THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE NANTES
COMUE UNIVERSITÉ BRETAGNE LOIRE

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

Par
Théo WINTERHALTER

Formalisation and Meta-Theory of Type Theory

Thèse présentée et soutenue à l'Université de Nantes, le 18 septembre 2020
Unité de recherche : LS2N

Rapporteurs avant soutenance :

Andrej BAUER	Professor, University of Ljubljana
Herman GEUVERS	Professor, Radboud University Nijmegen

Composition du Jury :

Président :	Gilles DOWEK	Directeur de recherche, Inria
Examinateurs :	Hugo HERBELIN	Directeur de recherche, Inria
	Assia MAHBOUBI	Chargée de recherche, Inria Rennes
	Bas SPITTERS	Associate professor, Aarhus University
Dir. de thèse :	Nicolas TABAREAU	Directeur de recherche, Inria Rennes
Co-dir. de thèse :	Matthieu SOZEAU	Chargé de recherche, Inria Rennes

Invité(s) :

Andreas ABEL	Professor, Chalmers / Göteborg University

PhD Thesis

Formalisation and Meta-Theory of Type Theory

Théo Winterhalter

Acknowledgements

(There is no real order in which I'm citing people!)

You are about to read my thesis which I wrote during lockdown, an experience that I survived together with Laura whom I'd like to thank deeply.

I would of course like to thank my family who have been a great support in my life: my mom, Lancelot, Tess, Martin, Elena and my grandmother who left us before I could finish this work.

This work has had many readers helping me find bugs and typos hidden—or not so hidden—inside. I'd like to thank Meven, Loïc, Kenji, Xavier, Gaëtan and Matthieu for their useful remarks, and particularly Pierre who helped me greatly on writing my introductory chapters and Andreas who gave me many pointers on how to improve this document. Regarding the manuscript, I would like to thank the reviewers Andrej and Herman for accepting to spend their time reading this long material and for the useful comments they gave me. I would also of course like to thank the whole jury with whom I had nice interactions in the past and hopefully in the future: Andreas, Assia, Bas, Gilles and Hugo.

I am very grateful to Nicolas and Matthieu for advising me for those three years and even before when I was their intern. During my stay in Nantes I had very fruitful and interesting interactions with everyone at the Gallinette team (past and present, even briefly, even never): Xavier, Meven, Loïc, Joachim, Gaëtan, Simon, Marie, Kenji, Julien, Assia, Guillaume, Hervé, Rémi, Nicolas, Matthieu, Pierre-Marie, Guilhem, Ambroise, Étienne, Pierre, Maxime, Igor, Kevin, Benedikt, Danil, Antoine, Enzo, Lucas and Cyril. A special mention for Ambroise whom I thank for his thanking me in his own thesis with the goal of having me thank him in return.

I would like to thank the teachers that led me to where I am today, namely M. Marizy, M. Lyx, M. Gonzales, M. Marino, Ms. Chireux, and M. Alessandri. I would also like to thank Paul who coached me during my preparatory class and has been a very good friend of mine ever since. I would like to thank everyone I met at ENS Cachan, in particular my teachers (including teachers at the MPRI) that were a great influence on me. I will mention my friends Mathias, Léo (and the other Léos), Théis, Félicien, Itsaka, Pierre-Léo, Mathieu, Victor, Olivier, Julien, Pierre, Thomas, Charlie, Damien, Paul and François.

I would like to thank Andreas and everyone I met at Chalmers who made me discover dependent types and Sweden: Andrea, Anders, Inari, Jesper, Vincent, Hans, Simon, Bassel, Guilhem, Peter, Ana, Thierry, Nils Anders, Léo, Victor and Paul. Thanks to everyone I met at Paris 7: Pierre, Théo, Cyril, Étienne, Maxime, Jovana, Zeinab, Kostia. I would like to thank Andrej and everyone I met at Ljubljana (or from Ljubljana): Philipp, Niels, Brett, Matija, Alex, Anja and Peter.

I must have forgotten people, so I thank them, they will know.

…and you! (Please open a pull-request.)

How to read this thesis

I believe that a document of this size will be more often viewed on computer than on paper. This how I personally view most papers I read anyway. As such, I believe that the document should be adapted for such a medium, unlike most LaTeX documents. While trying to make my own kind of document, I stumbled upon the great kaobook class which corresponds almost exactly to what I was looking for: citations, notes, reminders can be put in a large margin.

Citations like [1] will go in the margin, as well as in the Bibliography. The margin also contain side notes[1], replacing the usual footnotes.

[1]: Boulier et al. (2017), 'The Next 700 Syntactical Models of Type Theory'

1: No more going back and forth between the end and top of the page.

I will also use margin notes that are not anchored in the main document but are usually relevant to the paragraph or figure on their left.

Margin notes are not placed entirely automatically. I tend to adjust the height so it is at the right place.

Theorem 0.1 *Theorems and definitions will be in these yellow boxes.*

I will also use the margin to set up reminders in green boxes. Their purpose is to avoid the reader having to go back to a previous chapter they might not have even read and still grasp the meaning of the statements at hand.

Reminder: of something

Body of the reminder.

On the other hand, I will sometimes refer to concepts for the first time without taking the time to introduce them because they are secondary to the main point. In some cases those will be accompanied by a short definition on the side, in a yellowish box.

Definition: of something else

This is supposedly a definition

Introduction et résumé en français

This chapter is the French translation of Chapter 1 (Introduction). You can also skip to the table of contents.

La théorie des types se trouve à l'interface entre programmation et logique formelle, les deux mondes se nourissant l'un l'autre. Des assistants de preuve peuvent être basés dessus, ce qui en fait également des langages de programmation à part entière, avec l'avantage supplémentaire qu'ils peuvent produire des programmes certifiés.

Mon intérêt principal réside dans l'étude de la théorie des types tout en s'appuyant sur les outils qu'elle fournit : j'étudie la théorie des types *au sein même* de la théorie des types. Ainsi, je me suis concentré sur la formalisation de la théorie des types dans l'assistant de preuve Coq [2] et particulièrement sur deux points :

[2] : The Coq development team (2020), *The Coq proof assistant reference manual*

- Comment peut-on transformer efficacement une preuve utilisant une notion très forte d'égalité appelée *reflection*, en une preuve s'appuyant sur une notion très faible d'égalité ?
- Comment peut-on accroître la confiance dans notre système, dans mon cas Coq, en utilisant ce système lui-même comme cadre pour l'étudier ?

La *reflection* et la notion d'égalité faible sont définies en Sous-section 'Extensional Type Theory' et Sous-section 'Weak Type Theory' respectivement, les deux dans le Chapitre 6.

Contributions. Mes contributions se trouvent principalement dans la Partie 'Elimination of Reflection' page 96 et la Partie 'A verified type-checker for Coq, in Coq' page 128, correspondant aux publications suivantes [3-5]. Bien que les chapitres précédant ces deux parties soient principalement introductifs et correspondent à un état de l'art partiel, ils contiennent en réalité d'autres contributions, à savoir le travail que j'ai fait avec Andrej Bauer sur le modèle cardinal dans le Chapitre 8 (Models of type theory) actuellement non publié, et le travail avec Andrej Bauer et Philipp Haselwarter sur la formalisation de la théorie des types appelée `formal-type-theory` [6] et que je présente brièvement dans le Chapitre 9 (Syntax and formalisation of type theory).

[3] : Winterhalter et al. (2019), 'Eliminating Reflection from Type Theory'
[4] : Sozeau et al. (2020), 'Coq Coq correct! verification of type checking and erasure for Coq, in Coq'
[5] : Sozeau et al. (2020), 'The MetaCoq Project'

[6] : Bauer et al. (2016), *Formalising Type Theory in a modular way for translations between type theories*

1 Assistants à la preuve

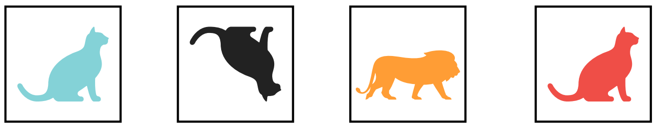
On dit souvent "assistant de preuve", et je vais également le faire, mais il semblerait que la formulation "assistant à la preuve" soit plus correcte.

L'un de mes objectifs est d'améliorer et de mieux comprendre les assistants de preuve, mais qu'est-ce qu'un assistant de preuve ? Je pense que nous pouvons les voir comme des *chatbots* – c'est-à-dire des programmes avec lesquels vous pouvez converser – qui sont là pour vous aider à énoncer et à prouver des théorèmes. Ils ne sont pas particulièrement intelligents et ne feront pas le travail pour vous, ils sont très pointilleux car ils ne comprennent pas toujours ce que vous dites et vous devez être très précis ou ils vous feront remarquer vos erreurs.

L'utilisateur aura des bulles bleues, légèrement à gauche, tandis que l'assistant de preuve répondra en vert (réponse positive) ou en rouge (réponse négative) sur le côté droit.

Ici, je représente l'utilisateur à gauche, en conversation avec l'assistant de preuve, à droite, que j'imagine comme un ordinateur portable avec un coq à l'intérieur.

Voyons maintenant en détail comment une telle conversation peut se dérouler.



Cette œuvre d'art moderne sera le cadre de l'échange entre l'utilisateur et l'assistant de preuve, et nous supposerons qu'ils sont tous les deux d'accord sur ce cadre. Nous avons des chats dans des boîtes (bien que l'un des chats soit plutôt un lion), de différentes couleurs et soumis à différentes forces de gravitation.

Quand je dis qu'ils sont d'accord, je veux dire que, en particulier, l'assistant de preuve a des connaissances de base sur les chats et les boîtes, par exemple, vous n'avez pas besoin de montrer qu'un chat est un félin. Il sait également ce que contient chaque boîte, par exemple que la quatrième boîte contient un chat rouge.

Au début, l'utilisateur fait connaître son objectif à l'assistant de preuve, en disant pour quel théorème il a besoin de son aide.

Je veux *prouver* :

« Chaque boîte »

La phrase ci-dessus ne ressemble pas à un énoncé que nous pouvons prouver, et l'assistant de preuve se plaindra à juste titre.

Erreur : Énoncé incomplet.

En effet, nous sommes allé trop vite et avons oublié la moitié de ce que nous voulions dire. Cette fois, on lui donne une phrase complète.

Je veux *prouver* :

« Chaque boîte *est* un félin »

Maintenant, l'énoncé est légèrement incorrect, ou pas assez précis pour notre gentil assistant.

Erreur : Cela n'a pas de sens de comparer des boîtes et des félins !

Nous devons réviser le théorème demandé et réaliser que nous ne voulons pas prouver que toutes les boîtes sont des félins – bien qu'un humain puisse comprendre ce que nous entendons par là – mais que toutes les boîtes contiennent, chacune, un félin.

Je veux *prouver* :

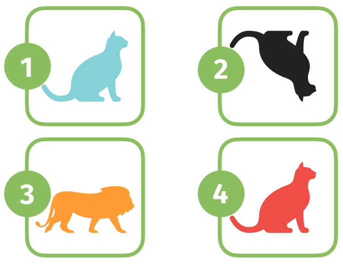
« Chaque boîte *contient* un félin »

Ok ! Comment voulez-vous prouver cet énoncé ?

L'assistant de preuve a maintenant compris ce que nous voulions dire. La partie interactive de la preuve commence maintenant. Nous voulons prouver une propriété sur les boîtes, comme il y en a quatre à considérer, nous pouvons dire à l'assistant de preuve que nous avons l'intention d'examiner chaque cas, un par un.

Fais une analyse de cas sur les boîtes.

En regardant chaque boîte séparément, cela nous laisse avec 4 cas à considérer.



Dans chaque cas, il faut prouver que c'est un félin.

On nous présente quatre cas, et autant d'énoncés à prouver. Nous sommes plutôt paresseux, d'autant plus que les quatre cas seront similaires. Heureusement pour nous, l'assistant de preuve est capable de comprendre que nous voulons traiter plusieurs cas de manière similaire, *et* est également capable de faire des preuves très basiques sans l'intervention de l'utilisateur.

Tous les cas sont triviaux.

Laissez-moi vérifier...

En effet ! L'énoncé est maintenant prouvé.

Une fois la preuve terminée, l'assistant de preuve vous le dit et vous pouvez passer à d'autres théorèmes à prouver. J'ai déjà montré que vous devez être explicite et non ambigu lorsque vous parlez à l'assistant de preuve. Vous devez également être correct. L'assistant de preuve ne se fiera pas aveuglément à ce que vous dites. Considérons maintenant un théorème qui n'est pas prouvable.

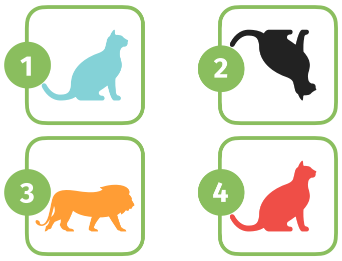
Je veux *prouver* :

« Chaque boîte *contient* un chat »

Ok ! Comment voulez-vous prouver cet énoncé ?

Fais une analyse de cas sur les boîtes.

En regardant chaque boîte séparément, cela nous laisse avec 4 cas à considérer.



Dans chaque cas, il faut prouver que c'est un chat.

Tous les cas sont triviaux.

Laissez-moi vérifier...

Je n'ai pas réussi à prouver tous ces énoncés.

Ici, l'assistant de preuve nous dit que tous les cas ne sont pas triviaux, nous décidons donc de les examiner un par un pour comprendre ce qui ne va pas.

Traitons chaque cas, un par un.

Nous prouvons maintenant le cas 1 :

Veuillez prouver qu'il s'agit d'un chat.

Lorsque nous nous concentrons sur un cas, l'assistant de preuve nous rappelle ce que nous devons prouver spécifiquement. Ici, nous revenons à l'utilisation de notre stratégie antérieure et affirmons simplement que c'est évident.

C'est trivial.

En effet !

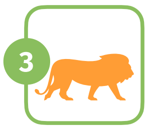
Nous prouvons maintenant le cas 2 :

Veuillez prouver qu'il s'agit d'un chat.

C'est trivial.

En effet !

Nous prouvons maintenant le cas 3 :



Veuillez prouver qu'il s'agit d'un chat.

Laisse tomber.

Cette fois, nous voyons que nous n'avons pas de chat donc nous n'essayons même pas de le prouver. Nous abandonnons la preuve, c'est-à-dire que nous renonçons à prouver un énoncé que nous savons maintenant faux. Une chose importante à noter ici est que l'assistant de preuve ne sait toujours pas que l'énoncé est incorrect, mais seulement que la tentative de preuve était elle-même invalide. Nous pouvons montrer que l'énoncé est faux en prouvant sa négation :

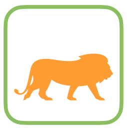
Je veux *prouver* :
« Il y a une boîte qui ne contient pas de chat »

Ok ! Comment voulez-vous prouver cet énoncé ?

La façon la plus simple de prouver que quelque chose existe est de fournir ce que l'on appelle un *témoin* d'existence. Ici, nous recherchons une boîte qui ne contient pas de chat. Il n'y a qu'une seule boîte sans chat, comme nous l'avons découvert lors de notre précédente preuve, la troisième.

Prends la troisième boîte.

Prenons la troisième boîte.



Veuillez prouver que ce n'est *pas* un chat.

C'est trivial.

En effet !

L'énoncé est maintenant prouvé.

Un assistant de preuve nous aide donc à la fois à faire des preuves en construisant des parties de la preuve lui-même et à vérifier qu'elles sont correctes, par exemple en s'assurant qu'on n'a pas oublié un cas où nous avions un lion plutôt qu'un chat.

Les assistants de preuve sont également intéressants et utiles pour d'autres personnes que l'utilisateur qui fait la preuve. Disons que j'ai prouvé un théorème compliqué en utilisant mon assistant de preuve préféré, les personnes qui font confiance à mon assistant de preuve devront simplement lui demander s'il est d'accord avec mon affirmation. En fait, les assistants de preuve génèrent généralement ce que l'on appelle des *certificats* qui peuvent être vérifiés par d'autres sans avoir à comprendre les détails de la preuve.

CERTIFICAT

DE PREUVE

« Chaque boîte *contient* un félin »

L'image peut être trompeuse, je parle vraiment d'un objet concret que les gens peuvent analyser à l'aide de leur propre assistant de preuve pour vérifier qu'une preuve est bien valide.

Quand bien même, pourquoi devrions-nous faire confiance à l'assistant de preuve ? Que pouvons-nous prouver exactement avec des assistant à la preuve ? Et comment pouvons-nous les améliorer ? Une partie de mon travail est centrée sur ces questions, et sur l'étude des assistants de preuve basés sur ce que l'on appelle la théorie des types.

2 Preuves, types et programmes

Afin d'expliquer le titre “Formalisation et méta-théorie de la théorie des types”, j'ai besoin d'expliquer ce qu'est la *théorie des types*. Avant cela, je pense que je dois introduire la *théorie de la preuve* (Chapitre 2) : l'étude des preuves elles-mêmes. En effet, pour construire un assistant de preuve, il faut comprendre ce qu'est une preuve, formellement, c'est-à-dire en termes très précis et non ambigus.

Il s'avère que certaines notions formelles de preuves ont des relations complexes avec la programmation comme nous le verrons dans le Chapitre 3. L'idée est que les programmes peuvent être considérés comme des preuves et les preuves comme des programmes. Par exemple, la preuve qu'un proposition A implique la proposition B est un programme qui prend une preuve de A en entrée et renvoie une preuve de B en sortie. Un tel programme est dit de type $A \longrightarrow B$. Les types sont un moyen de décrire les données manipulées par un programme. Les étudiants sont souvent amenés à écrire la fameuse fonction de test d'une année bissextile qui, étant donnée une année sous forme d'entier (décrit par le type **`int`**), répond s'il s'agit d'une année bissextile ou non. Cette information binaire – oui ou non – est encodée dans ce que nous appelons des booléens (**`bool`**). La correspondance de Curry-Howard indique que les programmes de type A peuvent être considérés comme des preuves de la proposition A et vice-versa.

Danse le Chapitre 4, je décris enfin ce que j'appelle la théorie des types. La théorie des types est un cadre qui tire parti de cette correspondance et est donc au cœur d'assistants de preuve bien connus tels que Coq [2] et Agda [7]. Je suis moi-même un utilisateur de Coq et cette thèse est écrite en mettant l'accent sur cet assistant de preuve et ses fondements. Dans le Chapitre 5, je décris les définitions habituelles qui accompagnent la théorie des types. J'explique comment nous pouvons raisonner dans un tel contexte avec plus que des implications ($\longrightarrow$). C'est là que j'introduis par exemple comment faire un récurrence sur des entier naturels dans la théorie des types.

[2] : The Coq development team (2020), *The Coq proof assistant reference manual*

[7] : Norell (2007), *Towards a practical programming language based on dependent type theory*

Définition : Récurrence sur les entiers naturels

Si $P(n)$ est une propriété sur l'entier naturel n, et si

1. $P(0)$;
2. $P(m)$ implique $P(m+1)$ pour tout m ;

alors on a $P(n)$ pour tout n.

Le Chapitre 6 est dédié à différentes formulations et variantes de la théorie des types. En effet, il n'y a pas qu'une seule théorie de ce genre. Différentes théories des types ont des propriétés différentes, je répertorie les plus importantes tout en disant quelle théorie possède quelle propriété dans un tableau récapitulatif.

Le reste de la partie introductive se concentrera davantage sur la partie “méta-théorie” du titre. Alors que la théorie est l'endroit où l'on effectue nos preuves et où l'on écrit nos programmes, la méta-théorie est l'endroit où l'on raisonne sur la théorie elle-même ! Dans le Chapitre 8, j'explique comment construire ce que l'on appelle des modèles de théorie des types, qui consistent à interpréter la théorie et à lui donner un sens – ou une *sémantique* – à l'intérieur de la méta-théorie. À la fin, je suggère comment la méta-théorie peut même être une forme de théorie des types également. La représentation de la théorie des types en théorie des types est l'objet du Chapitre 9 tandis que le Chapitre 10 se concentre sur la façon dont nous pouvons donner un sens à une théorie des types, en utilisant une autre théorie des types.

3 Élimination de la *reflection*

Ensuite, nous avons la Partie ‘Elimination of Reflection'. Le titre peut fair peur, mais il tombe dans la catégorie “que puis-je prouver avec mon assistant de preuve ?” Et “comment puis-je améliorer les assistants de preuve ?”

La théorie des types est à l'interface de la logique et de la programmation. Traditionnellement, les langages de programmation sont conçus avec l'idée que, à la fin, les programmes seront *exécutés* ou *évalués*. En d'autres termes, les programmes *calculent*. Le calcul est également un élément important de la théorie des types. Le calcul correspond à la simplification des preuves, mais surtout il incarne la notion de *constructivisme*. Supposons que, dans l'exemple précédent, vous prouvez qu'il “existe un chat avec un problème de gravité”, si vous exécutez cette preuve, elle calculera et produira deux informations importantes : qui est le chat fautif – dans ce cas, le noir dans la deuxième boîte – et pourquoi il a des problèmes de gravité (donnée comme une autre preuve qui correspondra à peu près à montrer que le chat est à l'envers).

Le calcul est également utile lors de la réalisation de preuves. Par exemple, $2{+}2$ *vaut* 4, donc pas besoin de transporter des informations concernant la relation entre les deux. $2 + 2$ s'évalue en 4, autrement dit, deux ensembles de deux boîtes à chat correspondent juste à un ensemble de quatre boîtes à chat, et l'assistant de preuve le sait.

Il existe différents degrés de calcul qui peuvent être mis dans une théorie des types, et on peut se demander quels impacts le calcul a sur la théorie. Je présente un spectre de trois théories différentes. D'un côté, une théorie, appelée théorie extentionnelle des types (ETT), qui est très libérale en ce qui concerne le calcul en ce qu'elle identifie les choses qui sont prouvées ou supposées égales (et pas seulement celles qui s'évaluent en un même résultat). Dans un tel contexte, le calcul n'est plus dirigé et nous parlons simplement des choses comme étant *égales*. Ce principe, transformant une preuve en égalité, est appelé *reflection* de l'égalité. L'autre extrême est une théorie, appelée théorie faible des types (WTT), où nous supprimons la notion même de calcul. Au milieu, nous avons la théorie intentionnelle des types (ITT) qui a une notion de calcul "usuelle", définie comme l'évaluation des fonctions. Cette dernière est à la base de Coq et Agda.

Le calcul est *dirigé* : $2 + 2$ se simplifie, ou s'évalue, en 4, et pas l'inverse. $2 + 2$ et $3 + 1$ sont égaux parce que les deux s'évaluent en 4 :

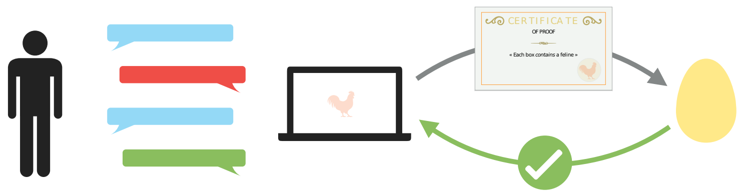
$$2 + 2 \rightarrowtriangle 4 \leftarrowtriangle 3 + 1$$

Ma contribution est une traduction de ETT vers ITT et WTT, montrant que le calcul n'est pas une propriété essentielle – bien que très pratique – de la théorie des types du point de vue logique : en effet, je montre que nous pouvons encore prouver les mêmes énoncés. Cela signifie que nous pouvons raisonner sur la prouvabilité des énoncés dans WTT, qui est plus simple. La traduction se fait dans la théorie des types elle-même – dans ce cas Coq – et fournit ainsi un programme transformant une preuve dans ETT en une preuve dans ITT ou WTT qui peut donc être utilisé pour calculer de nouvelles preuves.

4 Un vérificateur de type pour **Coq**, en **Coq**

Enfin, dans Partie 'A verified type-checker for Coq, in Coq', je répondrai à la question "Comment peut-on augmenter la confiance dans un assistant de preuve ?" En me concentrant une fois de plus sur Coq, je présente une vérification du noyau de Coq, en utilisant Coq lui-même.

Coq dispose d'un noyau, c'est-à-dire un programme assez petit qui vérifie que les preuves sont bien correctes. La partie interactive de l'assistant de preuve, ainsi que l'automatisation – c'est-à-dire ce qui fait que l'assistant de preuve trouve des preuves pour nous – ne font pas partie du noyau et sont plutôt construites au-dessus de lui. Si vous voulez, la partie qui vérifie les certificats est distincte du reste et, en tant que telle, est plus facile à étudier.



L'œuf à droite représente le noyau. L'assistant de preuve lui envoie essentiellement le certificat qu'il a généré après discussion avec l'utilisateur et le noyau le valide (ou non).

Si l'assistant de preuve est défectueux, le certificat sera incorrect et rejeté par le noyau. Tant que le noyau est correct, il n'est pas crucial que le reste soit complètement correct. Certes, c'est plus agréable quand tout est correct, mais seule l'exactitude du noyau est cruciale. Comme je l'ai dit, des efforts sont fournis pour garder le noyau aussi petit que possible afin que des humains puissent l'inspecter et affirmer son exactitude.

Malheureusement, même un assistant de preuve aussi largement utilisé que Coq n'est pas exempt d'erreurs : un bogue critique a été trouvé à peu près chaque année au cours des vingt dernières années. Ceux-ci sont généralement résolus assez rapidement, mais leur simple existence est néanmoins gênante. Une preuve incorrecte peut avoir des conséquences désastreuses. Coq n'est pas seulement utilisé pour prouver des théorèmes mathématiques, mais aussi pour vérifier des programmes et s'assurer qu'ils ne planteront pas ou ne se retrouveront pas dans des configurations indésirables. Ces programmes que nous voulons vérifier sont souvent utilisés dans les parties critiques des logiciels, par exemple dans les pilotes automatiques et les fusées, où la plus petite erreur peut coûter des millions… et des vies.

Je fais partie d'un projet – le projet MetaCoq – qui a pour but de spécifier et vérifier le noyau de Coq lui-même, dans Coq. Cela signifie que nous produisons un noyau séparé, entièrement écrit et vérifié dans Coq qui peut également vérifier les certificats indépendamment.

Lors de la vérification du noyau, nous devons nous fier aux propriétés méta-théoriques de la théorie de Coq. Malheureusement, toutes ne peuvent pas être prouvés dans Coq lui-même. Cela est dû aux théorèmes d'incomplétude de Gödel qui impliquent à peu près qu'une théorie comme celle de Coq ne peut pas prouver sa propre cohérence. Ainsi, nous devons supposer certaines de ces propriétés méta-théoriques, en particulier celles suffisamment fortes pour impliquer la cohérence de Coq, donc qui ne peuvent pas être prouvées. Nous proposons donc un changement de paradigme : à la place d'avoir un noyau relativement petit auquel on fait confiance, nous nous appuyons sur une petite base de propriétés supposées sur la théorie, théorie qui a été bien étudiée au fil des années et qui ne souffre pas des bogues découverts dans l'implantation. Il y a d'autres défis auxquels nous devons faire face, l'un d'entre eux est la preuve que les définitions que nous fournissons pour construire le noyau sont terminantes, cela a été beaucoup plus compliqué que prévu et cela nous a forcés à rendre explicite plusieurs invariants qui sont implicites dans l'implantation actuelle.

