 [cs, math]*, May 2017.

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages Meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68. ACM, 2007.

[AS12] Andreas Abel and Gabriel Scherer. On irrelevance and algorithmic equality in predicative type theory. *arXiv preprint arXiv:1203.4716*, 2012.

[Ass14] Ali Assaf. A calculus of constructions with explicit subtyping. In *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39, 2014.

[Bar99] Bruno Barras. *Auto-Validation d'un Système de Preuves Avec Familles Inductives*. PhD Thesis, 1999.

[BCH14] Marc Bezem, Thierry Coquand, and Simon Huber. A Model of Type Theory in Cubical Sets(BCH). In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–128, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BCH17] Marc Bezem, Thierry Coquand, and Simon Huber. The univalence axiom in cubical sets (BCH 2). *arXiv:1710.10941 [cs, math]*, October 2017.

[BG05] Bruno Barras and Benjamin Grégoire. On the role of type decorations in the calculus of inductive constructions. In *International Workshop on Computer Science Logic*, pages 151–166. Springer, 2005.

[BGL+16] Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Mike Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT Library: A formalization of homotopy type theory in Coq. *arXiv:1610.04591 [cs, math]*, October 2016.

[BJP10] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In *ACM Sigplan Notices*, volume 45, pages 345–356. ACM, 2010.

[BL11] Jean-Philippe Bernardy and Marc Lasson. Realizability and parametricity in pure type systems. In *International Conference on Foundations of Software Science and Computational Structures*, pages 108–122. Springer, 2011.

[BPT17] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Certified Programs and Proofs (CPP 2017)*, pages 182 – 194, Paris, France, January 2017.

[BT17] Simon Boulier and Nicolas Tabareau. Model structure on the universe in a two level type theory. 2017.

[Cap17] Paolo Capriotti. Models of Type Theory with Strict Equality. *arXiv:1702.04912 [cs]*, February 2017.

[CCHM16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A constructive interpretation of the univalence axiom (CCHM). *arXiv:1611.02108 [cs, math]*, November 2016.

[CH86] Thierry Coquand and Gérard Huet. *The Calculus of Constructions*. PhD Thesis, INRIA, 1986.

[Cha09] James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009.

[CHM18] Thierry Coquand, Simon Huber, and Anders Mörtberg. On Higher Inductive Types in Cubical Type Theory (CCHM2). *arXiv:1802.01170 [cs, math]*, February 2018.

[Cub] Cubical Type Theory.

[DF92] Oliver Danvy and Andrzex Filinski. Representing control: A study of the CPS transformation. *Mathematical structures in computer science*, 2(4):361–391, 1992.

[DN05] Olivier Danvy and Lasse R. Nielsen. CPS transformation of beta-redexes. *Information Processing Letters*, 94(5):217–224, 2005.

[Dyb95] Peter Dybjer. Internal type theory. In *International Workshop on Types for Proofs and Programs*, pages 120–134. Springer, 1995.

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65(2):525–549, 2000.

[Geu92] Herman Geuvers. The Church-Rosser property for beta eta-reduction in typed lambda-calculi. In *Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium On*, pages 453–460. IEEE, 1992.

[GG08] Nicola Gambino and Richard Garner. The identity type weak factorisation system. *Theoretical Computer Science*, 409(1):94–109, December 2008.

[Gil] Gaëtan Gilbert. Coq with SProp.

[Gog05] Healfdene Goguen. Justifying Algorithms for $b\eta$-Conversion. In *Foundations of Software Science and Computational Structures*, Lecture Notes in Computer Science, pages 410–424. Springer, Berlin, Heidelberg, April 2005.

[Her09] Hugo Herbelin. On a few open problems of the Calculus of Inductive Constructions and on their practical consequences, 2009.

[Hir09] Philip S. Hirschhorn. *Model Categories and Their Localizations*. Number 99. American Mathematical Soc., 2009.

[Hof97] Martin Hofmann. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

[Hof12] Martin Hofmann. *Extensional Constructs in Intensional Type Theory*. Springer Science & Business Media, 2012.

[Hov07] Mark Hovey. *Model Categories*. Number 63. American Mathematical Soc., 2007.

[HS97] Martin Hofmann and Thomas Streicher. Lifting Grothendieck Universes, 1997.

[HS98] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. *Twenty-five years of constructive type theory (Venice, 1995)*, 36:83–111, 1998.

[Hub] Simon Huber. *Cubical Interpretations of Type Theory - Huber Thesis*. PhD thesis.

[JLP+16] Guilhem Jaber, Gabriel Lewertowski, Pierre-Marie Pédrot, Matthieu Sozeau, and Nicolas Tabareau. The definitional side of the forcing. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 367–376. ACM, 2016.

[Kap] Ambrus Kaposi. Tt-in-tt.

[Kap17] Ambrus Kaposi. *Type Theory in a Type Theory with Quotient Inductive Types*. Thesis (University of Nottingham only), March 2017.

[KL12] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. *arXiv:1209.6336 [cs]*, September 2012.

[Las12] Marc Lasson. *Réalisabilité et Paramétricité Dans Les Systèmes de Types Purs*. PhD Thesis, Ecole normale supérieure de lyon-ENS LYON, 2012.