Je parlerai des théorèmes d'incomplétude de Gödel dans le Chapitre 2 (Proof theory).

Dans l'ensemble, nous fournissons une spécification de Coq, dans Coq, avec quelques propriétés sur la théorie que nous devons supposer. À partir de cela nous vérifions la correction d'un noyau de Coq alternatif.

Contents

Introduction 1

Type theory is set at the interface between programming and formal logic, feeding off and nourishing both worlds. Proof assistants can be built on it, making them full-fledged programming languages as well, with the added benefit of producing certified programs.

My main interest lies in the study of type theory while relying on the tools it provides: I study type theory *within* type theory. As such I focused on the formalisation of type theory in the Coq proof assistant [2] and particularly on two points:

[2]: The Coq development team (2020), *The Coq proof assistant reference manual*

- ▶ How can we effectively turn a proof using a very strong notion of equality called *reflection*, into a proof relying on a very weak notion of equality?
- ▶ How can we improve the trust in our system, in my case Coq, using this system itself as a framework to study it?

Reflection is defined in Subsection 'Extensional Type Theory', and weak equality in Subsection 'Weak Type Theory', both in Chapter 6.

Contributions. My contributions will mainly be found in Part 'Elimination of Reflection' on page 96, and Part 'A verified type-checker for Coq, in Coq' on page 128, corresponding to the following published articles [3–5]. While the chapters before these two parts are mainly introductory and corresponding to a rough state-of-the-art, they actually contain other contributions, namely work I did with Andrej Bauer on the cardinal model in Chapter 8 (Models of type theory) that we did not publish, and work I did with Andrej Bauer and Philipp Haselwarter on formalising type theory called `formal-type-theory` [6] and that I present briefly in Chapter 9 (Syntax and formalisation of type theory).

[3]: Winterhalter et al. (2019), 'Eliminating Reflection from Type Theory'
[4]: Sozeau et al. (2020), 'Coq Coq correct! verification of type checking and erasure for Coq, in Coq'
[5]: Sozeau et al. (2020), 'The MetaCoq Project'
[6]: Bauer et al. (2016), *Formalising Type Theory in a modular way for translations between type theories*

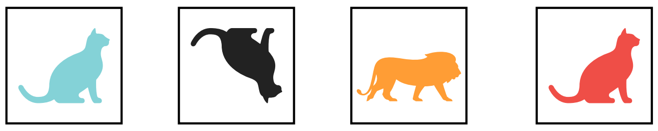
1.1 Proof assistants

One of my goals is improving and understanding better proof assistants, but what is a proof assistant? I think we can see them as chatbots—that is, those programs that you can converse with—that are there to help you assert and prove theorems. They are not particularly smart and will not get the job done for you, but they are very annoying because they do not always understand what you say and you have to be very precise or they will point out your mistakes.

The user will have blue speech bubbles, slightly on the left, while the proof assistant will answer in green (for good) or red (for bad) on the right-hand side.

Here, I represent the user on the left, conversing with the proof assistant, on the right, that I picture as a laptop with a rooster inside.

Let us now see in detail how such a conversation can go.



This piece of modern art will be the setting for the exchange between the user and the proof assistant, and we will assume they both agree on it. We have cats in boxes (although one of the cats is rather a lion), of different colours, and subject to different gravitation forces.

When I say they agree, I mean that, in particular, the proof assistant has basic knowledge of cats and boxes, for instance you don't need to show it that a cat is a feline. It also knows what each box contains, for instance that the fourth box contains a red cat.

At the beginning, the user makes their purpose known to the proof assistant, saying for which theorem they require their assistance.

I want to *prove*:

« Each box »

The sentence above doesn't sound like a statement we can prove, and the proof assistant will rightfully complain.

Error: Incomplete statement.

Indeed, we were too fast and forgot half of what we wanted to say. Next we give it a complete sentence.

I want to *prove*:

« Each box *is* a feline »

This time, the statement is slightly incorrect, or not precise enough for our good assistant.

Error: It does not make sense to compare boxes and felines!

We have to revise our wanted theorem, and realise that we do not want to prove that all boxes are feline, although a human might understand what we mean by that, but that all of the boxes contain, each, a feline.

I want to *prove*:

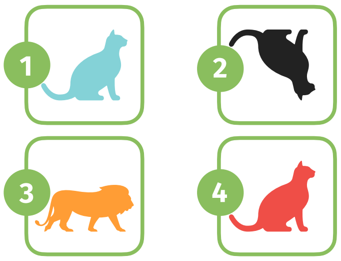
« Each box *contains* a feline »

Ok! How do you want to prove this statement?

The proof assistant has now understood what we meant. The interactive part of the prove now begins. We want to prove a property on the boxes. As there are four to consider, we can tell the proof assistant that we intend to look at each case, one by one.

Do a case analysis on boxes.

Looking at each box separately, this leaves us with 4 cases to consider.



In each case you have to prove you have a feline.

We are presented with four cases, and as many statements to prove. In this case, we are rather lazy, especially since all four cases will be similar. Fortunately for us, the proof assistant is capable of understanding that you want to deal with several cases in similar ways, *and* is also capable of doing very basic proofs without the user.

All cases are trivial.

Let me check that...

Indeed! Your statement is now proven.

Once the proof is complete, the proof assistant tells you so and you can move on to other theorems to prove. I already showed that you have to be explicit and non ambiguous when talking to the proof assistant. You also have to be correct. The proof assistant will not trust blindly in what you say. Let us now consider a theorem that is not provable.

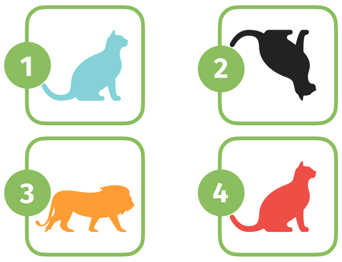
I want to *prove*:

« Each box *contains* a cat »

Ok! How do you want to prove this statement?

Do a case analysis on boxes.

Looking at each box separately, this leaves us with 4 cases to consider.



In each case you have to prove you have a cat.

All cases are trivial.

Let me check that...

I did not succeed in proving all these statements.

Here the proof assistant tells us that not all cases are trivial so we decide to go over them one by one to understand what is wrong.

Focus on each case one by one.

We are now proving case 1:

Please prove that this is a cat.

When we focus on a case, the proof assistant reminds us of what we have to prove specifically. Here we revert to using our earlier strategy and simply say that it is plain to see.

This is trivial.

Indeed!

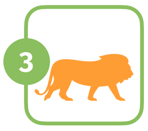
We are now proving case 2:

Please prove that this is a cat.

This is trivial.

Indeed!

We are now proving case 3:



Please prove that this is a cat.

Abort.

This time, we see we do not have a cat so we do not even try to prove it. We abort the proof, that is we give up on proving a statement we know to be wrong now. One important thing to note here is that the proof assistant still does not know the statement is incorrect, only that the proof attempt was itself invalid. We can show it to be incorrect by proving its negation:

I want to *prove*:
« There is a box which doesn't contain a cat »

Ok! How do you want to prove this statement?

The easiest way of proving that something exists is by providing a so-called *witness* of existence. Here we are looking for a box which does not contain a cat. There is only one box with no cat in it, as we found out during our previous proof, the third.

Take the third box.

Taking the third box.

Please prove that this is *not* a cat.

This is trivial.

Indeed!

Your statement is now proven.

A proof assistant is thus both helping us do proofs by building parts of it on its own, and verifying that they are correct, for instance by not forgetting one case where we had a lion rather than a cat.

Proof assistants are also interesting and useful for other people than the user doing the proof. Say that I have proven a complicated theorem using my favourite proof assistant, people who trust my proof assistant will simply have to ask it if it agrees with my claim. In fact, proof assistants usually generate so-called *certificates* that can be checked by others without having to understand the details of the proof.

CERTIFICATE

OF PROOF

« Each box *contains* a feline »

The picture can be misleading, I am really talking about a concrete object that people can run through their own proof assistant to check that a proof is indeed valid.

Even then, why should we trust the proof assistant? What exactly can we prove with them? And how can we improve them? Part of my work is focused on these questions, and on the study of proof assistants based on something called type theory.

1.2 Proofs, types and programs

In order to explain the title “Formalisation and Meta-Theory of Type Theory” I need to explain what *type theory* is. Even before that, I believe I have to introduce *proof theory* (Chapter 2): the study of proofs themselves. Indeed, in order to build a proof assistant one needs to understand what a proof is, formally, *i.e.* in very precise and non-ambiguous terms.

It turns out that certain formal notion of proofs have intricate relations with programming as we shall see in Chapter 3. The idea is that programs can be seen as proofs, and proofs as programs. For instance a proof that a proposition A implies the proposition B is a program that takes a proof of A as input and outputs a proof of B. Such a program is given type $A \to B$. Types are a way of describing the data the programs manipulate. Students will often write the infamous leap year function which, given a year as an integer (described by the type **`int`**), returns whether it is a leap year or not. This binary information—yes or no—is encoded into what we call booleans (**`bool`**). Typing as another use however. The Curry-Howard correspondence states that programs of type A can be seen as proofs of propositon A and vise-versa.

In Chapter 4, I finally describe what I call type theory. Type theory is a framework taking advantage of this correspondence and is thus at the core of well-known proof assistants such as Coq [2] and Agda [7]. I myself am a Coq user and this thesis is written with a focus on this proof assistant and its foundations. In Chapter 5, I describe usual definitions that come with type theory. I explain how we can reason in such a setting with more than implications. This is where I introduce for instance the way of doing induction on natural numbers in type theory.

[2]: The Coq development team (2020), *The Coq proof assistant reference manual*

[7]: Norell (2007), *Towards a practical programming language based on dependent type theory*

Definition: Induction on natural numbers

If $P(n)$ is a property on natural number n, then if we have

1. $P(0)$;
2. $P(m)$ implies $P(m+1)$ for any m;

then we have $P(n)$ for any n.

Chapter 6 is dedicated to different formulations and variants of type theory. Indeed, there is not just one such theory. Different type theories have different properties, I catalogue the most important ones while telling which theory enjoys which property in a summarising table.

The remaining of the introductory part will focus more on the 'meta-theory' part of the title. While the theory is where you conduct your proofs and write your programs, the meta-theory is where you reason about the theory itself! In Chapter 8, I explain how we can build so-called models of type theory, which consist in interpreting the theory and giving it meaning—or *semantics*—inside the meta-theory. At the end, I hint at how the meta-theory can even be a form of type theory itself. The representation of type theory inside itself is the subject of Chapter 9 while Chapter 10 focuses on how we can give meaning to a type theory, using another type theory.

1.3 Elimination of reflection

Next, we have Part 'Elimination of Reflection'. The title can be scary but it falls in the category of "what can I prove with my proof assistant?" and "how can I improve proof assistants?"

Type theory is at the interface of logic and programming. Traditionally, programming languages are designed with the idea that, eventually, programs will be *run*, or *evaluated*. In other words, programs *compute*.

Now, computation is also an important part of type theory. Computation corresponds to the simplification of proofs, but also more importantly it embodies the notion of *constructivism*. Say, in the earlier example, that you prove that “there exists a cat with a gravity issue,” if you execute this proof, it will compute and yield two important informations: who is the guilty cat—in this case the black one in the second box—and why it has gravity issues (as another proof which will roughly correspond to showing the cat is upside down).

Computation is also useful when doing proofs. For instance, $2 + 2$ *is* 4, so no need to carry around any information relating the two. $2 + 2$ will evaluate to 4, so two sets of two cat boxes is really just a set of four cat boxes, and the proof assistant knows it.

Now there are various degrees of computation that can be put inside a type theory, and one might wonder what impacts it has on it. I present a spectrum of three different theories. On one side a theory, called Extensional Type Theory (ETT), which is very liberal with respect to computation in that it identifies any two things that we prove or assume equal, in such a setting, computation is no longer directed and we simply talk about things being *equal*. This principle, turning a proof into an equality, is called *reflection* of equality. The other extreme is a theory, called Weak Type Theory (WTT), where we remove the very notion of computation. In the middle, we have Intensional Type Theory (ITT) which has a ‘regular’ notion of computation, defined as the evaluation of functions. The latter is the base for Coq and Agda.

Computation is *directed*: $2+2$ simplifies, or evaluates, to 4, not the converse. $2+2$ and $3 + 1$ are equal because they both evaluate to 4:

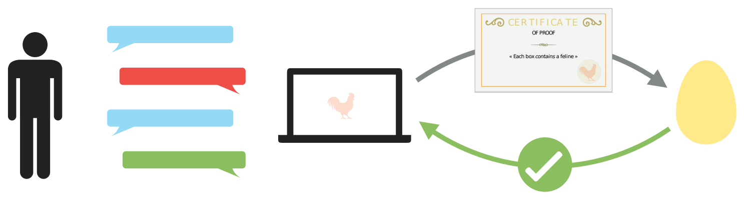
$$2 + 2 \rightarrow 4 \leftarrow 3 + 1$$

My contribution is a translation from ETT to ITT and WTT, showing that computation is not an essential property—albeit highly practical—of type theory from the logical point of view: indeed, I show that we can still prove the same statements. It means we can reason about provability of statements in WTT, which is simpler. The translation is done in type theory itself—in this case Coq—and as such provides a program turning a proof in ETT into a proof in ITT or WTT that can actually be used to compute new proofs.

1.4 A verified type-checker for **Coq**, in **Coq**

Finally, in Part ‘A verified type-checker for Coq, in Coq’, I will address the question “How can we increase the trust in a proof assistant?” Focusing once more on Coq, I present a verification of Coq’s kernel, using Coq itself.

Coq features a kernel, that is a rather small program that checks that proofs are indeed correct. The interactive part of the proof assistant, as well as the automation—*i.e.* the thing that makes the proof assistant find proofs for us—are not part of the kernel and rather built above it. If you want, the part that checks the certificates is separate from the rest and as such is easier to survey.



The egg on the right represents the kernel. The proof assistant essentially sends it the certificate it generated after discussing with the user and the kernel validates it (or not).

If the proof assistant is broken, the certificate will be incorrect and rejected by the kernel. As long as the kernel is correct, it is not crucial that the rest is completely correct. Granted, it is nicer when everything is correct, but only the correctness of the kernel is crucial. As I said, efforts are made to keep the kernel as small as possible so that humans can inspect it and assert of its correctness.

Unfortunately even a proof assistant as widely used as Coq is not exempt from mistakes: one critical bug had been found roughly every year for the past twenty years. These are usually fixed rather quickly but their mere existence is troublesome nonetheless. An incorrect proof can have desastrous consequences. Coq is not only used to prove mathematical theorems, but also to verify programs and make sure they will not crash or end up in unwanted configurations. Such programs that we want to verify are often used in critical parts of softwares, for instance in autopilots and rockets, where the smallest mistake can cost millions… and lives.

I am part of a project—the MetaCoq project—to specificy and verify Coq's kernel itself, within Coq. This means we produce a separate kernel, fully written and verified in Coq that can also check certificates independently.

When verifying the kernel we need to rely on meta-theoretical properties of Coq's theory. Unfortunately, not all of them can be proven within Coq itself. This is because of Gödel's incompleteness theorems which roughly implies that a theory like that of Coq cannot prove its own consistency. As such, we need to assume some of these meta-theoretical properties, especially those strong enough to imply the consistency of Coq, hence which cannot be proven. Thus, we propose a paradigm shift: instead of having a relatively small kernel that is trusted, we rely on a small base of properties assumed on the theory which has been well studied over the years and does not suffer form the shortcomings of the implementation. There are other challenges we must face, one of which is proving that the definitions we provide for the kernel are terminating, this was much more complicated than anticipated and forced us to make explicit a lot of invariants which are kept implicit (and sometimes unknown) in the current implementation.

I will talk about Gödel's incompleteness theorems in Chapter 2 (Proof theory).

All in all, we provide a specification of Coq, within Coq, with a few trusted properties about the theory with which we verify the soundness of an alternative Coq kernel.

Proofs, Types and Programs

Proof theory 2

My work is done in the wide domain of proof theory. Proof theorists are interested in the way to prove things, in the 'proof' object itself, *e.g.* we might want to check that a proof uses some specific rules of deduction. We may also want to prove that a system is not contradictory, *i.e.* that it does not entail both an assertion and its negation. This allows us to understand more about proof reuse, about transfer of a property to another system, or about some intrinsic properties of the proof itself. The study of proofs also makes it possible to define clear systems outlining formally what a proof is and when it is valid. This leads to the notion of certificate that one can check independently. The epitome of this is the ability to write proofs that are checkable by a computer, shifting the trust one needs to put within every proof, to the system which validates them all.

2.1 How to prove something

A social construct?

Before we start proving something, we must know precisely what it is we want to prove. In informal mathematics, the statement will be a sentence, involving concepts that the writer and reader agree on. The proof then consists of a sequence of sentences and argument that convince the same readers.

For instance, the Pythagorean theorem states that "the area of the square whose side is the hypotenuse of a right triangle is equal to the sum of the areas of the squares on the other two sides."

With this definition, a proof is a subjective concept, it depends on the reader's capacity to understand and potentially fill the gap themselves about understood statements and properties. It also usually involves a fair bit of *trusting*: you may not understand a proof, but will believe in the common effort of the community to verify the proof or, even better, reproduce it. As such, *consensus* seems key in the scientific community.

One way to reach consensus much faster is to have statements and proofs really precise and unambiguous, described in formal systems. This approach still has shortcomings, will all readers check every tiny detail of the proof once it is laid out extensively? This runs the risk of having the *idea* lost in a sea of information. To me, this calls for computer-verified proofs—and maybe even automated or computer aided proofs—coming with paper proofs exposing the ideas so that the reader can focus on the interesting part of the proof while trusting only the tool and not the human that used it.

Even then, there is room for question on whether this really constitutes a proof. For instance, how *hard* is it for the computer to *see* that the proof is indeed correct? One definition would be to say, as long as it takes a finite amount of time, it is good, but if it takes ages, we will not have any certainty. In [8], de Bruijn suggests that a proof should be self-evident. You should not have to think for hours before seeing that it is indeed correct, and the same holds for computers.

[8]: De Bruijn (1991), 'A plea for weaker frameworks'

In the remainder of the section I will address the way we *write* statements and proofs.

Formal statements

What do formal statements look like? Probably something like

Since I want to be as general as possible, this talk about formal statements will be pretty informal.

$$\forall n \in \mathbb{N}. \exists m \in \mathbb{R}. f(n) = \mathrm{e}^{\phi(m)} \wedge g(n) > m$$

It involves defined symbols, and logical connectives to make something precise. In particular you will note the universal ($\forall$) and existential ($\exists$) quantifiers, equality ($=$), logical conjunction ($\wedge$) and a comparison operator ($>$). We can assume $\mathbb{N}$, $\mathbb{R}$, f, g, h, ϕ and e to be defined prior to the statement (using similar formalism).

To define this, we give a syntax of propositions, mutually with a syntax of sets on which we want to quantify. Because equality can mention elements however we also have to provide a syntax for those, and maybe one for function symbols.

$$\begin{array}{rcl} P, Q & ::= & \top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \to Q \\ & \mid & \forall x \in E.P \mid \exists x \in E.P \mid u = v \\ E & ::= & \mathbb{N} \mid \mathbb{R} \mid \dots \\ u, v & ::= & x \mid \mathrm{e}^u \mid u + v \mid f(u) \mid \dots \\ f, g, h & ::= & \dots \end{array}$$

Here $P \to Q$ denotes 'P implies Q', often written $P \implies Q$. In my domain things are different and we use a single arrow.

This presentation of syntax is called Backus–Naur Form (BNF) and it says that there are expressions that we will write with P or Q that can be either $\top$ or $\bot$ or $P \wedge Q$ where P and Q are both of the same form, and so on. For instance, $\bot \wedge (\top \vee (\top \to \bot))$ is one such expression.

Parentheses are not part of the syntax but are there to lift any ambiguity and distinguish $P \wedge (Q \vee R)$ from $(P \wedge Q) \vee R$ for example.

Here P, Q are propositions, E stands for a set, u, v are mathematical expressions whereas f, g, h are function symbols.

Logical connectives ($\wedge, \vee, \forall, \dots$) and operations ($+, \times, \dots$) are the building blocks of statements, as such we call them *constructors*.

Coming up with a correct syntax like those can be pretty painful so formalisms tend to be as minimal as possible, the other advantage being that it is much easier to reason on the statements when there are not hundreds of syntactical constructs.

Of course, giving a syntax of statements is not enough. We must give these symbols a semantics (*i.e.* a meaning) to know what it means to prove them. For instance, we might want to specify that the symbol $+$ has the properties which are expected of addition.

Inference rules

We need to define what it means to prove a statement given some hypotheses. For instance we have

$$A, B \vdash A \wedge B$$

to denote the fact that A and B as hypotheses, *entail* the proposition $A \wedge B$ (read '*A and B*'). This is called a *judgement*: $A, B, C \vdash D$ means that, *assuming A, B and C*, then D holds.

$$\begin{array}{rcl} \Gamma, \Delta & ::= & \bullet \mid \Gamma, P \\ \mathcal{J} & ::= & \Gamma \vdash P \end{array}$$

$\mathcal{J}$ here stands for *judgement*.

We can now move on to the notion of *inference* rule. They are what defines the logic that we consider, they dictate what judgements can be *derived*—*i.e.* proven—and how. The simplest of rules usually is the so-called *axiom* rule stating that assuming A, you can prove A.

$$\frac{}{A \vdash A}$$

or more generally

$$\frac{A \in \Gamma}{\Gamma \vdash A}$$

The line separates one judgement below, the conclusion, to one, several or possibly no judgements above. They represent requirements to conclude the lower part. In this case, one does not need to assume anything to conclude that A entails A. The condition $A \in \Gamma$ is not a judgement itself but really a side condition. Going back to conjunction, the rule[1] to prove one is the following.

1: It is actually one of many possible rules. It all depends on the logic.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

That is to say, to prove $A \wedge B$ (under hypotheses Γ), it *suffices* to prove A and B (under the same hypotheses). This is called an *introduction* rule because it allows us to introduce a connective in the conclusion. Sometimes, we want to be able to conclude something from a complex assumption like $A \wedge B$, this is instead called an *elimination* rule.

Once again, this is one of many ways to proceed. This setting corresponds to *natural deduction* and was introduced by Gentzen; he also introduced *sequent calculus* where there are no elimination rules, but introduction rules on the left and on the right.

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad\qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

If you can prove $A \wedge B$ then you can prove A (you can also prove B but that is another rule). We then have similar rules for disjunction ($A \vee B$ reads '*A or B*').

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash P \qquad \Gamma, B \vdash P}{\Gamma \vdash P}$$

To prove $A \vee B$ you only need to prove either A or B, hence the two introduction rules. On the contrary if you want to prove P knowing $A \vee B$, you have to provide a proof for the two different cases: either A holds, or B holds, in both instances P should hold.

Amongst the most important rules are those related to implication.

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B} \qquad \frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

They are interesting because they show a level of interaction between the entailement and the implication. To prove that $A \to B$, you only assume A and show B. Moreover, if you know $A \to B$ and A, then you have B. This last rule is called the *modus ponens*, that is the elimination of the implication.

The purpose of these rules is to build so called *derivations* which correspond to proofs that a judgement holds. The proof is complete if there are no dangling judgements at the top. For instance the judgement $A \vdash (A \to B) \to B$ is proven with the derivation:

$$\cfrac{\cfrac{\cfrac{}{A, A \to B \vdash A \to B} \qquad \cfrac{}{A, A \to B \vdash A}}{A, A \to B \vdash B}}{A \vdash (A \to B) \to B}$$

We are using the introduction and elimination rules for implication, as well as the axiom rule twice.

2.2 Proof frameworks

There are many proof frameworks and logics and I will not review all of them, but I will present the most relevant ones to the rest of my work.

Intuitionistic logic

Intuitionistic logic corresponds to the general logical setting of my work. Gentzen describes it using a natural deduction system called NJ [9].

[9]: Gentzen (1935), 'Untersuchungen über das logische Schließen. I'

The syntax is given by

$$\begin{array}{rcl} P, Q, A, B & ::= & \top \mid \bot \mid P \wedge Q \mid P \vee Q \mid P \to Q \mid \neg P \\ \Gamma, \Delta & ::= & \bullet \mid \Gamma, P \\ \mathcal{J} & ::= & \Gamma \vdash P \end{array}$$

which corresponds to *propositional logic*.

I will give the rules in two separate bundles. We first have the logical rules—*i.e.* those that pertain to the logical constructors of propositional logic:

$$\frac{A \in \Gamma}{\Gamma \vdash A}(ax) \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}(\wedge I) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}(\wedge E_1)$$

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}(\wedge E_2) \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}(\vee I_1) \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}(\vee I_2)$$

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}(\vee E) \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}(\to I)$$

$$\frac{\Gamma \vdash A \to B \qquad \Gamma \vdash A}{\Gamma \vdash B}(\to E) \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}(\neg I)$$

$$\frac{\Gamma \vdash \neg A \qquad \Gamma \vdash A}{\Gamma \vdash \bot}(\neg E) \qquad \frac{}{\Gamma \vdash \top}(\top I) \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}(\bot E)$$

Next to the rules I write their name in parentheses. ax stands for 'axiom'. The Is and Es in rules indicate whether each rule is an introduction rule or an elimination rule.

The rest are structural rules that allow us to weaken the hypotheses (*i. e.* add a new, unused hypothesis in the context), reorder them, and remove duplicates.

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}(W) \qquad \frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}(C) \qquad \frac{\Gamma, A, B, \Delta \vdash A}{\Gamma, B, A, \Delta \vdash A}(P)$$

W stands for weakening, C for contraction and P for permutation.

This logic is *constructive* in that, from every proof derivation from no hypotheses one can extract a proof where the last rule is an introduction rule. For instance, every proof of $\vdash A \vee B$ must contain either a proof of $\vdash A$ or a proof of $\vdash B$. This is obtained by a process known as *cut-elimination* which removes all elimination rules that would be the last rule. For instance the modus ponens $(\to E)$ can be removed when it follows an introduction of implication $(\to I)$ rule, I will show an example for which we set $P \coloneqq A \to \neg\neg A$ and prove $\vdash P \wedge P$.

Notice that these are proofs in the empty context.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{P \vdash P}(ax) \qquad \dfrac{}{P \vdash P}(ax)}{P \vdash P \wedge P}(\wedge I)}{\vdash P \to P \wedge P}(\to I) \qquad \dfrac{\dfrac{\dfrac{}{A, \neg A \vdash A}(ax)}{A \vdash \neg\neg A}(\neg I)}{\vdash P}(\to I)}{\vdash P \wedge P}(\to E)$$

Now the idea is that, to remove the elimination of implication $(\to E)$ which is the last rule, we replace the axiom rules corresponding to $P \vdash P$ in the derivation above $\vdash P \to P \wedge P$ (in yellow) by the derivation of $\vdash P$ on the right (in blue).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, \neg A \vdash A}(ax)}{A \vdash \neg\neg A}(\neg I)}{\vdash P}(\to I) \qquad \dfrac{\dfrac{\dfrac{}{A, \neg A \vdash A}(ax)}{A \vdash \neg\neg A}(\neg I)}{\vdash P}(\to I)}{\vdash P \wedge P}(\wedge I)$$

As you can see we end up with an introduction rule last. However the blue derivation has been duplicated.

Classical logic

Classical logic is the logic most people have in mind where any proposition is either "true" or "false" (but not either provable or provably contradictory).

It can be obtained by extending NJ with the Law of Excluded Middle (LEM):

$$\frac{}{\Gamma \vdash P \vee \neg P}$$

It is also possible to define it differently by allowing more that one proposition on the right-hand side of judgements, in what Gentzen called NK.

While this seems like a very practical property to have, classical logic is not constructive and generally does not behave as well as intuitionistic logic when it comes to the conception of proof assistant. In particular it doesn't allow us to present the Curry-Howard correspondence in a way that is as simple as the one I will give in Chapter 3 (Simple type theory).

Mechanised proofs

Once we have a formal logic, it makes sense to *teach* its rules to a computer to use it for what we call *mechanised proofs*. We nowadays have many of those, be it under the name *proof assistants* or *automated theorem provers*.

Automated theorem provers are tools that will take statements as inputs and attempt to prove or disprove them, sometimes taking hints from the user when stuck. They are very attractive because the user usually does not have to learn about the logic involved, just trust that it *works*[2]. However it can be a hassle when they fail to prove or disprove something and the user has to figure out the right way to state things so that the tool manages to progress.

2: Ideally that it is consistent

Proof assistants require more work from the part of the user but are usually more malleable and robust. They constitute a framework in which the user can state and prove lemmata. Again there are several proof assistants: Isabelle/HOL is one of the most commonly used and based on Higher Order Logic (HOL) [10], but I am more familiar with proof assistants based on type theory such as Coq [2] and Agda [7]. The next introductory chapters will focus on this notion of proof framework.

[10]: Nipkow et al. (2002), *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*

[2]: The Coq development team (2020), *The Coq proof assistant reference manual*

[7]: Norell (2007), *Towards a practical programming language based on dependent type theory*

2.3 Limitations: Gödel's incompleteness theorems

When you have a proof framework or logic, there are two main things you want to know about it:

- Is it *consistent*? An inconsistent system is one in which we can prove an assertion and its negation.
- Is it *complete*? That is, are all propositions either provable or provably contradictory (*i.e.* their negation is provable)?

Completeness is not to be confused with the LEM which gives a proof of $P \vee \neg P$ for each P but not a proof of either P or $\neg P$ for every P.

Consistency can be reformulated equivalently as the absence of proof of false ($\perp$).

Ideally we would want our favourite system to enjoy *both* of these properties, the proof of which would be conducted in the very same system. In 1931 [11] Gödel shattered all hopes of ever accomplishing that.

[11]: Gödel (1931), 'Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I'

Theorem 2.3.1 (Gödel's first incompleteness theorem) *Every consistent formal system that supports arithmetic is incomplete.*

This means, that, except in very small systems, consistency and completeness cannot hold at the same time. Usually we go with consistency because inconsistent systems are trivially complete (every property is provable) and not of much interest.

Some people still work in inconsistent logics so let us stress the *usually*.

Theorem 2.3.2 (Gödel's second incompleteness theorem) *Every consistent formal system that supports arithmetic cannot prove its own consistency.*

A system that is able to prove itself consistent is actually inconsistent. This means that we have to rely on the scientific *consensus* I mentioned ealier, because the system in which you prove one system consistent is stronger and needs to be trusted as well.

In both these theorems we talk about the fact that the theory should support arithmetic for them to apply. By arithmetic we mean Robinson arithmetic [12] rather than Peano arithmetic, as proved in [13]. It is an axiomatisation of a set $\mathbb{N}$ with distinguished member 0 and a successor (unary) operation on $\mathbb{N}$ written S. Moreover it features two binary operations called addition and multiplication written with infix notations $+$ and $\times$. The axioms are the following.

[12]: Robinson (1950), 'An essentially undecidable axiom system'
[13]: Bezboruah et al. (1976), 'Gödel's second incompleteness theorem for Q'

$$\begin{array}{ll}
\forall x. & \mathsf{S}\ x \neq 0 \\
\forall x\ y. & \mathsf{S}\ x = \mathsf{S}\ y \to x = y \\
\forall x. & x = 0 \vee \exists y.\ x = \mathsf{S}\ y \\
\forall x. & x + 0 = x \\
\forall x\ y. & x + \mathsf{S}\ y = \mathsf{S}\ (x + y) \\
\forall x. & x \times 0 = 0 \\
\forall x\ y. & x \times \mathsf{S}\ y = x \times y + x
\end{array}$$

The first axiom states that 0 cannot be the successor of another natural number (*i.e.* some $n + 1$). The second says that the successor operation is injective: if the successors of x and y are equal, then

$x = y$. The third says that every natural number is either 0 or the successor of another natural number, this means that every natural number is a succession of S and then 0.

$$\begin{array}{cccccc} 0 & \mathsf{S}\,0 & \mathsf{S}\,(\mathsf{S}\,0) & \mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,0)) & \mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,(\mathsf{S}\,0))) & \dots \\ 0 & 1 & 2 & 3 & 4 & \dots \end{array}$$

This requirement of arithmetic is linked to the fact natural numbers can be used to encode statements—think how, for instance, this document is represented using binary code, *i.e.* numbers comprised of 0s and 1s. A process know as 'arithmetisation' allows us to use arithmetic properties on natural numbers to talk about the properties of the statement they represent. The 'trick' is then to produce a statement n that is equivalent to 'n is not provable' thus leading to a contradiction.

I have not made the statements of Gödel's incompleteness theorems very precise as it is beyond the scope of my work. It is very easy to find references that are not in German on the subject, like the well-written Wikipedia page dedicated to the subject.

These theorems do not prevent us from using formal systems of reasoning as long as one embraces the fact that some statements will inevitably be independent—neither provable nor disprovable—from the theory. We must develop other means to increase the trust we have in those systems so that we can use them, as we shall see later.

Simple type theory 3

When studying programming languages theoretically, λ-calculus imposes itself as the prototypical example. A model of computation much simpler than Turing machines, it becomes extremely useful in combination with a so-called type system. This combination culminates into the Curry-Howard isomorphism that relates programming and logic in profound ways, serving as a foundation for type theory and modern logic.

I will not do a thorough analysis and history of the subject but I will give an account of my understanding of it, limiting myself to points that I find relevant to my thesis. The interested reader can refer to [14].

[14]: Cardone et al. (2006), 'History of lambda-calculus and combinatory logic'

3.1 The λ-calculus

λ-calculus can be reasonably called the simplest programming language. It consists basically of functions, variables and applications. If you are familiar with OCaml these constructs are summarised in the example below.

```
(fun x -> x) u
```

Here, we have the identity function `fun x -> x`—*i.e.* the function which maps `x` to `x`—applied to some expression `u`. In mathematical textbooks, we would define the identity function (for natural numbers) as follows.

$$id : \left(\begin{array}{ccc} \mathbb{N} & \to & \mathbb{N} \\ x & \mapsto & x \end{array} \right)$$

The whole expression would be written $id(u)$.

The λ-calculus provides a third way of writing the same thing.

$$(\lambda x.\ x)\ u$$

The little λ corresponds to the declaration of a function, in this case *binding* the x before the dot, in the expression after it. Application of a function is marked with a space, such as $u\ v$ meaning u applied to argument v. Formally, the grammar of λ-calculus is:

$$t, u, v \coloneqq x \mid \lambda x.\ t \mid t\ u$$

x is a placeholder for any variable name, typicaly in the range of x, y and z.

and that is it.

In a term (the expressions of λ-calculus), the variable names are considered irrelevant, for instance, the λ-terms $\lambda x.\ x$ and $\lambda y.\ y$ are deemed equivalent, because they both define the identity function.

The operation replacing x for y in the term is called α-renaming; we thus talk about α-equality:

$$\lambda x.\ x =_\alpha \lambda y.\ y$$

Of course, that alone is not sufficient to describe a programming language, it is missing a key component: evaluation of programs. Indeed, without it, variables are meaningless.

We want to say that the application of a function should always yield some result. For instance, the identity function $\lambda x.\ x$ when applied to u should naturally reduce to u itself.

The purpose of a variable is to be *instantiated*. Before we talk about this, we need to talk about *bound* and *free* variables. If you take the expression $(\lambda x.\ x)\ y$ you can see two variables x and y; the two do not behave the same: x is below a λ-abstraction that *introduces* (or *binds*) it whereas y is *free*, no λ-abstraction constrains it. Of course, this status changes if the term is put inside another:

$$\lambda y.\ (\lambda x.\ x)\ y$$

This times both y and x are *bound*.

A term with free variables is called *open*, while a term with only bound variables is called *closed*.

The only variables that can be instantiated are the *free* ones. For example, it would not make sense to replace the x in the identity $\lambda x.\ x$ function, it would somehow break it. *Substitution* is the operation that replaces free variables with other expressions. We will write

Sometimes this fact is summarised in the term *capture-avoiding* meaning that bound variables are not touched. I will refrain from using a qualifier that basically means that it is not a nonsensical definition.

$$t[x \leftarrow u]$$

to mean the term t were all *free* occurrences of the variable x are replaced by the term u.

For example,

$$((\lambda x.\ x)\ y)[y \leftarrow u] = (\lambda x.\ x)\ u$$

Now that we have substitution, we are armed to deal with reduction. Reduction is defined from the so-called β-reduction

$$(\lambda x.\ t)\ u \rightarrow_\beta t[x \leftarrow u]$$

It is essentially saying that when applied to an argument, a function reduces to its body were the argument replaces the variable it was binding. Reduction $\rightarrow$ is then obtained by taking the contextual closure of $\rightarrow_\beta$, *i.e.* allowing reduction in each subterm:

$$\frac{u \rightarrow u'}{u\ v \rightarrow u'\ v} \qquad \frac{v \rightarrow v'}{u\ v \rightarrow u\ v'} \qquad \frac{t \rightarrow t'}{\lambda x.\ t \rightarrow \lambda x.\ t'}$$

$$\frac{}{(\lambda x.\ t)\ u \rightarrow t[x \leftarrow u]}$$

There is a lot more to say on the *untyped* λ-calculus, but this out of scope of this document.

3.2 Types for programs

Not all programs make sense. Consider the term

$$\delta \coloneqq \lambda x.\ x\ x$$

giving x as argument to x should already feel wrong somehow, but if you give δ as argument to itself you get

$$\Omega \coloneqq \delta\ \delta$$

Now let us have a look at its computational behaviour:

$$\begin{array}{rcl} \Omega & \coloneqq & \delta\ \delta \\ & = & (\lambda x.\ x\ x)\ \delta \\ & \rightarrow_\beta & \delta\ \delta \\ & = & \Omega \end{array}$$

The problem should become apparent: Ω reduces to itself! This means that the evaluation of Ω will not terminate. Generally, that is not something you want to have. In case you have more structure in your language, such as in OCaml where you have integers, you also want to restrict some operations like addition to expressions that are of the right *kind*—in our case, integers. In OCaml, expressions like `true + 3` or `1 + fun x -> 2` should be—and are!—rejected.

The way we prevent such problematic terms—*i.e.* those that do not have semantics—is by using *types*. The λ-calculus is pretty simple, the only things we can do is define functions and apply them, as such we introduce a type of functions $A \rightarrow B$. The expression $A \rightarrow B$ denotes the type of functions that take an argument of type A and produce a value of type B. If we go back to OCaml for a bit, a function like `fun x -> x + x` will have type `int -> int` while another making use of different data structures like

```
fun b -> if b then "Yes" else "No"
```

will be of type `bool -> string`. Going back to λ-calculus, the important rule is that when you have a function f of type $A \rightarrow B$—which we will write $f : A \rightarrow B$—and an argument $u : A$, the result of the application will have type B:

$$\frac{f : A \rightarrow B \qquad u : A}{f\ u : B}$$

The usefulness of types is best summarised in the famous quote:

> "Well-typed programs cannot go wrong."
>
> — Robin Milner [15]

[15]: Milner (1978), 'A theory of type polymorphism in programming'

A program can 'go wrong' if it needs to evaluate a nonsensical expression like 1 applied to 0 which might have unexpected behaviours on a computer.

As I mentioned earlier, λ-terms can sometimes be open terms—*i.e.* contain free variables—so in order to type them, we need to know the type of those variables. This information is called an environment or *context*. We can thus give the syntax of types and contexts.

$$\begin{array}{rcl} A, B & ::= & o \mid A \to B \\ \Gamma, \Delta & ::= & \bullet \mid \Gamma, x : A \end{array}$$

o stands for base types. A context is a list of type assignments for the variables, implicitly the variables are distinct.

The typing rules of the Simply Typed λ-calculus (STL) are the following.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\ t : A \to B} \qquad \frac{\Gamma \vdash u : A \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\ v : B} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

There are many interesting properties that STL enjoys, one of the most important is that it is terminating, *i.e.* reduction cannot go indefinitely. This excludes *bad* terms like Ω.

See Chapter 7 (Desirable properties of type theories) for more on this.

3.3 The Curry-Howard isomorphism

Now, if we look back on the rules I gave for implication in Chapter 2 (Proof theory), there is a parallel with the typing rules of STL.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\ t : A \to B} \qquad \frac{\Gamma \vdash u : A \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\ v : B} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}$$

If we forget about the terms for a bit, they are exactly the *same*. The Curry-Howard isomorphism sets up a parallel between types and propositions and between programs and proofs. This is the origin of a very fruitful relationship between the two worlds.

The correspondence does not stop there, computation also plays a part in it. Consider the term $(\lambda x.x)\ y$ it can be typed as follows (assuming y has type A).

$$\frac{\dfrac{\dfrac{}{y : A, x : A \vdash x : A}}{y : A \vdash \lambda x.x : A \to A} \qquad \dfrac{}{y : A \vdash y : A}}{y : A \vdash (\lambda x.x)\ y : A}$$

However, we know it can compute with β-reduction:

$$(\lambda x.x)\ y \to_\beta y$$

This yields a simpler term with a simpler typing derivation.

$$\frac{}{y : A \vdash y : A}$$

This means that β-reduction allows for simplification of proofs. This corresponds to the *cut-elimination* I introduced in Chapter 2 (Proof theory) which removes all superfluous applications of *cut* (or *modus ponens* in our case) in the proof. Here it removes direct applications of a function to an argument. Thanks to that, a proof of $A \longrightarrow B$ is in fact an algorithm turning a proof of A into a proof of B.

All this is very useful and was exciting for me. However, simple types are quite limited [16] and do not allow for a proper handling of things like quantifiers (∀ and ∃). Hence the need for *polymorphism* and *dependent* types. The latter will be the subject of the next chapter.

[16]: Zakrzewski (2007), 'Definable functions in the simply typed lambda-calculus'

Dependent types 4

The idea behind dependent types is that types now can *depend* on terms, *i.e.* terms can appear in types. This is very interesting because we can talk about things like $P\ n$ for a property P on a natural number n, equality $x = y$ for two terms x and y, and with it, we can support quantifiers.

Thus, $x = y$ becomes a type depending on x and y, and whose inhabitants are witnesses of the fact that x and y are equal.

It is not only useful on the logical side, but also on the programming language side: with it programs can be given much more precise types. For instance, you might want to specify that the division operator does not accept 0 for the denominator, or that the operation returning the tail of list only applies to non-empty lists. You can also have the type of lists of length n. Finally, we can take advantage of both and write *proofs* about the programs we wrote, using the very same language.

4.1 A minimal dependent type theory

Let me describe a very basic type theory featuring dependent types.

Π-types. The simplest way to get dependent types is to extend the STL—which only features arrow, or function, types—with dependent function types or Π-types.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A).\ B}$$

As you can see, we keep the λ-terms of earlier but now they represent dependent functions. Now, not only t can mention x, but B also. For instance we can write the polymorphic identity function as follows

Polymorphism is the ability to use a program at different types. The polymorphic identity function means that it is the identity function for any type that we desire: we can type it with $\mathbb{N} \to \mathbb{N}$, $\mathsf{bool} \to \mathsf{bool}$ or anything really.

$$\lambda(A : \mathsf{Type}).\ \lambda(a : A).\ a$$

I will explain later what the Type here means, but it should be intuitive: I am taking a *type* A as argument and then an element a of that type.

and it has type

$$\Pi(A : \mathsf{Type}).\ \Pi(a : A).\ A$$

a fact that we can write as

$$\lambda\ (A : \mathsf{Type})\ (a : A).\ a : \Pi\ (A : \mathsf{Type})\ (a : A).\ A$$

or even more concisely as

$$\lambda\ (A : \mathsf{Type})\ (a : A).\ a : \Pi\ (A : \mathsf{Type}).\ A \to A$$

$A \to B$ is no longer a definition, but a notation for the dependent function type which is not actually dependent $\Pi(\_ : A).\ B$.

since a is not mentioned in the type. Now we should be able to see the dependency on A. One can argue that A is a *type* and not a term, but in this setting, types are just a special kind of terms that happen to be of type Type.

We will also be able to write functions like

$$\lambda(n : \mathbb{N}).\ [0, 1, \dots, n] : \Pi(n : \mathbb{N}).\ \mathsf{vec}\ (n + 1)$$

where $\mathsf{vec}\ n$ is the type of lists of length n containing natural numbers (which we call vectors).

If we have some B : Type we might want to apply our polymorphic identity function to it to get the identity function on B, *i.e.*

$$\lambda(a : B).\ a : B \to B$$

For this we have to rely on substitutions again, not only in the terms after β-reduction, but also in types. This can be seen in the application rule:

$$\frac{\Gamma \vdash u : \Pi(x : A).\ B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\ v : B[x \leftarrow v]}$$

Once again it is pretty similar to the application rule of simple type theory except we have to account for the dependency. In our example—writing id for the polymorphic identity function—we have

$$\frac{\Gamma \vdash \mathsf{id} : \Pi(A : \mathsf{Type}).\ A \to A \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash \mathsf{id}\ B : B \to B}$$

What happens is the type of the application is

$$(A \to A)[A \leftarrow B] = B \to B$$

We can apply our vector function to 3 for instance to get

$$(\lambda(n : \mathbb{N}).\ [0, 1, \dots, n])\ 3 : \mathsf{vec}\ 4$$

The result of this application (*i.e.* after β-reduction) is

$$[0, 1, 2, 3]$$

which is indeed a list of length 4.

Universes. In the example above there is the peculiar type Type. This can be thought of as the type of types. If you know about Russell's paradox stating that there can be no set of all sets, you might be skeptical and indeed having Type of type Type is inconsistent. We will address this in more details in Subsection 'Universes in Coq' in Chapter 6 (Flavours of type theory). In this case, Type will be a special type—which we call *universe* as it is inhabited solely by types—that does not have a type itself. Having it is mainly to allow for quantifying over types, but we could also of course define some base types like the natural numbers nat and put it in

$$\mathsf{nat} : \mathsf{Type}$$

We will see more example of types in Chapter 5 (Usual definitions in type theory). For now we only have Π-types in them:

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

Here we evidence the fact B is indeed dependent over $x : A$ and that the type $\Pi(x : A).\ B$ is correctly constructed. This kind of universe is called a Russell universe, and there are also Tarski universes, the difference will be explained in Subsection 'Universes and types' of Chapter 9 (Syntax and formalisation of type theory).

Computation. As was the case in Chapter 3 (Simple type theory), we have β-reduction in dependent type theory:

$$(\lambda(x : A).\ t)\ u \to_\beta t[x \leftarrow u]$$

Now that terms can appear in types, reduction can also happen in them. For instance, we would probably want $\mathsf{vec}\ (3+1)$ and $\mathsf{vec}\ 4$ to be related. We will in fact have

$$\mathsf{vec}\ (3+1) \to \mathsf{vec}\ 4$$

If you consider the following concat function with concatenates two vectors

$$\mathsf{concat} : \Pi\ (n : \mathbb{N})\ (m : \mathbb{N})\ (v_1 : \mathsf{vec}\ n)\ (v_2 : \mathsf{vec}\ m).\ \mathsf{vec}\ (n+m)$$

we would have

$$\mathsf{concat}\ 3\ 1\ [0,1,2]\ [3] : \mathsf{vec}\ (3+1)$$

meaning that $\mathsf{vec}\ (3+1)$ can appear naturally. In this case, we expect the concatenation function to compute like this

$$\mathsf{concat}\ 3\ 1\ [0,1,2]\ [3] \to [0,1,2,3]$$

but as I presented before, $[0,1,2,3]$ has type $\mathsf{vec}\ 4$. As such we want to argue that $\mathsf{vec}\ (3+1)$ and $\mathsf{vec}\ 4$ are the same. For this we introduce the notion of *conversion*. Conversion $A \equiv B$ essentially means that A and B are equal up to reduction ($\to$); we say that A and B are *convertible*. In our case we would have

$$\mathsf{vec}\ (3+1) \equiv \mathsf{vec}\ 4$$

We can change a type for a convertible one in typing, essentially saying that if t has type A and A is convertible to B then t also has type B:

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

This allows us to say

$$\mathsf{concat}\ 3\ 1\ [0,1,2]\ [3] : \mathsf{vec}\ 4$$

directly.

It is often the case that reduction ($\to$) is confluent and terminating and that, as such, conversion is decidable.

See Chapter 7 (Desirable properties of type theories) for a definition of these properties.

I will now put the syntax and the rules together for clarity, at the risk of repeating myself.

$$\begin{array}{rcl} A, B, t, u & ::= & x \mid \lambda(x : A).t \mid t\ u \mid \Pi(x : A).B \mid \mathsf{Type} \\ \Gamma, \Delta & ::= & \bullet \mid \Gamma, x : A \end{array}$$

• is the empty context.

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

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash A : \mathsf{Type}}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A).\ B} \qquad \frac{\Gamma \vdash u : \Pi(x : A).\ B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\ v : B[x \leftarrow v]}$$

$$\frac{\Gamma \vdash t : A \qquad A \equiv B \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash t : B}$$

Here I changed a bit the typing rule for λ-abstraction to also ask for the domain to be well-formed. This is necessary to ensure we put legitimate types in the context.

Similarly, I ask that the type B is well-typed in the conversion rule.

Where conversion is defined as the congruent closure of β-reduction.

$$\frac{}{t \equiv t} \qquad \frac{u \equiv v}{v \equiv u} \qquad \frac{u \equiv v \qquad v \equiv w}{u \equiv w} \qquad \frac{}{(\lambda(x : A).\ t)\ u \equiv t[x \leftarrow u]}$$

$$\frac{A \equiv A' \qquad B \equiv B'}{\Pi(x : A).B \equiv \Pi(x : A').B'} \qquad \frac{A \equiv A' \qquad t \equiv t'}{\lambda(x : A).t \equiv \lambda(x : A').t'}$$

$$\frac{u \equiv u' \qquad v \equiv v'}{u\ v \equiv u'\ v'}$$

We usually also add a definition of well-formed contexts to ensure they are comprised of types that make sense and that the dependencies are in order:

$$\frac{}{\vdash \bullet} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A : \mathsf{Type}}{\vdash \Gamma, x : A}$$

As you can see, each type might depend on the previous variables.

4.2 Dependent types in **Coq**

In this thesis I will often refer to the type theory of Coq as well as give some definitions in it. I will give here a brief introduction to it with examples, but this does not aim at being a tutorial or a manual for Coq.

In Coq, Π-types are written

```
forall (x : A), B
```

and λ-abstractions are written

```
fun (x : A) => t
```

In both cases the domain (A) can be left out if Coq manages to infer it from the context:

`nat` is the built-in type of natural numbers.

```
fun x => x + 0 : nat -> nat
```

The polymorphic identity function of earlier is

```
fun A x => x : forall A, A -> A
```

We can write it as a definition in the system as follows:

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

The squigly brackets indicate that the argument A is implicit so that we can later write `id 0` so that Coq infers that A is `nat`.

Writing down functions is not the only way to define terms in Coq however. One of its strengths is the tactic mechanism that it is equipped with which allows the user to write proofs in an interactive way. Instead of writing the polymorphic identity function we could prove the mathematical statement $\forall A, A \to A$.

```
Fact id_as_proof :
  forall A, A -> A.
Proof.
  intro A.
  intro x.
  assumption.
Qed.
```

The way this works is that we first state what we wish to prove

```
forall A, A -> A
```

and how we want to refer to that fact afterwards (`id_as_proof`). After the keyword **`Proof`**, Coq is in interactive mode, telling the user what they have to prove to conclude the proof. At the beginning this is still the full statement

```
-----------------
forall A, A -> A
```

The horizontal bar separates the hypotheses (above) from the goal to prove (below).

The user can then write tactics to progress with the proof. I will write side by side the tactics used to progress and the goal after its execution.

To prove a quantified statement, as usual, you want to assume some element in particular and prove the statement for that one. This is what the tactic **`intro`** does.

```
intro A.          | A : Type
                  | -----------------
                  | A -> A
```

We have now *introduced* A in our context and need to prove `A -> A`.

```
intro A.          | A : Type
intro x.          | x : A
                  | -----------------
                  | A
```

We assumed some `x : A`, so it joined the assumptions, leaving us to prove A. Proving A, when we have A as an hypothesis should be pretty straightforward. The tatic **`assumption`** tells Coq that the goal can be concluded using one of the assumptions in the context (here x since `x : A` *i.e.* x is a proof witness of A).

```
intro A.
intro x.
assumption.
```

```
No more goals.
```

This generates a term, and when we write **Qed** it is checked by Coq to make sure it is correct.

As one should expect this produces the same term as before

```
id_as_proof = fun A x => x
            : forall A, A -> A
```

That is to say, the polymorphic identity function is a proof witness of

```
forall A, A -> A
```

There is of course much more to it and Chapter 5 (Usual definitions in type theory) will offer some usual definitions in type theory and in the context of Coq.

5 Usual definitions in type theory

Dependent type theory as presented in Chapter 4 (Dependent types) is rather barren. It really shines when extended with some interesting principles and datatypes. I will give an overview of these features—with the Coq proof assistant in mind—and focus mainly on those that are relevant to this thesis.

5.1 Inductive types and pattern-matching

Inductive types are probably the most emblematic feature of dependent type theory. They can be seen as an extension of the variant datatypes present in OCaml like the type of lists.

```
type 'a list =
| nil
| cons of 'a * 'a list
```

A list, say of integers, is either empty (`nil`) or some head `h : int` and some tail `t : int list`, written `cons h t`.

Here, the `'a` represents *any* type. For instance, `int list` is the type of lists of integers. This is called polymorphism.

They come in different flavours which I will try to explain.

Variants

The simplest case of inductive types is that of variants. They consist in a list of different options.

Booleans. bool is the type inhabited by true and false.

$$\frac{}{\Gamma \vdash \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

In Coq you would write it as follows:

```
Inductive bool : Type :=
| true
| false.
```

In other words, `true` and `false` are the only *constructors* of the type `bool`.