[Las14] Marc Lasson. Canonicity of Weak ω-groupoid Laws Using Parametricity Theory. *Electronic Notes in Theoretical Computer Science*, 308:229–244, October 2014.

[Li15] Nuo Li. *Quotient Types in Type Theory*. PhD Thesis, University of Nottingham, 2015.

[LK] Marc Lasson and Chantal Keller. ParamCoq.

[LOPS18] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal Universes in Models of Homotopy Type Theory. January 2018.

[Lum11] Peter LeFanu Lumsdaine. Model structures from higher inductive types. 2011.

[Luo89] Zhaohui Luo. ECC, an extended calculus of constructions. In *Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium On*, pages 386–395. IEEE, 1989.

[MM12] Saunders MacLane and Ieke Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer Science & Business Media, 2012.

[NF13] Fredrik Nordvall Forsberg. *Inductive-Inductive Definitions*. PhD thesis, Swansea University, 2013.

[NLaa] nLab, Categorical models of dependent types.

[NLab] nLab, Homotopy Type System.

[NPS90] Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory*, volume 200. Oxford University Press Oxford, 1990.

[Nuy18] Andreas Nuyts. Robust Notions of Contextual Fibrancy. *Presentation at HoTT/UF '18 workshop*, 2018.

[OP17] Ian Orton and Andrew M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. *arXiv:1712.04864 [cs]*, December 2017.

[OP18] Ian Orton and Andrew Mawdesley Pitts. Code supporting Axioms for Modelling Cubical Type Theory in a Topos, April 2018.

[Our05] Nicolas Oury. Extensionality in the calculus of constructions. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293. Springer, 2005.

[Ṕ] Pierre-Marie Pédrot. Coq Forcing.

[Pol92] Randy Pollack. Typechecking in pure type systems. In *WORKSHOP ON*, page 271, 1992.

[Pro13] The Univalent Foundations Program. Homotopy Type Theory: Univalent Foundations of Mathematics. *arXiv:1308.0729 [cs, math]*, August 2013.

[Rey83] John C. Reynolds. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[Sat17] Christian Sattler. The Equivalence Extension Property and Model Structures. *arXiv:1704.06911 [math]*, April 2017.

[SH12] Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22(2):153–180, 2012.

[Soz07] Matthieu Sozeau. Program-ing Finger Trees in Coq. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, pages 13–24, New York, NY, USA, 2007. ACM.

[Str91] Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness and Independence Results*. Progress in Theoretical Computer Science. Birkhäuser Basel, 1991.

[SU06] Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149. Elsevier, 2006.

[Swa16] Andrew Swan. An Algebraic Weak Factorisation System on 01-Substitution Sets: A Constructive Proof. *Journal of Logic and Analysis*, 2016.

[Tea] Coq Development Team. The Coq proof assistant reference manual.

[Voe13] Vladimir Voevodsky. A simple type system with two identity types, 2013.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

[WST18] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. Eliminating Reflection from Type Theory: To the Legacy of Martin Hofmann. July 2018.

[ZDK+15] Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *Journal of Functional Programming*, 25, 2015.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Etendre la théorie des types à l'aide de modèles syntaxiques

Mots clés : théorie des types, modèles, traduction de programme, métaprogrammation, théorie des types homotopique, modèle cubique

Résumé :

Cette thèse s'intéresse à la métathéorie de la théorie des types intuitionniste. Les systèmes que nous considérons sont des variantes de la théorie des types de Martin-Löf ou du Calcul des Constructions, et nous nous intéressons à la cohérence de ces systèmes ou encore à l'indépendance d'axiomes par rapport à ces systèmes. Le fil rouge de cette thèse est la construction de modèles syntaxiques, qui sont des modèles qui réutilisent la théorie des types pour interpréter la théorie des types.

Dans une première partie, nous introduisons la théorie des types à l'aide d'un système minimal et de plusieurs extensions potentielles

Dans une seconde partie, nous introduisons les modèles syntaxiques donnés par traduction de programme et donnons plusieurs exemples.

Dans une troisième partie, nous présentons Template-Coq, un plugin de métaprogrammation pour Coq. Nous montrons comment l'utiliser pour implémenter directement certains modèles syntaxiques.

Enfin, dans une dernière partie, nous nous intéressons aux théories des types à deux égalités : une égalité stricte et une égalité univalente. Nous proposons une relecture des travaux de Coquand et. al. et Orton et Pitts sur le modèle cubique en introduisant la notion de fibrance dégénérée.

Title : Extending type theory with syntactic models

Keywords : type theory, models, program translation, metaprogramming, homotopy type theory, cubical model

Abstract :

This thesis is about the metatheory of intuitionnistic type theory. The considered systems are variants of Martin-Löf type theory or of Calculus of Constructions, and we are interested in the coherence of those systems and in the independence of axioms with respect to those systems. The common theme of this thesis is the construction of syntactic models, which are models reusing type theory to interpret type theory.

In a first part, we introduce type theory by a minimal system and several possible extensions.

In a second part, we introduce the syntactic models given by program translation and give several examples.

In a third part, we present Template-Coq, a plugin for metaprogramming in Coq. We demonstrate how to use it to implement directly some syntactic models.

Last, we consider type theories with two equalities: one strict and one univalent. We propose a re-reading of works of Coquand et. al. and of Orton and Pitts on the cubical model by introducing degenerate fibrancy.