Of course, having those is not nearly enough without the usual if construct. For instance `if b then 0 else 1` will return `0` if b is `true` and `1` if it is `false`. This is already something that makes sense in the simple case[1], but with dependent types the case analysis is also

1: See Chapter 3 (Simple type theory)

dependent on the scrutinee (*i. e.* the boolean b in that case). The corresponding typing rule is an *elimination* rule as we have seen in Chapter 2 (Proof theory).

$$\frac{\Gamma, x : \mathsf{bool} \vdash P \qquad \begin{array}{c}\Gamma \vdash b : \mathsf{bool} \\ \Gamma \vdash u : P[x \leftarrow \mathsf{true}]\end{array} \qquad \Gamma \vdash v : P[x \leftarrow \mathsf{false}]}{\Gamma \vdash \mathsf{if}\ b\ \mathsf{return}\ x.P\ \mathsf{then}\ u\ \mathsf{else}\ v}$$

Before we break down the typing rule, let me show you the computational behaviour of if.

$$\begin{array}{lcl} \mathsf{if\ true\ return}\ x.P\ \mathsf{then}\ u\ \mathsf{else}\ v & \rightarrow & u \\ \mathsf{if\ false\ return}\ x.P\ \mathsf{then}\ u\ \mathsf{else}\ v & \rightarrow & v \end{array}$$

It is still the same as the well-known if, except that we are more liberal in the types given to the two branches: they do not have to match as they can now depend on the boolean. The $x.P$ notation means that P lives in a context extended by x (of type bool).

One can for instance write the following definition where the return type does *not* depend on the boolean:

$$\mathsf{P} := \lambda(b : \mathsf{bool}).\ \mathsf{if}\ b\ \mathsf{return}\ x.\mathsf{Type\ then\ bool\ else\ nat}$$

where nat is the type of natural numbers I am going to present later. Here, we have P true = bool and P false = nat. Using this as a return type, we can write a dependent version of the if:

$$\mathsf{if}\ b\ \mathsf{return}\ x.\ \mathsf{P}\ x\ \mathsf{then\ true\ else}\ 0$$

The idea is that in the branch where b is true, we have to provide a boolean, here true, and in the branch where b is false, we have to provide a natural number, I picked 0.

The if is actually just a notation for a more generic construction called *pattern-matching*. if b return $x.P$ then u else v is in fact the term

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{match}\ b\ \mathsf{return}\ x.P\ \mathsf{with}} \\ \mid \mathsf{true} & \Longrightarrow & u \\ \mid \mathsf{false} & \Longrightarrow & v \\ \multicolumn{3}{l}{\mathsf{end}} \end{array}$$

It describes the case analysis by saying which constructor is sent to which term. If the scrutinee—here b—*matches* one of the branches on left-hand side of $\Longrightarrow$, the whole expression will reduce to the corresponding right-hand side.

Unit. The unit type is similar to bool but has only one constructor written ().

$$\frac{}{\Gamma \vdash \mathsf{unit}} \qquad\qquad \frac{}{\Gamma \vdash () : \mathsf{unit}}$$

In Coq it is defined as:

```
Inductive unit : Type :=
| tt.
```

And the notation mechanism can help use write `tt` as `()`. Once again, pattern-matching allows us to inspect a proof of unit:

$$\frac{\Gamma \vdash u : \mathsf{unit} \qquad \Gamma, x : \mathsf{unit} \vdash P \qquad \Gamma \vdash v : P[x \leftarrow ()]}{\Gamma \vdash \left(\begin{array}{l}\mathsf{match}\ u\ \mathsf{return}\ x.P\ \mathsf{with} \\ \mid () \implies v \\ \mathsf{end}\end{array}\right) : P[x \leftarrow u]}$$

As all rules pertaining to pattern-matching, it is an elimination rule. In fact, pattern-matching is the only way—besides application—to express elimination in Coq.

This rule might seem a bit useless, but essentially it means that to prove anything involving a dependency on a term u of type unit, like $P[x \leftarrow u]$, it suffices to prove it assuming u is $()$: $P[x \leftarrow ()]$. When the term u was already $()$, the match can go away:

$$\begin{array}{l}\mathsf{match}\ ()\ \mathsf{return}\ x.P\ \mathsf{with} \\ \mid () \implies v \\ \mathsf{end}\end{array} \quad \rightarrowtriangle v$$

Sometimes, this type is called $\top$ as in the logical triviality.

Empty type. The empty (or false) type, $\bot$, is the dual of the unit type. This time it has no constructors *at all*.

$$\frac{}{\Gamma \vdash \bot}$$

In Coq, it is written in a rather queer manner.

```
Inductive False :=.
```

It represents the data that should never exist, so any term of type $\bot$ is a *contradiction* with the hypotheses at hand. Even though it does not have constructors, pattern-matching still makes sense on such terms.

$$\frac{\Gamma \vdash t : \bot \qquad \Gamma, x : \bot \vdash P}{\Gamma \vdash \left(\begin{array}{l}\mathsf{match}\ t\ \mathsf{return}\ x.P\ \mathsf{with} \\ \\ \mathsf{end}\end{array}\right) : P[x \leftarrow t]}$$

The pattern-matching does not have any branches, hence the empty space.

This is the essence of the *principle of explosion*: *ex falso quodlibet*, from falsehood, anything follows. Here we are able to conjure some inhabitant of $P[x \leftarrow t]$ from thin air. The P is typically not dependent on the proof of $\bot$, meaning that from an inhabitant of $\bot$ we can get an inhabitant of *any* type.

In dependent type theory we will define negation $\neg P$ as $P \to \bot$. As such, we might still end up with inhabitants of $\bot$ when dealing with hypotheses of the shape $\neg P$.

Recursive types

For now, I only presented inductive types that consist in enumerations of cases. Inductive types can be more complex and constructors can take subterms as arguments. The best and simplest example of those is that of natural numbers.

Natural numbers. The way we represent *unary* natural numbers in type theory is by saying a natural number is either 0 or the successor of another natural number n, *i.e.* $n+1$. We usually write the successor operation $\mathbf{S}$. So natural numbers are 0, $\mathbf{S}\ 0$, $\mathbf{S}\ (\mathbf{S}\ 0)$, etc. respectively representing 0, 1, 2, …

$$\frac{}{\Gamma \vdash \mathsf{nat}} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{nat}} \qquad \frac{\Gamma \vdash n : \mathsf{nat}}{\Gamma \vdash \mathbf{S}\ n : \mathsf{nat}}$$

$$\frac{\Gamma, x : \mathsf{nat} \vdash P \qquad \Gamma \vdash u_0 : P[x \leftarrow 0] \qquad \begin{array}{c}\Gamma \vdash n : \mathsf{nat}\\ \Gamma, m : \mathsf{nat} \vdash u_{\mathbf{S}} : P[x \leftarrow \mathbf{S}\ m]\end{array}}{\Gamma \vdash \left(\begin{array}{lll}\mathsf{match}\ n\ \mathsf{return}\ x.P\ \mathsf{with} & & \\ \mid 0 & \Longrightarrow & u_0 \\ \mid \mathbf{S}\ m & \Longrightarrow & u_{\mathbf{S}} \\ \mathsf{end} & & \end{array}\right) : P[x \leftarrow n]}$$

Notice how $u_{\mathbf{S}}$ is allowed to mention m. The variable is bound by the pattern $\mathbf{S}\ m$ on the left-hand side.

The first three rules are *introduction* rules, and as before, the typing rule for pattern-matching is an *elimination* rule.

The rules come together with the computation rules

$$\begin{array}{lllcl}\mathsf{match}\ 0\ \mathsf{return}\ x.P\ \mathsf{with} & & & & \\ \mid 0 & \Longrightarrow & u_0 & \rightarrowtriangle & u_0 \\ \mid \mathbf{S}\ m & \Longrightarrow & u_{\mathbf{S}} & & \\ \mathsf{end} & & & & \\ \mathsf{match}\ \mathbf{S}\ n\ \mathsf{return}\ x.P\ \mathsf{with} & & & & \\ \mid 0 & \Longrightarrow & u_0 & \rightarrowtriangle & u_{\mathbf{S}}[m \leftarrow n] \\ \mid \mathbf{S}\ m & \Longrightarrow & u_{\mathbf{S}} & & \\ \mathsf{end} & & & & \end{array}$$

As with the previous examples, this allows us to do case analysis on natural numbers but this is no longer sufficient to effectively reason on them. The bare minimum we would require is to do induction on natural numbers.

There are two main ways of achieving this.

- *Eliminators.* This method consists in assuming the induction principle of nat directly.

  $$\mathsf{natrec} : \Pi\ P.\ P\ 0 \to (\Pi n.\ P\ n \to P\ (\mathbf{S}\ n)) \to \Pi n. P\ n$$

  The type of natrec means that, for all proposition P on natural numbers, if $P\ 0$ holds and $P\ n$ implies $P\ (n+1)$, then P holds for all natural numbers, which is the well-known induction principle on ℕ. The eliminator comes with computation rules as well:

$$\begin{array}{lcl} \mathsf{natrec}\ P\ p_0\ p_{\mathbf{S}}\ 0 & \rightharpoonup & p_0 \\ \mathsf{natrec}\ P\ p_0\ p_{\mathbf{S}}\ (\mathbf{S}\ n) & \rightharpoonup & p_{\mathbf{S}}\ n\ (\mathsf{natrec}\ P\ p_0\ p_{\mathbf{S}}\ n) \end{array}$$

They tell us how to produce proofs from an induction. To get a proof of $P\ 0$ we simply use $p_0 : P\ 0$, and to get a proof of $P\ (n+1)$ we apply $p_{\mathbf{S}}$ which produces a proof of $P\ (n+1)$ from a proof of $P\ n$, the latter is obtained by applying the eliminator again. There are ways to generate eliminators automatically [17] and Coq does it to some extent. The troublesome part is getting the right computation rules and dealing with more complicated kinds of inductive types such as nested inductive types that I will not describe.

[17]: Kovács et al. (2020), 'Signatures and Induction Principles for Higher Inductive-Inductive Types'

- *Fixed-points*. The method used in Coq comes from a combination of pattern-matching and a fixed-point operator.

$$\frac{\Gamma \vdash \Pi\Delta.T \qquad \Gamma, f : \Pi\Delta.T \vdash t : \Pi\Delta.T \qquad f \vdash_n t \text{ termination checks}}{\Gamma \vdash \mathsf{fix}_n(f : \Pi\Delta.T).\ t : \Pi\Delta.T}$$

The notation $\Pi\Delta.T$ is to quantify over a whole context. If Δ is $x : A, y : B$, then $\Pi\Delta.T$ is $\Pi(x : A)\ (y : B).\ T$. When Δ is empty, this is just T.

As you can see there is an extra condition I called 'termination checking'. If we were not to restrict the definition of fixed-points to terminating functions we would end up with an inconsistent theory. Indeed, one could inhabit the empty type with the following trivial fixed-point which just calls itself directly:

$$\mathsf{fix}(f : \bot).\ f$$

The typing derivation showing it has type $\bot$ is the following

$$\frac{\dfrac{}{\vdash \bot} \qquad \dfrac{}{f : \bot \vdash f : \bot}}{\vdash \mathsf{fix}(f : \bot).\ f : \bot}$$

Here, I took $\Delta := \bullet$ and $T := \bot$.

The termination checking condition roughly verifies that the n-th argument in Δ (*i.e.* x_n if $\Delta = x_1 : A_1, \dots x_m : A_m$) is only fed to f (*i.e.* recursive calls) in a decreasing manner. I write n as the subscript to the judgment to indicate that is relevant to the termination checking. If the n-th argument is a natural number, it roughly means that every recursive call must be done on a smaller natural number. This so-called guard condition is a complicated matter and out of scope of this thesis. It is probably better studied in [18, 19]. In general this can be thought as only subterms of the argument are passed on to f (thus, the fixed-point above is illegal because there is no argument that becomes strictly smaller, it is however fine to call $f\ n$ to compute $f\ (\mathbf{S}\ n)$ as n is a (strict) subterm of $\mathbf{S}\ n$). The computational behaviour of the fixed-point operator is via the unfolding of its definition.

[18]: Giménez (1998), 'Structural recursive definitions in type theory'
[19]: Giménez (1994), 'Codifying guarded definitions with recursive schemes'

$$\mathsf{fix}_n(f : \Pi\Delta.T).\ t \rightharpoonup t[f \leftarrow \mathsf{fix}_n(f : \Pi\Delta.T).\ t]$$

Of course, doing this would defeat the purpose of the termination checker: ensuring that the obtained definition terminates. As you can see you can unfold the fixed-point indefinitely. The way we prevent that is by using a syntactical guard on the reduction rule. It is instead the following.

$$\begin{array}{l}(\mathsf{fix}_n(f : \Pi\Delta.T).\ t)\ u_1\ \ldots\ u_{n-1}\ (\mathbf{C}\ v_1\ \ldots\ v_m) \quad \rightharpoondown \\ t[f \leftarrow \mathsf{fix}_n(f : \Pi\Delta.T).\ t]\ u_1\ \ldots\ u_{n-1}\ (\mathbf{C}\ v_1\ \ldots\ v_m)\end{array}$$

$\mathbf{C}$ stands for a constructor like 0 or $\mathbf{S}$.

A fixed-point can thus only be unfolded when its recursive argument is a constructor. It will thus have to consume it and apply the recursive calls (*i.e.* the fixed-point itself now that is has been unfolded) on subterms of the constructor (typically one of the v_i). We can then write down the induction principle natrec with fix and match.

$$\begin{array}{l}\mathsf{fix}_4\ (f : \Pi\ P\ p_0\ p_\mathbf{S}\ n.\ P\ n). \\ \quad \lambda\ P\ p_0\ p_\mathbf{S}\ n. \\ \quad\ \ \mathsf{match}\ n\ \mathsf{return}\ x.\ P\ x\ \mathsf{with} \\ \quad\ \ \mid 0 \quad \Longrightarrow \quad p_0 \\ \quad\ \ \mid \mathbf{S}\ m \quad \Longrightarrow \quad p_\mathbf{S}\ m\ (f\ P\ p_0\ p_\mathbf{S}\ m) \\ \quad\ \ \mathsf{end}\end{array}$$

Notice how m is a subterm of $\mathbf{S}\ m$ in the recursive call to f. It is indeed its fourth argument.

It will have the same reduction rules as the eliminator shown above.

I will not go into that much detail for other inductive types as they will all follow more or less the same schema.

Parametrised inductive types

Parametrised inductive types, as the name suggests, can take parameters. In OCaml, parameters are necessarily types, in Coq they are often types but they can be natural numbers or a term of any other type. For instance, in the type list A, A is a parameter that specifies the nature of the objects in the list.

Lists. I already presented the list type of OCaml, and its Coq counterpart is not much different.

$$\frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{list}\ A} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{nil} : \mathsf{list}\ A} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash h : A \qquad \Gamma \vdash t : \mathsf{list}\ A}{\Gamma \vdash h :: t : \mathsf{list}\ A}$$

$$\frac{\begin{array}{c}\Gamma \vdash l : \mathsf{list}\ A \qquad \Gamma, x : \mathsf{list}\ A \vdash P \\ \Gamma \vdash u_{\mathsf{nil}} : P[x \leftarrow \mathsf{nil}] \qquad \Gamma, h : A, t : \mathsf{list}\ A \vdash u_{::} : P[x \leftarrow h :: t]\end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{match}\ l\ \mathsf{return}\ x.P\ \mathsf{with} \\ \mid \mathsf{nil} \quad \Longrightarrow \quad u_{\mathsf{nil}} \\ \mid h :: t \quad \Longrightarrow \quad u_{::} \\ \mathsf{end}\end{array} : P[x \leftarrow l]}$$

The computation rules are the ones you should expect by now. Using pattern-matching and fixed-points—as we did for natural numbers—

we get the induction principle on lists of type

$$\Pi\, A\ (P : \mathsf{list}\ A \to \mathsf{Type}).\ P\ \mathsf{nil} \to (\Pi\, h\ t.\ P\ t \to P\ (h :: t)) \to \Pi\, l.\ P\ l$$

Options. option A represents a potential datum of type A but it could also not be present. It is optional.

$$\frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{option}\ A} \qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{Some}\ a : \mathsf{option}\ A} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{None} : \mathsf{option}\ A}$$

$$\frac{\begin{array}{cc} \Gamma \vdash o : \mathsf{option}\ A & \Gamma, x : \mathsf{option}\ A \vdash P \\ \Gamma, a : A \vdash u_{\mathsf{Some}} : P[x \leftarrow \mathsf{Some}\ a] & \Gamma \vdash u_{\mathsf{None}} : P[x \leftarrow \mathsf{None}] \end{array}}{\Gamma \vdash \begin{array}{l} \mathsf{match}\ o\ \mathsf{return}\ x.P\ \mathsf{with} \\ \mid \mathsf{Some}\ a \implies u_{\mathsf{Some}} \\ \mid \mathsf{None} \implies u_{\mathsf{None}} \\ \mathsf{end} \end{array} : P[x \leftarrow o]}$$

It gives us an easy way of representing functions that are not defined on their whole domain (they return None when they are not). For instance we could write a division function divide such that

$$\mathsf{divide}\ x\ 0 = \mathsf{None}$$

meaning that division by 0 is not defined, while in other cases it would return Some, for example

$$\mathsf{divide}\ 10\ 5 = \mathsf{Some}\ 2$$

Sum types. Simple sums $A + B$ consist in a disjunction of cases. A proof of $A + B$ is either a proof of A or a proof of B.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A + B} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash B}{\Gamma \vdash \mathsf{inl}\ a : A + B} \qquad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash A}{\Gamma \vdash \mathsf{inr}\ b : A + B}$$

$$\frac{\begin{array}{cc} \Gamma \vdash p : A + B & \Gamma, x : A + B \vdash P \\ \Gamma, a : A \vdash u : P[x \leftarrow \mathsf{inl}\ a] & \Gamma, b : B \vdash v : P[x \leftarrow \mathsf{inr}\ b] \end{array}}{\Gamma \vdash \begin{array}{l} \mathsf{match}\ p\ \mathsf{return}\ x.P\ \mathsf{with} \\ \mid \mathsf{inl}\ a \implies u \\ \mid \mathsf{inr}\ b \implies v \\ \mathsf{end} \end{array} : P[x \leftarrow p]}$$

The rules should be reminiscent of the introduction and elimination rules of disjunction $A \lor B$ in NJ, presented in Chapter 2 (Proof theory).

Simple products. Simple products $A \times B$ are types of pairs of elements, one in A and one in B. None too surprising.

You can once again refer to Chapter 2 (Proof theory) and find a parallel with the rules regarding conjunction $A \wedge B$ in NJ.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B}$$

$$\frac{\Gamma \vdash p : A \times B \qquad \Gamma, x : A \times B \vdash P \qquad \Gamma, a : A, b : B \vdash t : P[x \leftarrow (a,b)]}{\Gamma \vdash \begin{array}{l} \textsf{match}\ p\ \textsf{return}\ x.P\ \textsf{with} \\ \mid (a,b) \implies t \\ \textsf{end} \end{array} : P[x \leftarrow p]}$$

This presentation of pairs is called *positive* as it uses a constructor. In the section on records, I will show the *negative* version.

The 'positive'/'negative' terminology is not standard but I believe it to be fairly usual in my field of expertise.

Σ-types. Dependent sums or Σ-types are a generalisation of simple products to dependent types. They are the way to represent existential quantifiers (except in a—usually—constructive way): $\Sigma(x : A).P\ x$ is proven by giving a term $t : A$ and a proof of $P\ t$.

In the simple case we have sums $A + B$, products $A \times B$ and exponentials B^A or $A \to B$. In the dependent case however, it is all shifted: we have sums $\Sigma A.B$ and products $\Pi A.B$.

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash \Sigma(x : A).B}$$

$$\frac{\Gamma, x : A \vdash B \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[x \leftarrow a]}{\Gamma \vdash \langle a, b \rangle : \Sigma(x : A).B}$$

$$\frac{\begin{array}{cc} & \Gamma \vdash p : \Sigma(x : A).B \\ \Gamma, x : \Sigma(x : A).B \vdash P & \Gamma, a : A, b : B[x \leftarrow a] \vdash t : P[x \leftarrow \langle a, b \rangle] \end{array}}{\Gamma \vdash \begin{array}{l} \textsf{match}\ p\ \textsf{return}\ x.P\ \textsf{with} \\ \mid \langle a,b \rangle \implies t \\ \textsf{end} \end{array} : P[x \leftarrow p]}$$

This captures the computational content of existentials: from a proof of existence you can extract a witness (the a in $\langle a, b \rangle$). This is another important example of constructivism.

Once more, this is the *positive* presentation of Σ-types. With this presentation it is also possible to restrict P in case we do not want to make it possible to extract the witness.

It is also worth noting that simple products are a particular case of Σ-type: $A \times B$ can be defined as $\Sigma(\_ : A).B$.

The _ is there to note that B does not depend on the variable in A.

Indexed inductive types

Inductive types can have parameters, but they can also have *indices*. They are similar to parameters in that they are arguments to the inductive type but, while parameters are always the same in the *conclusion* of the type of a constructor, indices can vary. I will show the two most talked about cases of indexed inductive types: vectors and equality.

In all the examples I showed, the parameters are *uniform*, *i.e.* they are the same everywhere, but there can be non uniform occurrences of them where the recursive argument is at a different parameter.

Vectors. Vectors are length-indexed lists: $\mathsf{vec}_A\ n$ is type of lists of length n whose elements inhabit A. In this case, A is a parameter and n is an index.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash n : \mathsf{nat}}{\Gamma \vdash \mathsf{vec}_A\ n} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{vnil} : \mathsf{vec}_A\ \boxed{0}}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash n : \mathsf{nat} \qquad \Gamma \vdash v : \mathsf{vec}_A\ n}{\Gamma \vdash \mathsf{vcons}\ a\ n\ v : \mathsf{vec}_A\ \boxed{(\mathbf{S}\ n)}}$$

I highlight the index for the constructors to show how it differs. vnil is the only list of length 0 and a vcons always adds one element to a vector.

$$\frac{\begin{array}{c}\Gamma \vdash v : \mathsf{vec}_A\ n \\ \Gamma, p : \mathsf{nat}, x : \mathsf{vec}_A\ p \vdash P \qquad \Gamma \vdash u_{\mathsf{vnil}} : P[p \leftarrow 0, x \leftarrow \mathsf{vnil}] \\ \Gamma, a : A, m : \mathsf{nat}, w : \mathsf{vec}_A\ m \vdash u_{\mathsf{vcons}} : P[p \leftarrow \mathbf{S}\ m, x \leftarrow \mathsf{vcons}\ a\ m\ w]\end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{match}\ v\ \mathsf{return}\ p.x.P\ \mathsf{with} \\ \mid \mathsf{vnil} \quad \Longrightarrow \quad u_{\mathsf{vnil}} \\ \mid \mathsf{vcons}\ a\ m\ w \quad \Longrightarrow \quad u_{\mathsf{vcons}} \\ \mathsf{end}\end{array} : P[p \leftarrow n, x \leftarrow v]}$$

The pattern-matching this time binds *two* variables in the return predicate: the variable representing the matched term as usual, but also the index (*i.e.* the length of the vector)! This is because it varies depending on the branch.

Vectors are pretty useful because, since you account for the length, you can write safer and more precise functions. For instance, the tail function on lists would land in an option type (in the case the list is nil), but here we can easily say it should only take some $\mathsf{vec}_A\ (\mathbf{S}\ n)$ as argument, ruling out the empty vector completely.

$$\begin{array}{l}\lambda\ A\ (n : \mathsf{nat})\ (v : \mathsf{vec}_A\ (\mathbf{S}\ n)). \\ \quad \mathsf{match}\ v\ \mathsf{return}\ p.x.\ \mathsf{vec}_A\ n\ \mathsf{with} \\ \quad \mid \mathsf{vcons}\ a\ m\ w \quad \Longrightarrow \quad w \\ \quad \mathsf{end}\end{array}$$

The return type is also more precise: the tail of a list of length $\mathbf{S}\ n$ is of length n.

Notice how I did not even provide a branch for vnil because it will always by ill-typed.This is unfortunately impossible in vanilla Coq but is supported in Agda natively and can be achieved in Coq with Program or the Equations plugin [20, 21]. With the typing rules I gave it is still possible to do it, it is just that we have to conclude using the elimination of $\bot$ in the vnil branch.

By vanilla Coq I mean Coq without anything extra like plugins.

[20]: Sozeau (2010), 'Equations: A Dependent Pattern-Matching Compiler'
[21]: Sozeau et al. (2019), 'Equations reloaded: high-level dependently-typed functional programming and proving in Coq'

Equality. As equality is special I will only give a brief definition before we discuss it in more detail later in this chapter. In Chapter 6 (Flavours of type theory) I will also show different notions of equality. We already have one notion of equality in conversion, sometimes called definitional equality. This notion is external in type theory and one cannot reason about conversion. As such we want a notion of equality on which we can reason, one which is internal to type theory. We talk this time of *propositional equality*. We want the type $u =_A v$ to represent equality of terms u and v of type A. When can you prove that u and v are *equal*? When they are the same!

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

$$\frac{\begin{array}{c}\Gamma \vdash u : A \qquad \Gamma \vdash v : A \qquad \Gamma \vdash e : u =_A v \\ \Gamma, x : A, p : u =_A x \vdash P \qquad \Gamma \vdash t : P[x \leftarrow u, p \leftarrow \mathsf{refl}_A\ u]\end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{match}\ e\ \mathsf{return}\ x.p.P\ \mathsf{with} \\ \mid \mathsf{refl} \implies t \\ \mathsf{end}\end{array} : P[x \leftarrow v, p \leftarrow e]}$$

This notion of 'being the same', is conversion: if u and v are convertible then $\mathsf{refl}_A\ u$ is a proof of $u =_A v$. This comes from the fact that, if $u \equiv v$, we have $u =_A u \equiv u =_A v$.

$\mathsf{refl}_A\ u$ is the reflexivity proof and it *unifies* u and v in its return type. Hence you can conclude that v is an index while A and u are parameters. The pattern-matching in the case where P does not depend on the equality p helps us recover Leibniz's principle stating that $u = v$ means that for every P, $P\ x \longrightarrow P\ y$. In particular this allows us to *rewrite* along equalities, *i.e.* changing something for something else equal to it in an expression.

Generally speaking indices are complex to deal with and it is related to how equality is complex in type theory, hence the multiple approaches there are to it.

Other inductive types

There are other kinds of inductive types and I shall go briefly over some of them.

Mutual inductive types. Sometimes you want to define two notions at the same time because they are linked or interleaved. Take the notion of odd and even numbers for instance. They can both be defined independently or one built on top of the other, but they can also be defined mutually: 0 is even, when n is even, $n+1$ is odd and when n is odd, $n+1$ is even. In Coq it goes like this.

```
Inductive even : nat -> Type :=
| even_0 : even 0
| even_S : forall n, odd n -> even (S n)

with odd : nat -> Type :=
| odd_S : forall n, even n -> odd (S n).
```

To deal with them you need mutual fixed-points.

Inductive inductive types and induction recursion. Pushing even further in that direction, come inductive inductive types [22] where the type of inductive types can also depend on the mutually defined inductive types. Induction recursion [23] allows you to define inductive types mutually with functions acting on them.

In the *mutual* case, only the constructors could mention the other inductive types.

[22]: Forsberg et al. (2010), 'Inductive-inductive definitions'

[23]: Dybjer (2000), 'A general formulation of simultaneous inductive-recursive definitions in type theory'

Both these features are not available in Coq yet, but are present in Agda.

5.2 Coinductive types and records

Not all data is best represented inductively, instead of constructors it is possible to talk about *destructors*, *i.e.* observations (in other words we are more interested in what to do with the data rather than how to build it).

Records

Records are datatypes containing different fields. In Coq they can be defined like this.

```
Record prod A B := pair {
  fst : A ;
  snd : B
}
```

This corresponds to another way of defining $A \times B$.

Historically in Coq, records are defined as inductive types with one constructor. The above definition is in fact syntactic sugar for

```
Inductive prod A B :=
| pair : A -> B -> prod A B.

Definition fst A B (p : prod A B) : A :=
  match p with
  | pair a b => a
  end.

Definition snd A B (p : prod A B) : B :=
  match p with
  | pair a b => b
  end.
```

This is what we call *positive* records, corresponding to the presentation of $A \times B$ and $\Sigma(x : A).B$ I gave earlier.

There is however an option in Coq to instead use a *negative* presentation:

```
Set Primitive Projections.
```

Once it is set the definition above of the record becomes primitive. The constructor `pair` is the one that becomes a definition:

```
Definition pair A B (a : A) (b : B) : prod A B :=
  {|
    fst := a ;
    snd := b
  |}.
```

The data is accessed using the projections `fst` and `snd`. They are *destructors* in that they correspond to ways to observe pairs, by opposition to the constructors of pairs of the positive version: `pair`.

I will now give the typing rules of negative Σ-types.

$$\frac{\Gamma \vdash A \qquad \Gamma, x : A \vdash B}{\Gamma \vdash \Sigma(x : A).B} \qquad \frac{\Gamma \vdash p : \Sigma(x : A).B}{\Gamma \vdash \pi_1\ p : A} \qquad \frac{\Gamma \vdash p : \Sigma(x : A).B}{\Gamma \vdash \pi_2\ p : B[x \leftarrow \pi_1\ p]}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A \vdash B \qquad \Gamma \vdash b : B[x \leftarrow a]}{\Gamma \vdash \langle a, b \rangle : \Sigma(x : A).B}$$

The way to think about them is that when providing $\langle a, b \rangle$ you are actually saying how the term will behave when projected (by either π_1 and π_2). This is illustrated by the computation rules.

$$\begin{array}{lcl} \pi_1\ \langle a, b \rangle & \rightarrow & a \\ \pi_2\ \langle a, b \rangle & \rightarrow & b \end{array}$$

I already hinted at this, but with this presentation it is not possible to restrict usage of π_1, it is always possible to recover the witness.

Coinductive types

Coinductive types are the dual of inductive types and are used to represent potentially infinite data. Streams of data are one such example: they are infinite lists, like lists they have heads and tails, except there always is a tail.

$$\frac{\Gamma \vdash A}{\Gamma \vdash \mathsf{stream}\ A} \qquad \frac{\Gamma \vdash s : \mathsf{stream}\ A}{\Gamma \vdash \mathsf{head}\ s : A} \qquad \frac{\Gamma \vdash s : \mathsf{stream}\ A}{\Gamma \vdash \mathsf{tail}\ s : \mathsf{stream}\ A}$$

In the definition of the coinductive type of streams we define destructors: head and tail are means to inspect a stream. We cannot use constructors to describe a stream however, as it is infinite. There are several ways to build such data but the one I find most convincing and beautiful is to use a dual approach to pattern-matching: *copattern-matching* [24], usually in combination with co-fixed-point. In Agda, the stream consisting only of zeroes can be defined as follows:

[24]: Abel et al. (2013), 'Copatterns: programming infinite structures by observations'

```
zeroes : Stream Nat
head zeroes = zero
tail zeroes = zeroes
```

while the stream of natural numbers starting at n is

```
seq : Nat → Stream Nat
head (seq n) = n
tail (seq n) = seq (succ n)
```

In will not go into further detail about those, coinductive types are not really well-behaved in Coq when they are not presented negatively, *i.e.* with primitive projection, so we mostly ignore them for the time being.

5.3 Equality

I will here describe some usual definition and concepts associated with equality in type theory, mostly from the point of view of Coq. Again, several notions of equality are of interest and I study them briefly in Chapter 6 (Flavours of type theory).

Basic properties of equality

Equality, by all means, should be a congruence, and it is. It might be surprising given that we *only* ask for it to be reflexive. Induction helps us recover these properties.

We can build a term eqrec corresponding to the eliminator for equality.

$$\begin{aligned}\mathsf{eqrec} \quad : \quad & \Pi\, A\ (u : A)\ (P : \Pi\,(x : A).\ u =_A x \longrightarrow \mathsf{Type}). \\ & P\ u\ (\mathsf{refl}_A\ \ u) \longrightarrow \\ & \Pi\, v\ (e : u =_A v).\ P\ v\ e\end{aligned}$$

Its definition is

$$\begin{aligned}&\lambda\ A\ u\ P\ t\ v\ e. \\ &\mathsf{match}\ e\ \mathsf{return}\ x.p.\ P\ x\ p\ \mathsf{with} \\ &\mid \mathsf{refl} \quad \Longrightarrow \quad t \\ &\mathsf{end}\end{aligned}$$

Reminder: Equality

Equality is given by the following rules.

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : A}{\Gamma \vdash u =_A v}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash u : A \\ \Gamma \vdash v : A \qquad \Gamma \vdash e : u =_A v \\ \Gamma, x : A, p : u =_A x \vdash P \\ \Gamma \vdash t : P[x \leftarrow u, p \leftarrow \mathsf{refl}_A\ u]\end{array}}{\Gamma \vdash \begin{array}{l}\mathsf{match}\ e \\ \mathsf{return}\ x.p.P \\ \mathsf{with} \\ \mid \mathsf{refl}\ \Rightarrow\ t \\ \mathsf{end}\end{array} : P[x \leftarrow v, p \leftarrow e]}$$

It is often the case that P does not mention the equality, in which case we get the simpler notion of *transport*.

$$\begin{aligned}\mathsf{transport} \quad : \quad & \Pi\, A\ (P : A \longrightarrow \mathsf{Type})\ (u\ v : A). \\ & u =_A v \longrightarrow \\ & P\ u \longrightarrow \\ & P\ v \\ := \quad & \lambda\ A\ P\ u\ v\ e\ t. \\ & \mathsf{match}\ e\ \mathsf{return}\ x.p.\ P\ x\ \mathsf{with} \\ & \mid \mathsf{refl} \quad \Longrightarrow \quad t \\ & \mathsf{end}\end{aligned}$$

I changed the order of the arguments a bit compared with eqrec because I find it more legible this way, and because I can, thanks to less dependencies.

Transport will be a good stepping stone to show that equality is a congruence.

$$\begin{array}{lll}
\mathsf{sym} & : & \Pi\, A\ (u\ v : A). \\
 & & \quad u =_A v \to \\
 & & \quad v =_A u \\
 & := & \lambda\, A\ u\ v\ e. \\
 & & \quad \mathsf{transport}\ A\ (\lambda x.\ u =_A x)\ u\ v\ e\ (\mathsf{refl}_A\ u)
\end{array}$$

$$\begin{array}{lll}
\mathsf{trans} & : & \Pi\, A\ (u\ v\ w : A). \\
 & & \quad u =_A v \to \\
 & & \quad v =_A w \to \\
 & & \quad u =_A w \\
 & := & \lambda\, A\ u\ v\ w\ e_1\ e_2. \\
 & & \quad \mathsf{transport}\ A\ (\lambda x.\ u =_A x)\ v\ w\ e_2\ e_1
\end{array}$$

$$\begin{array}{lll}
\mathsf{cong} & : & \Pi\, A\ B\ (f : A \to B)\ (u\ v : A). \\
 & & \quad u =_A v \to \\
 & & \quad f\ u =_B f\ v \\
 & := & \lambda\, A\ B\ f\ u\ v\ e. \\
 & & \quad \mathsf{transport}\ A\ (\lambda x.\ f\ u =_B f\ x)\ u\ v\ e\ (\mathsf{refl}_B\ (f\ u))
\end{array}$$

Notice how

$$\mathsf{transport}\ A\ (\lambda x.\ u =_A x)\ u\ v\ e$$

has type

$$u =_A u \to u =_A v$$

Independent principles

I will now present some usual notions associated with equality. I call them independent because they are neither provable in nor contradictory to the theory of Coq. This will be studied in Part 'Elimination of Reflection'.

Uniqueness of Identity Proofs (UIP). UIP is a principle saying that all proofs of an equality are themselves equal.

$$\mathsf{uip} : \Pi\, A\ (x\ y : A)\ (e\ e' : x =_A y).\ e = e'$$

In particular—or equivalently—all proofs of $x = x$ are equal to the reflexivity proof, this is Streicher's axiom K:

$$\mathsf{K} : \Pi\, A\ (x : A)\ (e : x =_A x).\ e =_{x =_A x} \mathsf{refl}_A\ x$$

This can be very useful to deal with higher-level equalities. This principle can be assumed in Coq but is not provable. In Agda however this principle holds—it can proven by pattern-matching on the equality—and has to be deactivated with an option [25] which restricts pattern-matching.

[25]: Cockx et al. (2016), 'Eliminating dependent pattern matching without K'

Functional extensionality (funext). Funext states that two functions are equal whenever they are pointwise equal.

$$\begin{array}{lll}
\mathsf{funext} & : & \Pi\, A\ B\ (f\ g : A \to B). \\
 & & \quad (\Pi x.\ f\ x =_B g\ x) \to \\
 & & \quad f =_{A \to B} g
\end{array}$$

There is also a dependent variant of funext.

$$\begin{aligned}\mathsf{funextD} \quad : \quad & \Pi\, A\, (B : A \to \mathsf{Type})\, (f\ g : \Pi.(x : A).\ B\ x).\\ & (\Pi x.\ f\ x =_{B\ x} g\ x) \to\\ & f =_{\Pi(x:A).\ B\ x}\ g\end{aligned}$$

This corresponds to the usual understanding of equality on functions. Some systems support it natively with computational content like Observational Type Theory (OTT) [26], Setoid Type Theory (STT) [27] or Cubical Type Theory (CubicalTT) [28, 29].

[26]: Altenkirch et al. (2007), 'Observational equality, now!'
[27]: Altenkirch et al. (2019), 'Setoid Type Theory—A Syntactic Translation'
[28]: Bezem et al. (2013), 'A Model of Type Theory in Cubical Sets'
[29]: Cohen et al. (2016), 'Cubical type theory: a constructive interpretation of the univalence axiom'

Heterogenous equality

Sometimes, you want to be able to compare terms of different types. This will happen typically with dependent types, *e.g.* with vectors $u : \mathsf{vec}_A\ (n + m)$ and $v : \mathsf{vec}_A\ (m + n)$. It feels like we should be able to compare them since their types are equal, unfortunately they are usually not convertible and as such $u = v$ would be ill-typed.

In such cases one has to consider *heterogenous* equalities—as opposed to the regular homogenous equality. John Major equality (JMeq) is one such notion of heterogenous equality, introduced by [30]. It is pretty similar to the equality type I showed earlier, except it uses an extra index for a second type.

[30]: McBride (2000), 'Dependently typed functional programs and their proofs'

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash a\ {}_A{\cong}_B\ b} \qquad\qquad \frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{hrefl}_A\ a : a\ {}_A{\cong}_A\ a}$$

As usual, it features pattern-matching. Unfortunately, it does not allow us to recover the property that *heterogenous* equalities where the two types are in fact the same like $u\ {}_A{\cong}_A\ v$, imply *homogenous* equality $u =_A v$. This has to be added as an extra axiom that is equivalent to UIP.

For this reason, I personally prefer the more explicit encoding of heterogenous equality with Σ-types:

$$t\ {}_T{\cong}_U\ u := \Sigma(p : T = U).\, p_*\ t = u$$

where we write p_* for the map from T to U, it is a notation for

$$p_* := \mathsf{transport}\ \mathsf{Type}\ (\lambda x.x)\ T\ U\ p.$$

When we have $u\ {}_A{\cong}_A\ v$, it means we have a proof of equality $p : A = A$ and a proof of $p_*\ u = v$. It becomes apparent that we need UIP to rewrite p to refl such that the second equality becomes $u = v$ as wanted.

Lemma 5.3.1 *Assuming UIP, if $\Gamma \vdash e : u\ {}_A{\cong}_A\ v$ then there exists p such that $\Gamma \vdash p : u =_A v$.*

Here the notion of heterogenous can be either of the two I presented as they are equivalent.

Flavours of type theory 6

Type theory comes in many different flavours and shapes, different formulations and properties. I will not try to be exhaustive but I will try to cover the main kind of dependent type theories I have encountered. I will not attempt to define properly the notion of *type theory*, there is work on this [31] but it is still a bit early to grasp the concept fully.

[31]: Haselwarter (2020), 'Effective Metatheory for Type Theory'

6.1 Computation and type theory

The first prism through which to see type theory through can be that of computation. Indeed, not all type theories feature it to the same extent. Though for some people in computation resides the essence of type theory, it is still worth it to investigate theories where conversion is defined differently.

Intensional Type Theory

Intensional Type Theory (ITT) is the name given to a wide range of type theories actually. Those could be described in the setting of Pure Type Systems (PTSs) [32]. The theories behind the proof assistants Coq and Agda [7]–respectively Predicative Calculus of Cumulative Inductive Constructions (PCUIC) and Martin-Löf Type Theory (MLTT)[1]—are variants of ITT.

[32]: Barendregt (1991), 'Introduction to generalized type systems'

[7]: Norell (2007), *Towards a practical programming language based on dependent type theory*

1: Note that MLTT also has various forms.

A PTS is a pretty basic type theory, it is parametrised by a collection of sorts $\mathcal{S}$, with so-called *rules* (R) and *axioms* (Ax). Its syntax features λ-abstractions, applications, variables, Π-types and sorts.

$$\begin{array}{lcl}
s & \in & \mathcal{S} \\
T, A, B, t, u, v & ::= & x \mid \lambda(x : A).t \mid t\ u \mid \Pi(x : A).B \mid s \\
\Gamma, \Delta & ::= & \bullet \mid \Gamma, x : A
\end{array}$$

Their computational behaviour is defined by a reduction relation ($\rightarrowtriangle$) which is the contextual closure of the β-reduction.

$$(\lambda(x : A).t)\ u \rightarrowtriangle_\beta t[x \leftarrow u]$$

For instance $\lambda(x : A).(\lambda(y : B)yx)\ t \rightarrowtriangle \lambda(x : A).tx$

The typing rules involve the rules and axioms we mentioned earlier.

$$\frac{(s, s') \in \mathsf{Ax}}{\Gamma \vdash s : s'} \qquad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2 \qquad (s_1, s_2, s_3) \in \mathsf{R}}{\Gamma \vdash \Pi(x : A).B : s_3}$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash \Pi(x : A).B : s}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A).B}$$

$$\frac{\Gamma \vdash t : \Pi(x : A).B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x \leftarrow u]} \qquad \frac{\Gamma \vdash t : A \qquad A \equiv B \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}$$

Axioms determine typing of sorts, and rules what dependent products are allowed. The last rule is the conversion rule, it is the rule that involves computation: basically you can exchange two computationally equal types in a typing judgement. The $\Gamma \vdash B : s$ part is to make sure the type we want to substitute still makes sense. In this case, conversion ($\equiv$) is defined as the reflexive, symmetric, transitive closure of reduction.

$t \equiv u$ is defined as $t\ (\leftarrow . \rightarrow)^{\star}\ u$

When talking about ITT we usually mean an extension of this with more concepts like some base (inductive) types and computation rules on their eliminators (pattern-matching). The conversion rule is also not always strictly derived from the reduction alone, it often includes η-rules, the most common being η-expansion of functions.

See Chapter 5 (Usual definitions in type theory).

$$f \equiv_\eta \lambda(x : A).f\ x$$

For it to make sense, expansion has to be limited to functions as we don't want to η-expand the natural number 0 to $\lambda(x : A).0\ x$. This means we need typing information[2], this is why in certain systems—like Agda—the conversion is also typed. The relation between typed and untyped conversion has been explored at several occasions [33]. Coq manages to verify η-conversion for functions and records without relying on a typed-conversion as we will see later in Chapter 24 (Conversion).

2: Also, the A—domain of the function—has to be inferred somehow.

[33]: Doorn et al. (2013), 'Explicit convertibility proofs in pure type systems'

For instance, η for pairs is $p \equiv (p.1, p.2)$ where $p.1$ and $p.2$ are the first and second projections of p.

Extensional Type Theory

Extensional Type Theory (ETT) is an extension of ITT where conversion is extended to capture all provable equalities, this principle is called *reflection (of equality)*. This of course implies that the considered ITT is equipped with an equality type.

Definition 6.1.1 *Reflection Rule*

$$\frac{\Gamma \vdash_{\mathsf{x}} e : u =_A v}{\Gamma \vdash_{\mathsf{x}} u \equiv v : A}$$

Here I use $\vdash_{\mathsf{x}}$ to mark that this is a judgement in ETT.

As you can see, this time I opted for a typed conversion, I think it makes more sense since the conversion is more semantical than syntactical. Also, you can note the little x subscript, its purpose is to mark the judgement as being *ex*tensional. Andromeda 1 and NuPRL implement variants of Extensional Type Theories [34, 35]. To see its usefulness, we are going to look at the definition of reversal of vectors in Coq, using an accumulator for the definiton to be tail-recursive.

[34]: Bauer et al. (2016), *The 'Andromeda' prover*
[35]: Constable et al. (2014), *The NuPrl system, PRL project*

```
Definition vrev {A n m} (v : vec A n) (acc : vec A m)
  : vec A (n + m) :=
  match v with
  | vnil => acc
  | vcons a n v => vrev v (vcons a m acc)
  end.
```

The recursive call of `vrev` returns a vector of length `n + S m` where the context expects one of length `S n + m`. In ITT and Coq, these types are not convertible, and thus the definition is not accepted, even though it feels like it is the right definition. ETT solves this problem by exploiting the fact that `n + S m = S n + m` is provable. You can still define it in Coq, but you have to explicitly transport along the above-mentioned equality which can result in some problems while reasoning on the resulting function and inconveniences overall.

ETT is not the ultimate solution however and suffers from many drawbacks the main of which being that type-checking is not decidable as we shall see in Chapter 11 (What I mean by elimination of reflection). Type-checking is usually decidable in proof assistants, which is part of what makes them usable in practice, I discuss this notion in Chapter 7 (Desirable properties of type theories). We will also explore the relation between ETT and ITT in Part 'Elimination of Reflection'.

Weak Type Theory

A Weak Type Theory (WTT) is on the other end of the spectrum: instead of extending conversion with everything that can be proven equal, conversion is removed altogether. Computation (like β-reduction) is now handled by propositional equality alone, and conversion of types is done using transports along said equality.

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash \beta(t, u) : (\lambda(x : A).\ t)\ u =_{B[x \leftarrow u]} t[x \leftarrow u]}$$

$$\frac{\Gamma \vdash T_1 : s \qquad \Gamma \vdash T_2 : s \qquad \Gamma \vdash p : T_1 = T_2 \qquad \Gamma \vdash t : T_1}{\Gamma \vdash \mathsf{transport}_{T_1,T_2}(p, t) : T_2}$$

This time it is a bit hard to advertise it for practical use in a proof assistant, it is nonetheless interesting. For one, its meta-theory is that much simpler, an even more attractive fact once combined with a translation from ITT (or ETT) to WTT as is the object of Part 'Elimination of Reflection'. Another point worth mentioning is that it really crystallises the notion that proofs are really just terms and do not require extra machinery to make sure they are indeed proofs (even when conversion is decidable, it might takes eons before two types are verified to be convertible) [8].

[8]: De Bruijn (1991), 'A plea for weaker frameworks'

One might also be tempted to call it minimal, but in order to simulate the congruence aspect of conversion, we have to extend the theory with principles to allow equalities under binders, and this has to be done for each binder (once for λ-abstractions—the usual functional extensionality—and once for Π-types—much less standard—at least).

6.2 Focus on the theory behind **Coq**

Coq is originally based on the Calculus of Constructions (CoC), or the Calculus of Inductive Constructions (CIC) [36], though it is nowadays rather called the Predicative Calculus of Cumulative Inductive Constructions (PCUIC). It is a variant of ITT but as the name suggests, it has several extra features, hinted at by the words *cumulative*, *inductive* and *predicative*.

[36]: Bertot et al. (2004), *Interactive Theorem Proving and Program Development*

Universes in **Coq**

Coq features a *predicative* hierarchy of universes $(\mathsf{Type}_i)_{i \in \mathbb{N}}$ such that $\mathsf{Type}_i : \mathsf{Type}_j$ for any $i < j$. The presence of several universes included in one another is not there for fun, it is there to circumvent Russell's paradox which shows it is inconsistent to have $\mathsf{Type} : \mathsf{Type}$. The Coq user however does not usually have to deal with those and instead relies on the so-called *typical ambiguity* [39]: in the Coq proof assistant one will simply write Type and Coq will infer constraints to know whether it is possible to appoint each occurrence of Type a *level*, *i.e.* a natural number, such that the universes are used consistently. For instance, $\Pi(A : \mathsf{Type}).\ \mathsf{Type} : \mathsf{Type}$ will actually correspond to

For type theory we would more likely talk about Girard's paradox [37] or Hurkens' paradox [38].

[37]: Girard (1972), 'Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur'
[38]: Hurkens (1995), 'A simplification of Girard's paradox'

[39]: Harper et al. (1991), 'Type checking with universes'

$$\Pi(A : \mathsf{Type}_i).\ \mathsf{Type}_j : \mathsf{Type}_k$$

with constraints $i < k$ and $j < k$, *i.e.* quantifying over a universe raises the universe level of the whole expression.

Alongside this hierarchy comes another universe, Prop which is the universe of propositions. This universe is *impredicative*. Impredicativity means that the definition of an object can quantify over the object itself. In the case of Prop it comes from the fact that a proposition (*i.e.* a type in Prop) can quantify over propositions: *e.g.* $\forall (P : \mathsf{Prop}).\ P$ is still a proposition. Actually impredicativity of Prop is a bit stronger as it accepts *any* quantification, so that $\forall (n : \mathbb{N}).\ n = n$ is also a proposition for instance. Prop however is of type Type_i for any i.

Note that the type $\forall (P : \mathsf{Prop}).\ P$ is actually the empty type or false proposition so it is reassuring that it *is* a proposition.

Recently, a new universe of propositions has been introduced [40]: SProp. This time, it is a universe of *strict* propositions in the sense that any two proofs of the same proposition are considered to be convertible. In other words, there is no computational content to proofs of strict propositions, and as such proofs are *irrelevant*, only their mere existence matters. This builds on the principle of proof irrelevance that is an axiom often considered, of type $\Pi(P : \mathsf{Prop})\ (x\ y : \mathsf{Prop}).\ x = y$. Although this addition is a really interesting subject,

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

For $P : \mathsf{SProp}$ and $p_1 : P$ and $p_2 : P$ then $p_1 \equiv p_2$.

it goes a bit beyond the scope of this thesis, in particular I do not consider it in my formalisations (for now).

When I introduced the typing rules for Prop and Type I showed that each universe can be typed in several other universes (actually in each case there are an infinite number of possibilities). The reality is even more intricate than that and it has to do with the word '*cumulative*' found in the name of the system. Coq features a notion of subtyping that is limited to universes and that is called *cumulativity*. In essence it says that $\mathsf{Type}_i \preceq \mathsf{Type}_j$ whenever $i \le j$, that is that bigger universes contain the smaller universes. Cumulativity is given by the following rules.

$$\frac{}{\Gamma \vdash \mathsf{Type}_i \preceq \mathsf{Type}_j}\,(i \le j) \qquad \frac{}{\Gamma \vdash \mathsf{Prop} \preceq \mathsf{Type}_i}$$

$$\frac{\Gamma \vdash A \equiv A' \qquad \Gamma, x : A \vdash B \preceq B'}{\Gamma \vdash \Pi\,(x : A).\ B \preceq \Pi\,(x : A').\ B'} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \preceq B \qquad \Gamma \vdash B}{\Gamma \vdash t : B}$$

Notice how the usual conversion rule is replaced by a cumulativity rule. Also, as opposed to usual subtyping definitions, there is no subtyping in contravariant positions, but rather conversion.

This convenience allows the user to forget as much as possible about universes.

Finally there is another point regarding universes that is not hinted at in the name: *universe polymorphism*. This feature introduced by [41] is complementary to cumulativity and allows us to consider definitions that quantify over universe levels (the i in Type_i). For instance, it allows us to write a truly polymorphic identity function

[41]: Sozeau et al. (2014), 'Universe polymorphism in Coq'

$$\mathsf{id}_i := \lambda(A : \mathsf{Type}_i)\ (x : A).\ x$$

Here, the i subscript is to bind level i in the expression. Notice that it is attached to the constant definition and not on the right-hand side; that is because universe levels are only quantified in a prenex-style, meaning you must quantify over *all* universe levels before introducing other variables. In particular there is no type of polymorphic identities.

One might notice that currently, universe polymorphism only talks about Type and does not make mention of Prop. This is a shortcoming that could be fixed using also *sort polymorphism* but at the time of writing of this document, another technology is used in place: *template polymorphism*. It is a heuristic orthogonal to universe polymorphism to define constants both at the Type-level and at the Prop-level at the same time. However, this seems to be the source of many bugs[3] and should be removed from Coq's kernel at some point in the relatively near future.

3: They can be found at `https://github.com/coq/coq/blob/bb6e78ef2c/dev/doc/critical-bugs#L98`.

Inductive types

Then comes the *inductive* part of the name. I already described inductive types in Section 5.1 (Inductive types and pattern-matching) so I will not repeat myself but their presence is the main justification for the 'I' in CIC. However, PCUIC is named that way because it features *cumulative inductive types* [42] (CIC—and even CoC—already has cumulativity). The idea is to extend the cumulativity relation beyond

[42]: Timany et al. (2018), 'Cumulative inductive types in Coq'

sorts to inductive types as well. I am going to use the same example than the paper introducing them: a record [4] definition of categories:

$$\text{Category}_{i,j} \coloneqq \{\text{Obj} : \text{Type}_i \ ; \ \text{Hom} : \text{Obj} \longrightarrow \text{Obj} \longrightarrow \text{Type}_j \ ; \ \ldots\}$$

4: Exploiting the fact that (non-primitive) records are actually inductive types with one constructor in Coq.

In PCUIC, you have the following cumulativity rule.

$$\frac{i \leq i' \qquad j \leq j'}{\text{Category}_{i,j} \preceq \text{Category}_{i',j'}}$$

This allows for more flexibility when dealing with polymorphic definitions.

let-bindings

This is not limited to Coq, but let-bindings behave differently than in simply typed programming languages. In languages like OCaml a let-binding can be simulated by a β-redex.

```
let x = u in t
```

will be equivalent to

```
(fun x -> t) u
```

It might be that the compiler does things a bit differently in both cases for optimisations purposes, but at least *in principle* they should be equivalent.

as both will evaluate `u` before substituting it for `x` in `t`. This works mainly because OCaml is a Call by Value (CbV) language. In Coq, replacing the first by the second will not always yield a well-typed expression. In the Coq expression `let x := u in t`, the term `t` actually lives in an extended context with `x := u`, meaning that the definition is also stored, not just its type and name. This information can be used to *recall* that `x` is convertible to `u`. The example below illustrates a case where the transformation yields an ill-typed term.

In OCaml we have $x : A \vdash t$.

```
Section Let.

  Context (u : nat).
  Context (f : u = u -> nat).

  Definition foo :=
    let x := u in f (eq_refl x).

  (** The term "eq_refl" has type "x = x" while it is expected
      to have type "u = u" (cannot unify "x" and "u").
  *)
  Fail Definition bar :=
    (fun x => f (eq_refl x)) u.

End Let.
```

`Fail` is just there to attest that the following definition will fail.

6.3 Equality in type theories

Section 5.3 (Equality) is already dedicated to the definition of equality in type theory, but there I only presented *one way* of dealing with equality, the one that is prevalent in type theory based proof assistants like Agda and Coq.

In Coq equality is defined as follows, *i.e.* in the way I presented in Chapter 5.

```
Inductive eq (A : Type) (u : A) : A -> Prop :=
| eq_refl : eq A u u.
```

This has some shortcomings[5] like equalities between equalities, and between functions are not really accounted for, in the sense that it is hard—often impossible—to prove them. This gives rise to axioms like UIP and funext, but axioms lack computational content.

5: At least for some people, many are content with the current status too.

UIP is given computational content in Agda, it is feasible so it is not really an argument against this definition.

Variants of type theory give different definitions and properties to equality.

Observational Type Theory (OTT)

The idea behind OTT [26] is that equality should be defined for each type specifically and correspond to observational equivalences. For functions, equality is *defined* as pointwise equality.

[26]: Altenkirch et al. (2007), 'Observational equality, now!'

$$f =_{A \to B} g \coloneqq \Pi\,(x : A).\ f\ x =_B f\ y$$

In this world, reflexivity becomes a property and is no longer part of the definition. More recently, this work has been extended to Setoid Type Theory (STT) [27] to support heterogenous equality and better fit the Setoid model [43].

[27]: Altenkirch et al. (2019), 'Setoid Type Theory—A Syntactic Translation'

[43]: Altenkirch (1999), 'Extensional equality in intensional type theory'

Homotopy Type Theory (HoTT)

HoTT [44] is rather based on the idea that UIP should not hold and that instead, higher equalities matter. Some types still enjoy a certain form of UIP however: they are called sets, or homotopy sets (hSet). This actually defines a hierarchy of types, by looking at their type of equality.

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

As the name suggests, equality is coming from an intuition of algebraic topology: homotopy. An equality $u = v$ is thus seen as a path connecting u and v.

Univalence. One of the leading principles coming with HoTT is the univalence axiom. The idea is to lift the folklore idea in mathematics that isomorphic objects are the *same*, into a well-defined notion: equivalent types are equal. The axiom is a bit more precise in that it states that the trivial map taking an equality of types $A = B$ to an equivalence $A \cong B$ is itself an equivalence. Univalence allows us to *transport* a result onto another equivalent setting. An interesting consequence of the univalence axiom is that it implies funext.

Equivalence $A \cong B$ means roughly that there exist maps $f : A \to B$ and $g : B \to A$ such that they are both inverse of each other.

Once again, this is an *axiom* and as such, it does not compute. There are ways [45] to overcome this in Coq which has its own HoTT library [46] but this still does not give a full-fledged computation rule for univalence. To achieve this, some proof assistants implement Cubical Type Theory.

[45]: Tabareau et al. (2018), 'Equivalences for free: univalent parametricity for effective transport'

[46]: Bauer et al. (2017), 'The HoTT library: a formalization of homotopy type theory in Coq'

Cubical Type Theory (CubicalTT). CubicalTT [29] does not build on regular type theory with an inductive equality but instead defines it differently, with intuition coming from cubical sets [28]. This time the analogy with paths is put to the extreme: there is a special construct $\mathbb{I}$ morally representing the interval $[0;1]$. Its end points are $\mathbb{0}$ and $\mathbb{1}$. An equality between two terms u and v of type A is now a map f from $\mathbb{I}$ to A such that $f\ \mathbb{0} \equiv u$ and $f\ \mathbb{1} \equiv v$. If we allow the map f to be dependently typed, we can easily have heterogenous equality:

$$f : (i : \mathbb{I}) \to A\ i$$

f is now an equality between $f\ \mathbb{0} : A\ \mathbb{0}$ and $f\ \mathbb{1} : A\ \mathbb{1}$. The main interest of this is that it gives computational content to univalence which is no longer an axiom but a theorem. Univalence implies funext, meaning that funext gets a computational content from it.

[29]: Cohen et al. (2016), 'Cubical type theory: a constructive interpretation of the univalence axiom'

[28]: Bezem et al. (2013), 'A Model of Type Theory in Cubical Sets'

There is a large-scale implementation of CubicalTT in Agda called CubicalAgda [47], amongst other efforts.

[47]: Vezzosi et al. (2019), 'Cubical agda: a dependently typed programming language with univalence and higher inductive types'

2-level Type Theories (2TTs). UIP and univalence are inconsistent together for a simple reason: univalence allows bool = bool to have two *distinct* proofs: one given by the identity equivalence, the other by the equivalence that swaps true and false, while UIP states there must be only one such proof. This in particular means that univalence is inconsistent in ETT. There is a solution to this which consists in assuming two equalities. This cannot be done naively however, since two equalities that cohabit collapse. For this, the two equalities must be compartmentalised to limit their interaction. Hence the name 2-level Type Theory (2TT). This idea was first introduced as Homotopy Type System (HTS) [48] in an extensional setting, but since then there have been other approaches to 2TT [49, 50] where only UIP is assumed.

[48]: Voevodsky (2013), *A simple type system with two identity types*

[49]: Altenkirch et al. (2016), 'Extending Homotopy Type Theory with Strict Equality'

[50]: Annenkov et al. (2017), 'Two-Level Type Theory and Applications'

Quotients

Implementing quotients is a notoriously problematic issue in type theory. In Coq this is still limited to specific cases where the quotient can effectively be expressed. OTT and STT should in principle help in solving that problem since they involve the idea that a type comes with its equality. The problem is usually to make everything compute and behave nicely.

The world of HoTT has brought several notions like that of Quotient Inductive Type (QIT) [51]. A Quotient Inductive Type (QIT) not only has term constructors, but also equality constructors: defining the type and quotienting it at the same time.

[51]: Altenkirch et al. (2016), 'Type theory in type theory using quotient inductive types'

7 Desirable properties of type theories

To compare different type theories there are several measures we can use in the form of usual or desirable properties that they might satisfy or not. After a brief presentation of the main ones I will summarise which of the theories of Chapter 6 (Flavours of type theory) has which properties in a table.

7.1 Properties

Weakening and substitutivity

Variables and binders are essential to type theory and as such we have to treat them with care, in particular we want our theories to be *compositional*, meaning that different blocks that make sense can be assembled into something that still makes sense. This is—in part—embodied in the two following properties.

Definition 7.1.1 (Weakening) *A type theory enjoys weakening when for any $\Gamma, \Xi \vdash t : A$ and $\vdash \Gamma, \Delta$ we have $\Gamma, \Delta, \Xi \vdash t : A$.*

Note that in more generality you might have to rename variables when weakening, so for this we usually introduce a *lifting* operator $\uparrow$ such that we have $\Gamma, \Delta, \Xi \vdash \uparrow_k^n t : \uparrow_k^n A$ where $n = |\Delta|$ and $k = |\Xi|$, that is the lengths of the contexts.

Weakening means that you can plug a term into a larger context. The version of weakening I present here is one amongst many and is sometimes called different names like 'thinning', but I will stick to 'weakening' in the remainder of this document.

Substitutions are the way to instantiate the variables that are bound. This happens for instance after a β-reduction. Whereas we substituted one term for one variable for the β-reduction, we can consider substitutions that map several variables to as many terms. Substitutions are typed using two contexts: $\sigma : \Gamma \to \Delta$ basically states that the substitution σ maps variables of Δ to terms typed in Γ.

Reminder: β-reduction

$$(\lambda(x : A).t)\ u \rightarrow_\beta t[x \leftarrow u]$$

$$\frac{\forall (x : A) \in \Delta,\ \Gamma \vdash \sigma(x) : A\sigma}{\sigma : \Gamma \to \Delta}$$

This is sometimes written $\Gamma \vdash \sigma : \Delta$ instead, and typing definitions vary a little depending on the definition of substitution but this is the basic idea.

Definition 7.1.2 (Substitutivity) *A type theory is substitutive when for any $\Delta \vdash t : A$ and any substitution $\sigma : \Gamma \to \Delta$, we have $\Gamma \vdash t\sigma : A\sigma$.*

More often than not, weakening and substitutivity also hold for reduction and conversion.

Inversion of typing

Inversion of typing cannot really be used as a means of comparing theories as it is always present in some form or the other. It is nonetheless a very useful property to state when reasoning on a type theory. It is saying than when we have $\Gamma \vdash t : A$, by analysing the shape of t we can get information on A (and sometimes even Γ).

For instance, if we have $\Gamma \vdash t\ u : T$, then by inversion of typing we know that there must exist A and B such that

$$\Gamma \vdash \Pi(x:A).\ B : s \qquad \Gamma \vdash t : \Pi(x:A).\ B \qquad \Gamma \vdash u : A$$

$$\Gamma \vdash T \equiv B[x \leftarrow u]$$

Reminder: Application rule

$$\frac{\begin{array}{c}\Gamma \vdash \Pi(x:A).\ B : s \\ \Gamma \vdash t : \Pi(x:A).\ B \qquad \Gamma \vdash u : A\end{array}}{\Gamma \vdash t\ u : B[x \leftarrow u]}$$

This can be proved by seeing that only two typing rules can be concluded with $\Gamma \vdash t\ u : T$: the application rule and the conversion rule. The result follows from a simple induction.

Now, it will not always be stated this way depending on the premises of the application rule, but also depending on the presence or not of a conversion rule. In the case of WTT for instance, there is no conversion so instead of $\Gamma \vdash T \equiv B[x \leftarrow u]$ we will have syntactic equality $T =_\alpha B[x \leftarrow u]$.

Sometimes it will be talking about cumulativity $\preceq$ instead of conversion $\equiv$: in that case we would have $\Gamma \vdash B[x \leftarrow u] \preceq T$ (we have to apply the application rule first and then possibly several time the cumulativity rule).

One usually proves inversion of typing for every term constructor, but I will not do it here.

Validity

The term *validity* might be a bit overloaded, and maybe not the norm when it comes to type theory, but I will use it to be consistent with the notion for Coq. This property states that the type on the right-hand side of the colon is indeed a type.

Definition 7.1.3 (Validity) *A type theory enjoys validity when from $\Gamma \vdash t : A$ one can deduce $\Gamma \vdash A$ (i.e. $\Gamma \vdash A : s$ for some sort s).*

Depending on the theory, we can often prove a similar property regarding contexts: namely that $\Gamma \vdash t : A$ implies that Γ is well-formed ($\vdash \Gamma$). Having this property mainly depends on whether the typing rules of things like sorts and variables ask for the context to be well-formed.

$$\frac{(x:A) \in \Gamma}{\Gamma \vdash x : A} \qquad \text{vs} \qquad \frac{\vdash \Gamma \qquad (x:A) \in \Gamma}{\Gamma \vdash x : A}$$

In a theory which does not have this requirement / property, many lemmata will only apply assuming the contexts involved are well-formed. The difference between presentations like these will be studied in Chapter 9 (Syntax and formalisation of type theory).

Unique / principal typing

Unique typing is a property saying that each term (given a context) has only one type up to conversion.

Definition 7.1.4 (Unique typing) *A type theory enjoys unique typing when $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ imply $\Gamma \vdash A \equiv B$.*

This usually means that the term is carrying enough information to recover its type. If one were to use Curry-style terms like $\lambda x.\ x$ then this property could not hold: indeed $\vdash \lambda x.\ x : \mathsf{unit} \longrightarrow \mathsf{unit}$ and $\vdash \lambda x.\ x : \mathsf{bool} \longrightarrow \mathsf{bool}$ both hold and the two arrow types are not related.

We will talk more in depth about Curry-style terms in Chapter 9 (Syntax and formalisation of type theory).

This property can be broken for other reasons however: *e.g.* the presence of subtyping (typically cumulativity). In such a case, unique typing can be relaxed to principal typing.

Definition 7.1.5 ((Weak) principal typing) *A type theory enjoys principal typing (in the weaker sense) when $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ imply $\Gamma \vdash t : C$ with $\Gamma \vdash C \preceq A$ and $\Gamma \vdash C \preceq B$.*

It might be perhaps more often understood in a stronger sense.

Definition 7.1.6 ((Strong) principal typing) *A type theory enjoys principal typing (in the stronger sense) when for every well typed term t in Γ, there exists a type P such that $\Gamma \vdash t : P$ and that whenever $\Gamma \vdash t : A$ we have $\Gamma \vdash P \preceq A$. P is called a principal type of t (in Γ).*

The idea is that the principal type is the most general type that can be given to a term.

Properties of reduction

When the type theory features reduction, we also want it to satisfy some requirements, aside from the fact that reduction behaves in a compositional way as I already explained in Subsection 'Weakening and substitutivity'.

First are basic properties of rewriting systems[1]: confluence and termination.

1: I will not go into details about what they are and it is not necessary to know about them to understand any of this.

Definition:

The transitive reflexive closure of reduction, written $\rightarrowtriangle^\star$, is defined (inductively) by the rules

$$\frac{}{t \rightarrowtriangle^\star t} \qquad \frac{t \rightarrowtriangle u \qquad u \rightarrowtriangle^\star v}{t \rightarrowtriangle^\star v}$$

Definition 7.1.7 (Confluence) *The reduction relation $\rightarrowtriangle$ is confluent if for every t such that $t \rightarrowtriangle^\star u$ and $t \rightarrowtriangle^\star v$ there exists w such that $u \rightarrowtriangle^\star w$ and $v \rightarrowtriangle^\star w$.*

Confluence is often summarised with the following diagram.

$$\begin{tikzcd}
& t \arrow[ld, "\star"'] \arrow[rd, "\star"] & \\
u \arrow[rd, dashed, "\star"'] & & v \arrow[ld, dashed, "\star"] \\
& \exists w &
\end{tikzcd}$$

Definition 7.1.8 (Weak normalisation) *Weak normalisation of reduction means that for every well-typed term t there exists a term n such that $t \rightarrow\!\!\triangleright^{\star} n \not\rightarrow\!\!\triangleright$. In such a case, n is called a* normal form *of t.*

A normal form is a term that cannot reduce any further as illustrated by the $n \not\rightarrow\!\!\triangleright$ notation.

Definition 7.1.9 (Strong normalisation) *Strong normalisation, or termination, of reduction means that for every well-typed term t, there is no infinite reduction sequence starting from t.*

When a system is confluent and terminating, there exists exactly one normal form for each well-typed term, we can thus talk about *the* normal form of a term.

These properties are related to typing in that they require the term to be well-typed in order to be applicable. That is because it is too easy to break termination with nonsensical terms. There is however a reduction property that is much more linked to typing: subject reduction, sometimes called type safety or type preservation.

Definition 7.1.10 (Subject reduction) *A type theory enjoys subject reduction when for every $\Gamma \vdash t : A$ and $t \rightarrow\!\!\triangleright u$, we also have $\Gamma \vdash u : A$.*

This means that evaluating a program of type, *e.g.*, nat, will return a value of type nat. In terms of proofs, this means that simplifying a proof of proposition P will still yield a proof of P. This is a very sensible property to have in a programming language or a proof assistant. Its absence is usually a sign that something is very wrong with the considered system: if evaluation might break things, why have evaluation at all?

Injectivity of Π-types

Another property slightly related to confluence and subject reduction is the injectivity of Π-types. It is part of the more general notion of injectivity of constructors and non-confusion of type constructors.

For Π-types, injectivity means that whenever

$$\Pi(x : A).B \equiv \Pi(x : A').B'$$

then we have both $A \equiv A'$ and $B \equiv B'$.

This property is usually proven using confluence of reduction so that $\Pi(x : A).B \equiv \Pi(x : A').B'$ is equivalent to both types reducing to the same term. Since the Π type constructor will never disappear from reduction, it necessarily means that A and A' reduce to the same term, hence are convertible (and the same holds for the codomains). The proof is not always so easy, and it very much depends on the definition used for conversion.

Injectivity of Π-types is crucial in proving that β-reduction is type-preserving, one of the key elements to the proof of subject reduction.

Injectivity of constructors states that for instance $\mathbf{S}\ n \equiv \mathbf{S}\ m$ implies $n \equiv m$ while non-confusion says that $0 \not\equiv \mathbf{S}\ n$ or $\mathsf{Type} \not\equiv A \to B$. Those are useful to achieve consistency which we will see later.

Canonicity

Canonicity is a way of saying that, for instance, booleans *really* are booleans *i.e.* they are either true or false. It is an important property, especially when seen through the constructivism prism. When expressed for Σ-types, this corresponds to the so-called *witness property*: a proof of $\Sigma(x : A).\ P\ x$ (in the empty context, so without assumption) is necessarily of the form $(t; p)$ with $t : A$ and $p : P\ t$.

Definition 7.1.11 (Canonicity) *A type theory enjoys canonicity when $\vdash t : \mathsf{bool}$ implies that either $t \equiv \mathsf{true}$ or $t \equiv \mathsf{false}$. More generally, every term of an inductive type is convertible to a construction of this type.*

Note that t is typed in the empty context.

This doesn't always hold. The presence of axioms usually hinders canonicity. Take for instance the LEM, stated using the sum types of Chapter 5:

$$\mathsf{lem} : \Pi\ (P : \mathsf{Type}).\ P + (P \to \bot)$$

$P \to \bot$ is the usual way to represent the negation of P: $\neg P$.

We can use this axiom to build a boolean:

$$\begin{array}{lll} \multicolumn{3}{l}{\mathsf{match\ lem\ bool\ return}\ x.\ \mathsf{bool\ with}} \\ \mid \mathsf{inl}\ b & \Longrightarrow & b \\ \mid \mathsf{inr}\ n & \Longrightarrow & \mathsf{true} \\ \mathsf{end} & & \end{array}$$

but it will never be convertible to either true or false because $\mathsf{lem\ bool}$ does not reduce to a constructor (in fact it does not reduce at all). There is no way of making this axiom into a definition that would compute, hinting at the fact that canonicity and constructivism are closely related.

Canonicity usually holds in a much stronger and useful sense in type theory:

Definition 7.1.12 (Computational canonicity) *Computational canonicity corresponds to the fact that whenever $\vdash t : \mathsf{bool}$, then either $t \rightarrow^\star \mathsf{true}$ or $t \rightarrow^\star \mathsf{false}$. More generally, every term of an inductive type reduces to a construction of this type.*

So computational canonicity ensures that whenever you prove that there exists some t such that $P\ t$, it is possible to evaluate the proof to extract the t in question.

Decidability of type-checking

A property is called *decidable* when there exists an algorithm that returns true whenever it holds, and false otherwise. Decidability of type-checking thus means the existence of a sound and complete type-checker:

Definition 7.1.13 (Decidability of type-checking) *Type checking is decidable for a type theory if there exists an algorithm such that, when given a context Γ, a term t and a type A, returns* true *when there is a derivation of $\Gamma \vdash t : A$ and* false *when there is not.*

There are variants of this of course, sometimes it will only work when we already know that $\vdash \Gamma$ and $\Gamma \vdash A$.

Sometimes we have a stronger property, as is the case for Coq, that is decidability of inference: there is an algorithm taking Γ and t and which returns some A such that $\Gamma \vdash t : A$ or an error stating that t is ill-typed.

Decidability of conversion is crucial in getting decidability of type checking, and can sometimes derive from confluence and strong normalisation.

Decidability of type-checking is an important property because it crystallises the fact that a term is indeed a *proof*, it is a good enough certificate that people can check independently. However, in a system that does not feature decidable type-checking, you can still provide certificates: for one you could simply store the full derivation as a proof; it is however possible to translate the proof to another system with decidable checking, like I will present in Part 'Elimination of Reflection'.

Consistency

To be able to use a type theory as a logic, it needs to be consistent. Consistency means that not all types are provable, or equivalently that the empty type is uninhabited.

I already discussed consistency in Chapter 2 (Proof theory).
See Chapter 5 (Usual definitions in type theory) for definition(s) of the empty type $\bot$.

Definition 7.1.14 (Consistency) *A type theory is said to be consistent when there is no term t such that $\vdash t : \bot$.*

This does not prevent you from having proofs of $\bot$ in inconsistent contexts.

Having canonicity in the theory is a good way to deduce consistency. Indeed, if there is a proof of $\bot$ in the empty context, there must be one which is a constructor, since there are no constructors for $\bot$ it is not possible.

It is worth noting that inconsistent type theories can still be of interest. For instance if you are interested in having effects in your type theory, though you might want to be able to know which terms might use effects leading to inconsistencies and use only *pure* terms to prove things [52, 53]. This allows for proofs on impure programs.

[52]: Pédrot et al. (2019), 'A Reasonably Exceptional Type Theory'
[53]: Pédrot et al. (2020), 'The Fire Triangle'

7.2 Summarising table

I will now try to summarise which theory enjoys which property in a table. Weakening, substitution and validity always hold in theories that I consider and inversion of typing always holds in some way or another so I will not put them in the table.

I will separate the theories in four groups sharing properties.

(A) Coq / Predicative Calculus of Cumulative Inductive Constructions
(B) Agda / Martin-Löf Type Theory, Cubical Type Theory
(C) Extensional Type Theory
(D) Weak Type Theory

	(A)	(B)	(C)	(D)
Unique (U) / Principal type (P)	P	U	U	U
Confluence	yes	yes	–	–
Normalisation	yes	yes	–	–
Subject reduction	yes	yes	–	–
Injectivity of Π-types	yes	yes	no	no
Computational / regular canonicity	comp	comp	reg	no
Decidability of type-checking	yes	yes	no	yes
Consistency	yes	yes	yes	yes

I write – for properties that do not apply like properties on reduction for any theory without a notion of reduction.

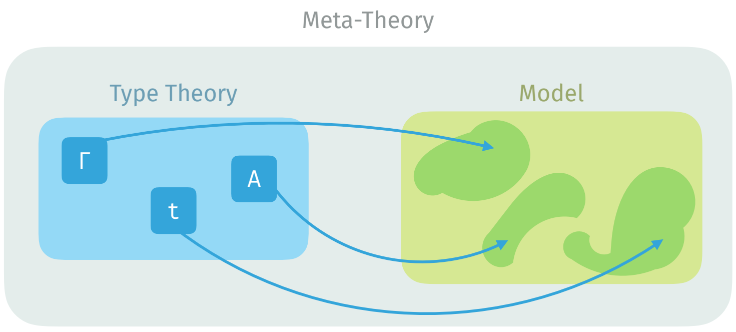
2-level variants of the theory should enjoy the same properties as their 1-level counterpart: HTS goes in (C) while 2TT goes in (B). HoTT can also be considered a variant of a type theory, and should be placed in either (A) or (B) depending on whether it features cumulativity or not. Likewise, ITT could go in either (A) or (B).

Models of type theory | 8

Justifying a logic is often achieved using *models*. A model consists in giving an interpretation to all constructs of the logic we want to study, such that its rules are still verified. There are several ways to get models of type theory, I will present some of the most common ones in this chapter, though my means of choice will be presented in depth in Chapter 10 (Translations).

8.1 What is a model?

A model of type theory is an interpretation of the concepts of a type theory into another theory or object, both living in the same meta-theory. To be more precise, a model is given by a class of objects



to interpret contexts, one for terms and one for types; but a model also provides an interpretation to judgments in such a way that the interpretation is coherent.

What can be proved using models

Consistency. I already briefly mentioned this but the main point of models is to prove consistency of a theory, relying on the already *known* consistency of the theory in which the model lives… At least in principle. It is however very rare[1] to *know* that a theory is consistent, instead we should see it as a theory we *trust*, typically a theory that is widely accepted as being consistent by the community of mathematicians. In this case we talk about *relative consistency*: the theory is consistent, relative to the consistency of the theory of its model. This also applies to the meta-theory in which we show that the interpretation is correct. As such, it is best to keep it as simple as possible to avoid relying on the consistency of too complicated objects.

1: This may work with very simple theories that cannot express arithmetic, as explained in Section 2.3 (Limitations: Gödel's incompleteness theorems) of Chapter 2

Independence. Another interesting application of models is showing *independence* of a proposition.

Definition 8.1.1 (Independent proposition) *A proposition P is said to be independent from a theory $\mathcal{T}$ when neither P nor $\neg P$ can be proven within $\mathcal{T}$.*

A way to prove that some P is independent from $\mathcal{T}$ is to give a model of $\mathcal{T}$ which validates P and another model which invalidates it (or validates $\neg P$). Indeed if any one of P or $\neg P$ could be proven in $\mathcal{T}$, then it would be valid in both models, leading to at least one of them being inconsistent.

The fact that a proposition can neither be validated nor invalidated in a theory can come as surprising for some, especially in a classical mindset. Gödel's incompleteness theorems have to do with this.

Reminder: Classical logic

Classical logic is often characterised by the presence of the LEM which consists of a proof of $A \vee \neg A$.

Models can also be useful to prove other properties like normalisation and even subject reduction.

8.2 Set-theoretic models

Set-theoretic models are models of type-theory where types are interpreted as sets. For instance the (non-dependent) type $A \to B$ will be interpreted as the set of set-theoretic functions from the interpretation of A to the interpretation of B:

$$[\![A \to B]\!] := \mathcal{F}([\![A]\!], [\![B]\!])$$

It is usual to write $[\![A]\!]$ for the interpretation of A.

Instead of describing all set-theoretic models abstractly I will rather focus on one example: the proof-irrelevant model of CIC. I will actually deal with the smaller case of CoC. This is exposed in [54]—following [55, 56]—in a very clear—and French—way, but without accounting for contexts and open terms. I will instead base my own exposition on that of [57], itself based upon the folklore model seen in [58–60]. The model is called *proof-irrelevant* because proofs of propositions (proofs of P : Prop) all collapse in the model: provable propositions are interpreted as the singleton $\{\emptyset\}$.

[54]: Miquel (2001), 'Le Calcul des Constructions implicite: syntaxe et sémantique'
[55]: Aczel (1998), 'On relating type theories and set theories'
[56]: Xi (2000), 'Imperative programming with dependent types'
[57]: Werner (1997), 'Sets in types, types in sets'

[58]: Coquand (1989), 'Metamathematical investigations of a calculus of constructions'
[59]: Dybjer (), 'Inductive sets and families in Martin-L of type theory and their set-theoretic semantics'
[60]: Howe (1991), 'On computational open-endedness in Martin-Lof's type theory'

First, I will remind what rules we are dealing with. This theory will only have two universes: Prop and Type. The model can be adapted to deal with more universes but it is a bit more complicated.

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

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

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash \Pi(x : A).B : s}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A).B}$$

$$\frac{\Gamma \vdash u : \Pi(x : A).B \qquad \Gamma \vdash v : A}{\Gamma \vdash u\ v : B[x \leftarrow v]} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \equiv B}{\Gamma \vdash t : B}$$

We have two rules for Π-types because we have an impredicative universe of propositions Prop. The rules where Prop is on the left are unnecessary because we have a weak cumulativity Prop $\leq$ Type.

s stands for any sort, *i.e.* Prop or Type.

Conversion $A \equiv B$ is simply the congruent closure of β-reduction.

As I already said, the idea in this proof-irrelevant model is to interpret propositions as either the empty set ($\emptyset$) for uninhabited propositions, or as the singleton $\{\emptyset\}$ for inhabited ones. These two sets are the ordinals 0 and 1 and I will write them as such. The interpretation of Prop is then $2 := \{0, 1\}$.

As you can see, in the typing rules we have to deal with terms in a context Γ, *i.e.* open terms. The interpretation of such a term is dependent on the context, this can be easily seen when considering $x : A \vdash x$ and $x : B \vdash x$, the two x do not necessarily mean the same thing. As such we will interpret terms together with their context: $[\![\Gamma \vdash t]\!]$, which will be set-theoretic functions of domain $[\![\Gamma]\!]$, without a precise codomain.

$[\![\Gamma]\!]$ is defined as follows:

$$\begin{array}{rcl} [\![\bullet]\!] & := & 1 \\ [\![\Gamma, x : A]\!] & := & \{(\gamma, a) \mid \gamma \in [\![\Gamma]\!] \wedge a \in [\![\Gamma \vdash A]\!](\gamma)\} \end{array}$$

mutually with $[\![\Gamma \vdash t]\!]$:

$$\begin{array}{rcl} [\![\Gamma \vdash \Pi(x : A).B]\!](\gamma) & := & \prod_{a \in [\![\Gamma \vdash A]\!](\gamma)} [\![\Gamma, x : A \vdash B]\!](\gamma, a) \\ [\![\Gamma \vdash \mathsf{Prop}]\!](\gamma) & := & 2 \\ [\![\Gamma \vdash \mathsf{Type}]\!](\gamma) & := & \mathcal{U} \\ [\![\Gamma \vdash \lambda(x : A).t]\!](\gamma) & := & a \in [\![\Gamma \vdash A]\!](\gamma) \mapsto [\![\Gamma, x : A \vdash t]\!](\gamma, a) \\ [\![\Gamma \vdash u\ v]\!](\gamma) & := & [\![\Gamma \vdash u]\!](\gamma)([\![\Gamma \vdash v]\!](\gamma)) \\ [\![\Gamma \vdash x_i]\!](\gamma) & := & \pi_2 \circ \pi_1^i(\gamma) \end{array}$$

Definition: Dependent product of sets

$\prod_{x \in A} B(x)$ is defined as the set of functions f from A to $\bigcup_{x \in A} B(x)$ such that $f(x) \in B(x)$.

where in the last equality

$$\Gamma = x_n : A_n, \ldots x_i : A_i, \ldots x_0 : A_0$$

One cannot distinguish between propositions and usual types in the current setting so in [55], Aczel introduces a non-standard encoding of set-theoretic functions such that the constant function $x \mapsto 0$ is *equal* to 0. In [61] they build a set-theoretic model for the whole of PCUIC using the same trick.

[55]: Aczel (1998), 'On relating type theories and set theories'

[61]: Timany et al. (2017), *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)*

I did not precise what $\mathcal{U}$ was. From $\mathcal{U}$ we only require that it is closed under products and power-sets and contains at least 2. It needs only be the cardinal $\aleph_0$ in our case, but if we wanted more universes—say a hierarchy of them—we could use inaccessible cardinals. We discuss them briefly in Section 8.3.

Consistency. Before we can claim it is a model, we need to show that whenever $\Gamma \vdash t : A$ then $[\![\Gamma \vdash t]\!] \in [\![\Gamma \vdash A]\!]$. This will in particular prove that every proof of a proposition is indeed sent to 0. It also allows us to conclude consistency since the empty type

This property is called *soundness*.

$$\bot \coloneqq \Pi(P : \mathsf{Prop}).P$$

is interpreted (in the empty context) as

$$[\![\vdash \bot]\!](0) = \prod_{X \in 2} X$$

i.e. the set of functions f such that $f(0) \in 0$ and $f(1) \in 1$, since there is no element in 0 there is no such f and as such

$$[\![\vdash \bot]\!] = 0$$

which is the empty set.

The proof of soundness is done by looking at each rule and showing they preserve the property: for instance if we take

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash \Pi(x : A).B : \mathsf{Type}}{\Gamma \vdash \lambda(x : A).t : \Pi(x : A).B}$$

we assume we have $[\![\Gamma, x : A \vdash t]\!](\delta) \in [\![\Gamma, x : A \vdash B]\!](\delta)$ and $[\![\Gamma \vdash \Pi(x : A).B]\!](\gamma) \in [\![\Gamma \vdash \mathsf{Type}]\!](\gamma)$ for every $\delta \in [\![\Gamma, x : A]\!]$ and $\gamma \in [\![\Gamma]\!]$ and show that

$$[\![\Gamma \vdash \lambda(x : A).t]\!](\gamma) \in [\![\Gamma \vdash \Pi(x : A).B]\!](\gamma)$$

If we reformulate, we want to show that for all $\gamma \in \Gamma$ and $a \in [\![\Gamma \vdash A]\!](\gamma)$ we have

$$[\![\Gamma, x : A \vdash t]\!](\gamma, a) \in [\![\Gamma, x : A \vdash B]\!](\gamma, a)$$

which corresponds to our hypothesis.

For some other cases like the conversion and application rules we need to show that convertible terms are sent to equal sets/functions and that the interpretation behaves well with respect to substitution. I will not dwell on the details as it has been dealt with nicely in the references I put at the beginning of this section.

Set-theoretic models are not the only way to make models of type theory of course. For instance they may not be best-suited to interpretation of computational content since β-convertible terms are interpreted as the same set. They provide however a nice way of seeing the relation between set-theory—the widely accepted framework in which to do mathematics—and type theory which we[2] advocate instead.

2: I will leave that one as imprecise as it is.

8.3 Categorical models

Categorical models are one of the most used way of defining models for type theories. There are several notions of categorical models: contextual categories [62] from which stem categories with attributes [63] and C-systems [64], Categories with Families (CwFs) [65], and more… I will focus on CwFs in this document.

[62]: Cartmell (1986), 'Generalised algebraic theories and contextual categories'
[63]: Hofmann (1994), 'On the interpretation of type theory in locally cartesian closed categories'
[64]: Voevodsky (2016), 'Subsystems and regular quotients of C-systems'
[65]: Dybjer (1995), 'Internal type theory'

Categories with families

A CwF is given by

1. a collection of objects (or contexts) Con;
2. morphisms (or substitutions) $\sigma : \Gamma \to \Delta$ between contexts $\Gamma, \Delta : \mathsf{Con}$;
3. for each context $\Gamma : \mathsf{Con}$, a collection $\mathsf{Ty}\ \Gamma$ of types in that context;
4. for each context $\Gamma : \mathsf{Con}$ and type $A : \mathsf{Ty}\ \Gamma$, a collection $\mathsf{Tm}\ \Gamma\ A$ of terms of type A;
5. an operation of substitution for types taking $\sigma : \Gamma \to \Delta$ and a type $A : \mathsf{Ty}\ \Delta$ to type $A[\sigma] : \mathsf{Ty}\ \Gamma$;
6. a similar substitution operation on terms taking $\sigma : \Gamma \to \Delta$ and a term $t : \mathsf{Tm}\ \Delta\ A$ to term $t[\sigma] : \mathsf{Tm}\ \Gamma\ A[\sigma]$;
7. a terminal object representing the empty context $\bullet : \mathsf{Con}$;
8. an extension operation taking $\Gamma : \mathsf{Con}$ and $A : \mathsf{Ty}\ \Gamma$ to a new object $\Gamma, A : \mathsf{Con}$ together with a morphism $\mathsf{p} : \Gamma, A \to \Gamma$ and a term $\mathsf{q} : \mathsf{Tm}\ (\Gamma, A)\ A[\mathsf{p}]$;
9. an operation taking $\sigma : \Gamma \to \Delta$ and term $a : \mathsf{Tm}\ \Gamma\ A[\sigma]$ to substitution $\sigma, a : \Gamma \to \Delta, A$ such that $\mathsf{p} \circ (\sigma, a) = \sigma$ and $\mathsf{q}[\sigma, a] = a$.

Here I use type-theoretic notations: $\Gamma : \mathsf{Con}$ means Γ an object of the catergory Con, $\sigma : \Gamma \to \Delta$ means σ a morphism from object Γ to object Δ, etc.

The two first have to give the structure of category, this in particular means that there is an identity substitution, and associative composition of those.

The category Set of sets is a CwF [66]:

[66]: Hofmann (1997), 'Syntax and semantics of dependent types'

1. objects are sets;
2. substitutions are given as set-theoretic functions, composed as functions;
3. types in $\mathsf{Ty}\ \Gamma$ are sets indexed by Γ;
4. terms of $\mathsf{Tm}\ \Gamma\ A$ are choice functions $t \in \mathcal{F}(\Gamma, \bigcup_{\gamma \in \Gamma} A_\gamma)$ such that $t(\gamma) \in A_\gamma$ for every $\gamma \in \Gamma$;
5. given $\sigma : \Gamma \to \Delta$ and $A : \mathsf{Ty}\ \Delta$, the type $A[\sigma]$ is defined as the Γ-indexed family $(A_{\sigma(\gamma)})_\gamma$;
6. given $\sigma : \Gamma \to \Delta$ and $t : \mathsf{Tm}\ \Delta\ A$, the term $t[\sigma]$ is defined as the choice function $t \circ \sigma$;
7. the empty context is given as the singleton $\{\emptyset\}$;
8. given $\Gamma : \mathsf{Con}$ and $A : \mathsf{Ty}\ \Gamma$, the extended context Γ, A is defined as the disjoint union $\coprod_{\gamma \in \Gamma} A_\gamma$, the morphism p and term q are defined as the first and second projection respectively;
9. given $\sigma : \Gamma \to \Delta$ and $a : \mathsf{Tm}\ \Gamma\ A[\sigma]$, the substitution σ, a is defined as the function $\gamma \mapsto (\sigma(\gamma), a(\gamma))$ which verifies the conditions.

6 holds because for every $\gamma \in \Gamma$,

$$(t \circ \sigma)(\gamma) = t(\sigma(\gamma)) \in A_{\sigma(\gamma)} = A[\sigma]_\gamma$$

Right now this only interprets a type theory with no type or term constructors. I will give a few example of those.

Dependent products. A CwF is said to have dependent products when it has the following elements:

- for any $A : \mathsf{Ty}\ \Gamma$ and $B : \mathsf{Ty}\ (\Gamma, A)$ there exists $\Pi\ A\ B : \mathsf{Ty}\ \Gamma$;
- for $b : \mathsf{Tm}\ (\Gamma, A)\ B$ there exists $\lambda b : \mathsf{Tm}\ \Gamma\ (\Pi\ A\ B)$;
- for $f : \mathsf{Tm}\ \Gamma\ (\Pi\ A\ B)$ and $u : \mathsf{Tm}\ \Gamma\ A$ there exists $\mathsf{app}(f, u) : \mathsf{Tm}\ \Gamma\ B[\mathsf{id}, u]$

such that for any $\sigma : \Delta \longrightarrow \Gamma$ and terms and types as above the following equations hold:

$$\begin{aligned}
(\Pi\ A\ B)[\sigma] &= \Pi\ A[\sigma]\ B[(\sigma \circ \mathsf{p}), \mathsf{q}] \\
(\lambda b)[\sigma] &= \lambda b[(\sigma \circ \mathsf{p}), \mathsf{q}] \\
(\mathsf{app}(f, u))[\sigma] &= \mathsf{app}(f[\sigma], u[\sigma]) \\
\mathsf{app}(\lambda b, u) &= b[\mathsf{id}, u] \\
\lambda \mathsf{app}(f[\mathsf{p}], \mathsf{q}) &= f
\end{aligned}$$

The first four equations are simply there to attest to the good behaviour of substitution. The last two are more interesting, corresponding to β-reduction and η-contraction respectively.

Reminder: η-expansion

η-expansion is defined as follows

$$t \rightharpoonup_\eta \lambda x.\, t\ x$$

η-contraction is the opposite relation. See Chapter 6.

Dependent sums. A CwF has (negative) dependent sums when:

- for any $A : \mathsf{Ty}\ \Gamma$ and $B : \mathsf{Ty}\ (\Gamma, A)$ there exists $\Sigma\ A\ B : \mathsf{Ty}\ \Gamma$;
- for $a : \mathsf{Tm}\ \Gamma\ A$ and $b : \mathsf{Tm}\ \Gamma\ B[\mathsf{id}, a]$ there exists the dependent pair $\langle a, b\rangle : \mathsf{Tm}\ \Gamma\ (\Sigma\ A\ B)$;
- for $p : \mathsf{Tm}\ \Gamma\ (\Sigma\ A\ B)$ there exists $p.1 : \mathsf{Tm}\ \Gamma\ A$;
- for $p : \mathsf{Tm}\ \Gamma\ (\Sigma\ A\ B)$ there exists $p.2 : \mathsf{Tm}\ \Gamma\ B[\mathsf{id}, p.1]$

See Chapter 6 (Flavours of type theory) for a discussion on negative vs positive sums.

such that

$$\begin{aligned}
(\Sigma\ A\ B)[\sigma] &= \Sigma\ A[\sigma]\ B[(\sigma \circ \mathsf{p}), \mathsf{q}] \\
\langle a, b\rangle[\sigma] &= \langle a[\sigma], b[\sigma]\rangle \\
(p.1)[\sigma] &= p[\sigma].1 \\
(p.2)[\sigma] &= p[\sigma].2 \\
\langle a, b\rangle.1 &= a \\
\langle a, b\rangle.2 &= b
\end{aligned}$$

Identity types. A CwF has identity types—another name for the equality type—when:

- for $A : \mathsf{Ty}\ \Gamma$ and $u, v : \mathsf{Tm}\ \Gamma\ A$ there exists $\mathsf{Id}_A\ u\ v : \mathsf{Ty}\ \Gamma$;
- for $A : \mathsf{Ty}\ \Gamma$ and $u : \mathsf{Tm}\ \Gamma\ A$ there exists $\mathsf{refl}_u : \mathsf{Tm}\ \Gamma\ (\mathsf{Id}_A\ u\ u)$;
- for $A : \mathsf{Ty}\ \Gamma$ and $u : \mathsf{Tm}\ \Gamma\ A$ and $C : \mathsf{Tm}\ (\Gamma, A, \mathsf{Id}_{A[\mathsf{p}]}\ u[\mathsf{p}]\ \mathsf{q})$ and $f : \mathsf{Tm}\ \Gamma\ \mathsf{C}[\mathsf{id}, u, \mathsf{refl}_u]$ and $v : \mathsf{Tm}\ \Gamma\ A$ and $e : \mathsf{Tm}\ \Gamma\ (\mathsf{Id}_A\ u\ v)$ there exists $\mathsf{J}_A\ u\ C\ f\ v\ e : \mathsf{Tm}\ \Gamma\ \mathsf{C}[\mathsf{id}, v, e]$

such that

$$\begin{aligned}
(\mathsf{Id}_A\ u\ v)[\sigma] &= \mathsf{Id}_{A[\sigma]}\ u[\sigma]\ v[\sigma] \\
(\mathsf{refl}_a)[\sigma] &= \mathsf{refl}_{a[\sigma]} \\
(\mathsf{J}_A\ u\ C\ f\ v\ e)[\sigma] &= \mathsf{J}_{A[\sigma]}\ u[\sigma]\ \mathsf{C}[(\sigma \circ \mathsf{p}, \mathsf{q}) \circ \mathsf{p}, \mathsf{q}]\ f[\sigma]\ v[\sigma]\ e[\sigma]
\end{aligned}$$

Booleans. A CwF has booleans when for any $\Gamma : \mathsf{Con}$ it has

- $\mathsf{bool} : \mathsf{Ty}\ \Gamma$;
- $\mathsf{true} : \mathsf{Tm}\ \Gamma\ \mathsf{bool}$;
- $\mathsf{false} : \mathsf{Tm}\ \Gamma\ \mathsf{bool}$;
- given $b : \mathsf{Tm}\ \Gamma\ \mathsf{bool}$ and $C : \mathsf{Ty}\ (\Gamma, \mathsf{bool})$ and $t : \mathsf{Tm}\ \Gamma\ C[\mathsf{id}, \mathsf{true}]$ and $f : \mathsf{Tm}\ \Gamma\ C[\mathsf{id}, \mathsf{false}]$, some $\mathsf{if}\ b\ \mathsf{return}\ C\ \mathsf{then}\ t\ \mathsf{else}\ f : \mathsf{Tm}\ \Gamma\ C[\mathsf{id}, b]$

such that

$$\begin{array}{rcl}
\mathsf{bool}[\sigma] & = & \mathsf{bool} \\
\mathsf{true}[\sigma] & = & \mathsf{true} \\
\mathsf{false}[\sigma] & = & \mathsf{false} \\
\begin{pmatrix} \mathsf{if}\ b & \\ \mathsf{return}\ C & \\ \mathsf{then} & t \\ \mathsf{else} & f \end{pmatrix}[\sigma] & = & \begin{array}{ll} \mathsf{if}\ b[\sigma] & \\ \mathsf{return}\ C[(\sigma \circ \mathsf{p}), \mathsf{q}] & \\ \mathsf{then} & t[\sigma] \\ \mathsf{else} & f[\sigma] \end{array} \\
\mathsf{if\ true\ return}\ C\ \mathsf{then}\ t\ \mathsf{else}\ f & = & t \\
\mathsf{if\ false\ return}\ C\ \mathsf{then}\ t\ \mathsf{else}\ f & = & f
\end{array}$$

Natural numbers. A CwF has natural numbers when, given $\Gamma : \mathsf{Con}$, it has

- $\mathsf{nat} : \mathsf{Ty}\ \Gamma$;
- $\mathsf{zero} : \mathsf{Tm}\ \Gamma\ \mathsf{nat}$;
- for $n : \mathsf{Tm}\ \Gamma\ \mathsf{nat}$, there exists $\mathsf{succ}\ n : \mathsf{Tm}\ \Gamma\ \mathsf{nat}$;
- for $n : \mathsf{Tm}\ \Gamma\ \mathsf{nat}$ and $C : \mathsf{Ty}\ (\Gamma, \mathsf{nat})$ and $z : \mathsf{Tm}\ \Gamma\ C[\mathsf{id}, \mathsf{zero}]$ and $s : \mathsf{Tm}\ \Gamma\ (\Pi\ \mathsf{nat}\ \Pi\ C\ C[(\mathsf{p}, \mathsf{succ}\ \mathsf{q}) \circ \mathsf{p}])$, there exists $\mathsf{natrec}_C\ n\ z\ s : \mathsf{Tm}\ \Gamma\ C[\mathsf{id}, n]$

For the successor I use Π-types when I could have instead put things in the context. Both are valid and I am often favorable to the idea that things should be kept separate as much as possible, but the reliance on application and everything allows us to avoid proving the same things several times, especially since manipulating contexts is so tedious with categories.

such that

$$\begin{array}{rcl}
\mathsf{nat}[\sigma] & = & \mathsf{nat} \\
\mathsf{zero}[\sigma] & = & \mathsf{zero} \\
(\mathsf{succ}\ n)[\sigma] & = & \mathsf{succ}\ n[\sigma] \\
(\mathsf{natrec}_C\ n\ z\ s) & = & \mathsf{natrec}_{C[(\sigma \circ \mathsf{p}), \mathsf{q}]}\ n[\sigma]\ z[\sigma]\ s[\sigma] \\
\mathsf{natrec}_C\ \mathsf{zero}\ z\ s & = & z \\
\mathsf{natrec}_C\ (\mathsf{succ}\ n)\ z\ s & = & \mathsf{app}(\mathsf{app}(s, n), \mathsf{natrec}_C\ n\ z\ s)
\end{array}$$

Tarski universes. A Tarski universe is given by

- a type $\mathsf{U} : \mathsf{Ty}\ \Gamma$;
- for $a : \mathsf{Tm}\ \Gamma\ \mathsf{U}$, a type $\mathsf{El}(a) : \mathsf{Ty}\ \Gamma$;
- for $a : \mathsf{Tm}\ \Gamma\ \mathsf{U}$, $b : \mathsf{Tm}\ \Gamma\ (\Pi\ \mathsf{El}(a)\ \mathsf{U})$, a term $\pi(a, b) : \mathsf{Tm}\ \Gamma\ \mathsf{U}$

such that

$$\begin{array}{rcl}
\mathsf{U}[\sigma] & = & \mathsf{U} \\
\mathsf{El}(a)[\sigma] & = & \mathsf{El}(a[\sigma]) \\
\mathsf{El}(\pi(a, b)) & = & \Pi\ \mathsf{El}(a)\ \mathsf{El}(\mathsf{app}(b[\mathsf{p}], \mathsf{q}))
\end{array}$$

This makes the universe closed under dependent types, we can of course extend this notion to include other constructions.

All of these constructions (dependent products, sums, natural numbers, universes...) can be defined in the **Set** model above. The **Set** model is a simpler case of the **Card** model I will give in the next section so I will not detail it.

Cardinal model

In the previous section I briefly presented the **Set** model. Here I will detail the cardinal model of type theory that we initially wrote with Andrej Bauer. It teaches us interesting things about syntax and especially about ETT.

For every set X there is a unique cardinal $|X|$ such that $X \cong |X|$. We choose a bijection $\beta_X : \mathcal{F}(X, |X|)$. I will write β for β_X when the X is understood.

The existence of such a bijection and this definition of cardinal rely on the axiom of choice. This is not constructive at all.

The **Card** CwF is defined very similarly to **Set**:

1. objects are cardinals;
2. substitutions are given as set-theoretic functions, composed as functions;
3. types in $\mathsf{Ty}\ \Gamma$ are cardinals indexed by Γ;
4. terms of $\mathsf{Tm}\ \Gamma\ A$ are choice functions $t \in \mathcal{F}(\Gamma, \bigcup_{\gamma\in\Gamma} A_\gamma)$ such that $t(\gamma) \in A_\gamma$ for every $\gamma \in \Gamma$;
5. given $\sigma : \Gamma \to \Delta$ and $A : \mathsf{Ty}\ \Delta$, $A[\sigma]$ is defined as the Γ-index family $(A_{\sigma(\gamma)})_\gamma$;
6. given $\sigma : \Gamma \to \Delta$ and $t : \mathsf{Tm}\ \Delta\ A$, $t[\sigma]$ is defined as the choice function $t \circ \sigma$;
7. the empty context is given as the cardinal 1;
8. given $\Gamma : \mathsf{Con}$ and $A : \mathsf{Ty}\ \Gamma$, the extended context Γ, A is defined as the cardinal of the disjoint union $\left|\coprod_{\gamma\in\Gamma} A_\gamma\right|$, the substitution p is defined as $\pi_1 \circ \beta^{-1}$ where π_1 is the first projection of the disjoint union, the term q is defined as $\pi_2 \circ \beta^{-1}$;
9. given $\sigma : \Gamma \to \Delta$ and $a : \mathsf{Tm}\ \Gamma\ A[\sigma]$, the substitution σ, a is defined as the function $\gamma \mapsto \beta(\sigma(\gamma), a(\gamma))$.

Let us verify the coherence conditions:

$$\begin{aligned} \mathsf{p} \circ (\sigma, a) &= \sigma \\ \mathsf{q}[\sigma, a] &= a \end{aligned}$$

Given $\sigma : \Gamma \to \Delta$, $a : \mathsf{Tm}\ \Gamma\ A[\sigma]$ and $\gamma \in \Gamma$ we have

$$\begin{aligned} (\mathsf{p} \circ (\sigma, a))(\gamma) &= \mathsf{p}((\sigma, a)(\gamma)) \\ &= \mathsf{p}(\beta(\sigma(\gamma), a(\gamma))) \\ &= (\pi_1 \circ \beta^{-1})(\beta(\sigma(\gamma), a(\gamma))) \\ &= \pi_1(\beta^{-1}(\beta(\sigma(\gamma), a(\gamma)))) \\ &= \pi_1(\sigma(\gamma), a(\gamma)) \qquad = \quad \sigma(\gamma) \end{aligned}$$

Notice how I verify equality of functions extensionally, that is because we are considering set-theoretic functions here.

and

$$\begin{aligned} (\mathsf{q}[\sigma, a])(\gamma) &= (\mathsf{q} \circ (\sigma, a))(\gamma) \\ &= \mathsf{q}((\sigma, a)(\gamma)) \\ &= \mathsf{q}(\beta(\sigma(\gamma), a(\gamma))) \\ &= (\pi_2 \circ \beta^{-1})(\beta(\sigma(\gamma), a(\gamma))) \\ &= \pi_2(\beta^{-1}(\beta(\sigma(\gamma), a(\gamma)))) \\ &= \pi_2(\sigma(\gamma), a(\gamma)) \qquad = \quad a(\gamma) \end{aligned}$$

I will now echo the previous section and show that **Card** admits all the constructions I introduced then.

Dependent products. Given $A : \mathsf{Ty}\ \Gamma$ and $B : \mathsf{Ty}\ (\Gamma, A)$ we define $\Pi\ A\ B : \mathsf{Ty}\ \Gamma$ as

$$(\Pi\ A\ B)(\gamma) :\equiv \left|\prod_{x \in A(\gamma)} B(\beta(\gamma, x))\right|$$

We define abstraction and application as follows:

With $b : \mathsf{Tm}\ (\Gamma, A)\ B$, $f : \mathsf{Tm}\ \Gamma\ (\Pi\ A\ B)$ and $u : \mathsf{Tm}\ \Gamma\ A$.

$$\begin{array}{lcl} (\lambda b)(\gamma) & :\equiv & \beta(x \mapsto b(\beta(\gamma, x))) \\ \mathsf{app}(f, u)(\gamma) & :\equiv & \beta^{-1}(f(\gamma))(u(\gamma)) \end{array}$$

Now we show

$$\begin{array}{rcl} (\Pi\ A\ B)[\sigma] & = & \Pi\ A[\sigma]\ B[(\sigma \circ \mathsf{p}), \mathsf{q}] \\ (\lambda b)[\sigma] & = & \lambda b[(\sigma \circ \mathsf{p}), \mathsf{q}] \\ (\mathsf{app}(f, u))[\sigma] & = & \mathsf{app}(f[\sigma], u[\sigma]) \\ \mathsf{app}(\lambda b, u) & = & b[\mathsf{id}, u] \\ \lambda \mathsf{app}(f[\mathsf{p}], \mathsf{q}) & = & f \end{array}$$

$$\begin{array}{rcl} (\Pi\ A[\sigma]\ B[(\sigma \circ \mathsf{p}), \mathsf{q}])(\gamma) & = & \left|\prod_{x \in A[\sigma](\gamma)} B[(\sigma \circ \mathsf{p}), \mathsf{q}](\beta(\gamma, x))\right| \\ & = & \left|\prod_{x \in A(\sigma(\gamma))} B(((\sigma \circ \mathsf{p}), \mathsf{q})(\beta(\gamma, x)))\right| \\ & = & \left|\prod_{x \in A(\sigma(\gamma))} B(\beta(\sigma(\gamma), x))\right| \\ & = & (\Pi\ A\ B)(\sigma(\gamma)) \\ & = & (\Pi\ A\ B)[\sigma](\gamma) \end{array}$$

In the argument of B what happens is

$$\begin{array}{cl} & ((\sigma \circ \mathsf{p}), \mathsf{q})(\beta(\gamma, x)) \\ = & \beta((\sigma \circ \mathsf{p})(\beta(\gamma, x)), \mathsf{q}(\beta(\gamma, x))) \\ = & \beta((\sigma(\mathsf{p}(\beta(\gamma, x)))), \mathsf{q}(\beta(\gamma, x))) \\ = & \beta((\sigma(\pi_1(\gamma, x))), \pi_2(\gamma, x)) \\ = & \beta(\sigma(\gamma), x) \end{array}$$

$$\begin{array}{rcl} (\lambda b[(\sigma \circ \mathsf{p}), \mathsf{q}])(\gamma) & = & \beta(x \mapsto b[(\sigma \circ \mathsf{p}), \mathsf{q}](\beta(\gamma, x))) \\ & = & \beta(x \mapsto b(((\sigma \circ \mathsf{p}), \mathsf{q})(\beta(\gamma, x)))) \\ & = & \beta(x \mapsto b(\beta(\sigma(\gamma), x))) \\ & = & (\lambda b)(\sigma(\gamma)) \\ & = & (\lambda b)[\sigma](\gamma) \end{array}$$

$$\begin{array}{rcl} (\mathsf{app}(f, u))[\sigma](\gamma) & = & \mathsf{app}(f, u)(\sigma(\gamma)) \\ & = & \beta^{-1}(f(\sigma(\gamma)))(u(\sigma(\gamma))) \\ & = & \beta^{-1}(f[\sigma](\gamma))(u[\sigma](\gamma)) \\ & = & \mathsf{app}(f[\sigma], u[\sigma])(\gamma) \end{array}$$

$$\begin{array}{rcl} \mathsf{app}(\lambda b, u)(\gamma) & = & \beta^{-1}((\lambda b)(\gamma))(u(\gamma)) \\ & = & \beta^{-1}(\beta(x \mapsto b(\beta(\gamma, x))))(u(\gamma)) \\ & = & (x \mapsto b(\beta(\gamma, x))(u(\gamma)) \\ & = & b(\beta(\gamma, u(\gamma))) \\ & = & b[\mathsf{id}, u](\gamma) \end{array}$$

$$\begin{array}{rcl} (\lambda \mathsf{app}(f[\mathsf{p}], \mathsf{q}))(\gamma) & = & \beta(x \mapsto \mathsf{app}(f[\mathsf{p}], \mathsf{q})(\beta(\gamma, x))) \\ & = & \beta(x \mapsto \beta^{-1}((f \circ \mathsf{p})(\beta(\gamma, x)))(\mathsf{q}(\beta(\gamma, x)))) \\ & = & \beta(x \mapsto \beta^{-1}(f(\gamma))(x))) \\ & = & \beta(\beta^{-1}(f(\gamma))) \\ & = & f(\gamma) \end{array}$$

Dependent sums. Card has dependent sums, as defined as

$$(\Sigma\ A\ B)(\gamma) :\equiv \left|\coprod_{x \in A(\gamma)} B(\beta(\gamma, x))\right|$$

with constructors and destructors:

$$\begin{array}{lcl} \langle a, b\rangle(\gamma) & :\equiv & \beta(a(\gamma), b(\gamma)) \\ p.1 & :\equiv & \pi_1 \circ \beta^{-1} \circ p \\ p.2 & :\equiv & \pi_2 \circ \beta^{-1} \circ p \end{array}$$

We now show

$$\begin{array}{rcl} (\Sigma\ A\ B)[\sigma] & = & \Sigma\ A[\sigma]\ B[(\sigma \circ \mathsf{p}), \mathsf{q}] \\ \langle a, b\rangle[\sigma] & = & \langle a[\sigma], b[\sigma]\rangle \\ (p.1)[\sigma] & = & p[\sigma].1 \\ (p.2)[\sigma] & = & p[\sigma].2 \\ \langle a, b\rangle.1 & = & a \\ \langle a, b\rangle.2 & = & b \end{array}$$

$$\begin{array}{rcl} (\Sigma\ A[\sigma]\ B[(\sigma \circ \mathsf{p}), \mathsf{q}])(\gamma) & = & \left|\coprod_{x \in A[\sigma](\gamma)} B[(\sigma \circ \mathsf{p}), \mathsf{q}](\beta(\gamma, x))\right| \\ & = & \left|\coprod_{x \in A(\sigma(\gamma))} B(\beta(\sigma(\gamma), x))\right| \\ & = & (\Sigma\ A\ B)(\sigma(\gamma)) \\ & = & (\Sigma\ A\ B)[\sigma](\gamma) \\ & & \\ \langle a[\sigma], b[\sigma]\rangle(\gamma) & = & \beta(a(\sigma(\gamma)), b(\sigma(\gamma))) \\ & = & \langle a, b\rangle[\sigma](\gamma) \\ & & \\ (p[\sigma].1)(\gamma) & = & \pi_1(\beta^{-1}(p(\sigma(\gamma)))) \\ & = & (p.1)[\sigma](\gamma) \\ & & \\ (p[\sigma].2)(\gamma) & = & \pi_2(\beta^{-1}(p(\sigma(\gamma)))) \\ & = & (p.2)[\sigma](\gamma) \\ & & \\ (\langle a, b\rangle.1)(\gamma) & = & \pi_1(\beta^{-1}(\langle a, b\rangle(\gamma))) \\ & = & \pi_1(\beta^{-1}(\beta(a(\gamma), b(\gamma)))) \\ & = & a(\gamma) \\ & & \\ (\langle a, b\rangle.2)(\gamma) & = & \pi_2(\beta^{-1}(\langle a, b\rangle(\gamma))) \\ & = & \pi_2(\beta^{-1}(\beta(a(\gamma), b(\gamma)))) \\ & = & b(\gamma) \end{array}$$

Identity types. Crucially, Card has identity types. Given $A : \mathsf{Ty}\ \Gamma$ and $u, v : \mathsf{Tm}\ \Gamma\ A$, we define $\mathsf{Id}_A\ u\ v : \mathsf{Ty}\ \Gamma$ to be the type family

$$(\mathsf{Id}_A\ u\ v)(\gamma) :\equiv \{0 \mid u(\gamma) = v(\gamma)\}$$

The set $\{0 \mid u(\gamma) = v(\gamma)\}$ is indeed a cardinal number, namely 0 when $u(\gamma) \neq v(\gamma)$ and 1 when $u(\gamma) = v(\gamma)$. In fact, $u(\gamma) = v(\gamma)$ is equivalent to $0 \in (\mathsf{Id}_A\ u\ v)(\gamma)$. Thus, we define $\mathsf{refl}_u : \mathsf{Tm}\ \Gamma\ (\mathsf{Id}_A\ u\ u)$ as

$$(\mathsf{refl}_u)(\gamma) :\equiv 0$$

which is the only possible function. Finally, given A : Ty Γ and u : Tm Γ A and C : Tm $(\Gamma, A, \mathsf{Id}_{A[\mathsf{p}]}\ u[\mathsf{p}]\ \mathsf{q})$ and f : Tm Γ $C[\mathsf{id}, u, \mathsf{refl}_u]$ and v : Tm Γ A and e : Tm Γ $(\mathsf{Id}_A\ u\ v)$ we can build J by first remarking that for any $\gamma \in \Gamma$ we have $e(\gamma) \in (\mathsf{Id}_A\ u\ v)(\gamma)$ meaning that $e(\gamma) = 0$ and that $u(\gamma) = v(\gamma)$. As such we define it as

$$(\mathsf{J}_A\ u\ C\ f\ v\ e)(\gamma) \coloneqq f(\gamma)$$

We check

$$\begin{array}{rcl} (\mathsf{Id}_A\ u\ v)[\sigma] & = & \mathsf{Id}_{A[\sigma]}\ u[\sigma]\ v[\sigma] \\ (\mathsf{refl}_a)[\sigma] & = & \mathsf{refl}_{a[\sigma]} \\ (\mathsf{J}_A\ u\ C\ f\ v\ e)[\sigma] & = & \mathsf{J}_{A[\sigma]}\ u[\sigma]\ C[(\sigma \circ \mathsf{p}, \mathsf{q}) \circ \mathsf{p}, \mathsf{q}]\ f[\sigma]\ v[\sigma]\ e[\sigma] \end{array}$$

$$\begin{array}{rcl} \mathsf{Id}_{A[\sigma]}\ u[\sigma]\ v[\sigma] & = & \{0 \mid u[\sigma](\gamma) = v[\sigma](\gamma)\} \\ & = & \{0 \mid u(\sigma(\gamma)) = v(\sigma(\gamma))\} \\ & = & (\mathsf{Id}_A\ u\ v)(\sigma(\gamma)) \\ & = & (\mathsf{Id}_A\ u\ v)[\sigma](\gamma) \end{array}$$

$$\begin{array}{rcl} \mathsf{refl}_{a[\sigma]}(\gamma) & = & 0 \\ & = & (\mathsf{refl}_a)(\sigma(\gamma)) \\ & = & (\mathsf{refl}_a)[\sigma](\gamma) \end{array}$$

The equality for J is proven by going through $f[\sigma](\gamma)$.

Now that we have equality I will show why we do not need an eliminator in this particular case.

Theorem 8.3.1 *The cardinal model validates equality reflection.*

Reminder: Reflection rule

$$\frac{\Gamma \vdash e : u =_A v}{\Gamma \vdash u \equiv v : A}$$

See Definition 6.1.1.

Proof. Assuming that we have p : Tm Γ $(\mathsf{Id}_A\ u\ v)$ we want to show that u and v are equal choice functions, *i.e.* that $u(\gamma) = v(\gamma)$ for all $\gamma \in \Gamma$. Because $0 = p(\gamma) \in (\mathsf{Id}_A\ u\ v)(\gamma)$, it follows that $u(\gamma) = v(\gamma)$. □

The cardinal model is even stronger than that because it validates isomorphism reflection. An isomorphism $A \cong B$ is given by two functions $f : A \longrightarrow B$ and $g : B \longrightarrow A$ that are inverses of each other (for the type-theoretic equality).

Definition 8.3.1 (Isomorphism reflection) *Isomorphism reflection states that isomorphic types are* convertible:

$$\frac{\Gamma \vdash e : A \cong B}{\Gamma \vdash A \equiv B}$$

This consists of a strong invariance principle stating that isomorphic types cannot be distinguished. Univalence is another invariance principle.

Theorem 8.3.2 *The cardinal model validates isomorphism reflection.*

Proof. Assume we are given isomorphic types $A, B : \mathsf{Ty}\ \Gamma$, that is two functions $f : \mathsf{Tm}\ \Gamma\ (\Pi\ A\ B[\mathsf{p}])$ and $g : \mathsf{Tm}\ \Gamma\ (\Pi\ B\ A[\mathsf{p}])$ as well as two equalities showing they are mutual inverses:

$$\begin{aligned} e_1 &: \mathsf{Tm}\ \Gamma\ (\Pi\ A\ \mathsf{Id}_{A[\mathsf{p}]}\ \mathsf{app}(g[\mathsf{p}], \mathsf{app}(f[\mathsf{p}], \mathsf{q}))\ \mathsf{q}) \\ e_2 &: \mathsf{Tm}\ \Gamma\ (\Pi\ B\ \mathsf{Id}_{B[\mathsf{p}]}\ \mathsf{app}(f[\mathsf{p}], \mathsf{app}(g[\mathsf{p}], \mathsf{q}))\ \mathsf{q}) \end{aligned}$$

We want to show equality of A and B that is $A(\gamma) = B(\gamma)$ for every $\gamma \in \Gamma$. Given such γ, we have $f(\gamma) \in \left|\prod_{x \in A(\gamma)} B[\mathsf{p}](\beta(\gamma, x))\right|$ which is to say $f(\gamma) \in |\mathcal{F}(A(\gamma), B(\gamma))|$, similarly $g(\gamma) \in |\mathcal{F}(B(\gamma), A(\gamma))|$. We thus have two functions

$$\begin{array}{rcl} \beta^{-1}(f(\gamma)) & \in & \mathcal{F}(A(\gamma), B(\gamma)) \\ \beta^{-1}(g(\gamma)) & \in & \mathcal{F}(B(\gamma), A(\gamma)) \end{array}$$

We show they are in bijection using p and q:

$$\begin{array}{rcl} \beta^{-1}(p(\gamma)) & \in & \prod_{a \in A(\gamma)} \{0 \mid \beta^{-1}(g(\beta^{-1}(f(\gamma)))) = a\} \\ \beta^{-1}(q(\gamma)) & \in & \prod_{b \in B(\gamma)} \{0 \mid \beta^{-1}(f(\beta^{-1}(g(\gamma)))) = b\} \end{array}$$

Since $A(\gamma)$ and $B(\gamma)$ are cardinals in bijection, they are necessarily equal: $A(\gamma) = B(\gamma)$. □

Similarly the cardinal model validates strange equalities like that of Cantor space and Baire space $2^{\mathbb{N}} = \mathbb{N}^{\mathbb{N}}$ which violates injectivity of Π-types.

Reminder: Injectivity of Π-types

Whenever $\Pi(x : A).B \equiv \Pi(x : A').B'$ then we have both $A \equiv A'$ and $B \equiv B'$.

All this proves that in type theory, one cannot distinguish isomorphic types.

Booleans. Card has booleans, defined as

$$\begin{array}{rcl} \mathsf{bool}(\gamma) & = & 2 \\ \mathsf{true}(\gamma) & = & 1 \\ \mathsf{false}(\gamma) & = & 0 \\ (\mathsf{if}\ b\ \mathsf{return}\ C\ \mathsf{then}\ t\ \mathsf{else}\ f)(\gamma) & = & \begin{cases} t(\gamma) & \text{if } b(\gamma) = 1 \\ f(\gamma) & \text{otherwise} \end{cases} \end{array}$$

The equations are

$$\begin{array}{rcl} \mathsf{bool}[\sigma] & = & \mathsf{bool} \\ \mathsf{true}[\sigma] & = & \mathsf{true} \\ \mathsf{false}[\sigma] & = & \mathsf{false} \\ \begin{pmatrix} \mathsf{if}\ b & \\ \mathsf{return}\ C & \\ \mathsf{then} & t \\ \mathsf{else} & f \end{pmatrix}[\sigma] & = & \begin{array}{ll} \mathsf{if}\ b[\sigma] & \\ \mathsf{return}\ C[(\sigma \circ \mathsf{p}), \mathsf{q}] & \\ \mathsf{then} & t[\sigma] \\ \mathsf{else} & f[\sigma] \end{array} \\ \mathsf{if\ true\ return}\ C\ \mathsf{then}\ t\ \mathsf{else}\ f & = & t \\ \mathsf{if\ false\ return}\ C\ \mathsf{then}\ t\ \mathsf{else}\ f & = & f \end{array}$$

The first three hold trivially. I prove the remaining below.

$$\begin{aligned}
\begin{pmatrix} \text{if } b[\sigma] \\ \text{return } C[(\sigma \circ \mathsf{p}), \mathsf{q}] \\ \text{then} \quad t[\sigma] \\ \text{else} \quad f[\sigma] \end{pmatrix}(\gamma) &= \begin{cases} t(\sigma(\gamma)) & \text{if } b(\sigma(\gamma)) = 1 \\ f(\sigma(\gamma)) & \text{otherwise} \end{cases} \\
&= \begin{pmatrix} \text{if } b \\ \text{return } C \\ \text{then} \quad t \\ \text{else} \quad f \end{pmatrix}[\sigma](\gamma)
\end{aligned}$$

$$\begin{aligned}
\text{if true return } C \text{ then } t \text{ else } f &= \begin{cases} t(\gamma) & \text{if true}(\gamma) = 1 \\ f(\gamma) & \text{otherwise} \end{cases} \\
&= t(\gamma)
\end{aligned}$$

$$\begin{aligned}
\text{if false return } C \text{ then } t \text{ else } f &= \begin{cases} t(\gamma) & \text{if false}(\gamma) = 1 \\ f(\gamma) & \text{otherwise} \end{cases} \\
&= f(\gamma)
\end{aligned}$$

Natural numbers. We define natural numbers as the cardinal of $\mathbb{N}$:

$$\mathsf{nat}(\gamma) := \aleph_0$$

Zero is the constant function

$$\mathsf{zero}(\gamma) := 0$$

while the successor of a natural number is defined as

$$(\mathsf{succ}\ n)(\gamma) := n(\gamma) + 1$$

The eliminator is a bit more complex but not too surprising:

$$\begin{aligned}
&(\mathsf{natrec}_C\ n\ z\ s)(\gamma) := \begin{cases} z(\gamma) & \text{when } n(\gamma) = 0 \\ f(\gamma) & \text{when } n(\gamma) = m + 1 \end{cases} \\
&\text{where } f := \mathsf{app}(\mathsf{app}(s, (x \mapsto m)), \mathsf{natrec}_C\ (x \mapsto m)\ z\ s)
\end{aligned}$$

The equalities are fairly straightforward.

$$\begin{aligned}
\mathsf{nat}[\sigma] &= \mathsf{nat} \\
\mathsf{zero}[\sigma] &= \mathsf{zero} \\
(\mathsf{succ}\ n)[\sigma] &= \mathsf{succ}\ n[\sigma] \\
(\mathsf{natrec}_C\ n\ z\ s)[\sigma] &= \mathsf{natrec}_{C[(\sigma\circ\mathsf{p}),\mathsf{q}]}\ n[\sigma]\ z[\sigma]\ s[\sigma] \\
\mathsf{natrec}_C\ \mathsf{zero}\ z\ s &= z \\
\mathsf{natrec}_C\ (\mathsf{succ}\ n)\ z\ s &= \mathsf{app}(\mathsf{app}(s, n), \mathsf{natrec}_C\ n\ z\ s)
\end{aligned}$$

Tarski universes. If κ is an inaccessible cardinal then we get a universe by taking

$$\begin{aligned}
\mathsf{U}(\gamma) &:= \kappa \\
\mathsf{El}(a)(\gamma) &:= |a(\gamma)| \\
\pi(a, b)(\gamma) &:= \textstyle\prod_{x \in |a(\gamma)|} |\beta^{-1}(b(\gamma))(x)|
\end{aligned}$$

Definition: Inaccessible cardinal

A cardinal κ is inaccessible if it is uncountable, it is not a sum of fewer than κ cardinals that are smaller than κ, and is closed under exponentials (or power sets).

A perhaps more interesting universe is given by using the $\aleph$ function:

$$\begin{array}{lcl} \mathsf{U}(\gamma) & := & \kappa \\ \mathsf{El}(a)(\gamma) & := & \aleph_{a(\gamma)} \\ \pi(a,b) & := & \mu \end{array}$$

where $\aleph_\mu = \prod_{x\in\mathsf{El}a(\gamma)} \mathsf{El}\beta^{-1}(b(\gamma))(x)$.

I will stop here regarding cardinals, or as to why this universe also supports natural numbers, equality and such, because the details do not provide new insights about this model.

Definition: Aleph number

The aleph numbers are cardinals starting from $\aleph_0$ the cardinal of natural numbers, $\aleph_1$ being the next larger cardinal and so on...$\aleph_\alpha$ is actually defined for any ordinal α but it is beyond the scope of this thesis.

This definition of universe works because inaccessible cardinals are fixed-points of the $\aleph$ function.

A warning about syntax. Together all these show that the cardinal model of type theory is indeed a model of type theory with some quirks, teaching us that isomorphic objects cannot be distinguished. There is another important teaching it brings us about syntax.

Once we have reflection, we have to be careful how we deal with β-reduction. In the cardinal model we indeed have that $\mathsf{nat} \longrightarrow \mathsf{bool}$ and $\mathsf{nat} \longrightarrow \mathsf{nat}$ are *equal* types. As such the identity function $\lambda\mathsf{q} : \mathsf{Tm} \bullet (\mathsf{nat} \longrightarrow \mathsf{nat})$ can be typed $\lambda\mathsf{q} : \mathsf{Tm} \bullet (\mathsf{nat} \longrightarrow \mathsf{bool})$. Since $\mathsf{two} := \mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}) : \mathsf{Tm} \bullet \mathsf{nat}$ we have

$A \longrightarrow B$ will be a shorthand for $\Pi\, A\, B[\mathsf{p}]$.

$$\mathsf{app}(\lambda\mathsf{q}, \mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero})) : \mathsf{Tm} \bullet \mathsf{bool}$$

If we allow it to β-reduce we end up with $\mathsf{two} : \mathsf{Tm} \bullet \mathsf{bool}$ which does not hold as

$$\mathsf{two}(\gamma) = 2 \notin 2 = \mathsf{bool}(\gamma)$$

for any $\gamma \in \bullet$. Did we not prove that β-reduction held however? Let us replay the proof of β-reduction above in our specific case:

In fact there is only one such γ: 0.

$$\begin{array}{lcl} \mathsf{app}(\lambda\mathsf{q}, \mathsf{two})(\gamma) & = & \beta^{-1}((\lambda\mathsf{q})(\gamma))(\mathsf{two}(\gamma)) \\ & = & \beta^{-1}(\beta(x \mapsto \mathsf{q}(\beta(\gamma, x))))(\mathsf{two}(\gamma)) \\ & = & (x \mapsto \mathsf{q}(\beta(\gamma, x))(\mathsf{two}(\gamma)) \\ & = & \mathsf{q}(\beta(\gamma, \mathsf{two}(\gamma))) \\ & = & \mathsf{q}[\mathsf{id}, \mathsf{two}](\gamma) \end{array}$$

Simplifying away q and two this becomes

$$\begin{array}{lcl} \mathsf{app}(\lambda\mathsf{q}, \mathsf{two})(\gamma) & = & \beta^{-1}((\lambda\mathsf{q})(\gamma))(2) \\ & = & \beta^{-1}(\beta(x \mapsto x))(2) \\ & = & (x \mapsto x)(2) \\ & = & 2 \\ & = & \mathsf{q}[\mathsf{id}, \mathsf{two}](\gamma) \end{array}$$

The problematic part is the equality

$$\beta^{-1}(\beta(x \mapsto x))(2) = (x \mapsto x)(2)$$

where we simplified the expression $\beta^{-1} \circ \beta$ to the identity function. As usual, I left the subscripts implicit, but that is where the error is coming from: we actually have

$$\beta^{-1}_{2^{\aleph_0}} \circ \beta_{\aleph_0{}^{\aleph_0}}$$

which do not cancel each other out.

This is a case for the use type annotations on λ-abstractions and applications to block β-reduction when the types do not match.

8.4 Type-theoretic models

Type theory can also be a nice setting to give models to type theory. In fact categorical models can fall in this category: as Simon Boulier does in his thesis [67] we can define Con, substitutions, Ty and Tm within Coq or Agda.

[67]: Boulier (2018), 'Extending type theory with syntactic models'

```
Definition Con := Type.

(* Substitutions Γ → Δ *)
Definition Subs (Γ Δ : Con) := Γ -> Δ.

Definition Ty (Γ : Con) := Γ -> Type.

Definition Tm (Γ : Con) (A : Ty Γ) := forall γ : Γ, A γ.

Definition subsTy {Γ Δ} (σ : Subs Γ Δ) (A : Ty Δ) : Ty Γ :=
  fun γ => A (σ γ).

Notation "A [ σ ]" :=
  (subsTy σ A) (at level 0).

Definition subsTm {Γ Δ} (σ : Subs Γ Δ) {A} (t : Tm Δ A)
  : Tm Γ A[σ]
  := fun γ => t (σ γ).

Notation "t [ σ ]ᵗ" :=
  (subsTm σ t) (at level 0).

Definition empty : Con :=
  unit.

Definition cons (Γ : Con) (A : Ty Γ) : Con :=
  Σ (γ : Γ), A γ.

Notation "Γ , A" := (cons Γ A) (at level 20).

Definition p {Γ : Con} {A : Ty Γ} : Subs (Γ, A) Γ :=
  fun '(γ;a) => γ.

Definition q {Γ : Con} {A : Ty Γ} : Tm (Γ, A) A[p] :=
  fun '(γ;a) => a.

Definition scons {Γ Δ} (σ : Subs Γ Δ) {A} (a : Tm Γ A[σ])
  : Subs Γ (Δ, A)
  := fun γ => (σ γ ; a γ).

Lemma p_scons :
  forall Γ Δ (σ : Subs Γ Δ) A (a : Tm Γ A[σ]),
    p ∘ (scons σ a) = σ.
Proof.
  reflexivity.
Qed.

Lemma q_scons :
  forall Γ Δ (σ : Subs Γ Δ) A (a : Tm Γ A[σ]),
    subsTm (scons σ a) q = a.
Proof.
```

```
  reflexivity.
Qed.
```

It is a bit similar to the Set model I introduced earlier only this time we use (Coq) types instead of sets. An interesting thing to note is how the equalities linking p, q and σ, a are proven by **reflexivity**: this means that the equalities hold definitionally, *i.e.* for conversion.

This model also supports Π-types defined using the Π-types of Coq:

```
Definition Π {Γ} (A : Ty Γ) (B : Ty (Γ, A)) : Ty Γ :=
  fun γ => forall (x : A γ), B (γ ; x).

Definition λ {Γ A B} (b : Tm (Γ, A) B) : Tm Γ (Π A B) :=
  fun γ => fun (x : A γ) => b (γ ; x).

Definition app {Γ A B} (f : Tm Γ (Π A B)) (u : Tm Γ A)
  : Tm Γ B[scons id u]
  := fun γ => f γ (u γ).
```

Once again, the required equalities hold definitionally.

The Coq code is a tiny bit more verbose, as some argument are not infered automatically and have to be put in manually.

```
Lemma Πsubs :
  forall Γ Δ (σ : Subs Γ Δ) (A : Ty Δ) (B : Ty (Δ, A)),
    (Π A B)[σ] = Π A[σ] B[scons (σ ∘ p) q].
Proof.
  reflexivity.
Qed.

Lemma λsubs :
  forall Γ Δ (σ : Subs Γ Δ) A B (b : Tm (Δ, A) B),
    (λ b)[σ]ᵗ = λ (b[scons (σ ∘ p) q]ᵗ).
Proof.
  reflexivity.
Qed.

Lemma app_subs :
  forall Γ Δ (σ : Subs Γ Δ) A B (f : Tm Δ (Π A B)) (u : Tm Δ A),
    (app f u)[σ]ᵗ = app f[σ]ᵗ u[σ]ᵗ.
Proof.
  reflexivity.
Qed.

Lemma beta :
  forall Γ A B (b : Tm (Γ, A) B) (u : Tm Γ A),
    app (λ b) u = b[scons id u]ᵗ.
Proof.
  reflexivity.
Qed.

Lemma eta :
  forall Γ A B (f : Tm Γ (Π A B)),
    λ (app f[p]ᵗ q) = f.
Proof.
  reflexivity.
Qed.
```

It also supports a universe with a code for products. This time we rely on the universes of Coq: **Type**. The code of types will be the types themselves and so the decoding function will be the identity.

```
Definition U {Γ} : Ty Γ :=
  fun γ => Type.

Definition El {Γ} (a : Tm Γ U) : Ty Γ :=
  fun γ => a γ.

Definition π {Γ} (a : Tm Γ U) (b : Tm Γ (Π (El a) U))
```

```
  : Tm Γ U
  := fun γ => forall (x : a γ), (b γ x).
```

And the desired equalities hold.

```
Lemma Usubs :
  forall Γ Δ (σ : Subs Γ Δ),
    U[σ] = U.
Proof.
  reflexivity.
Qed.

Lemma Elsubs :
  forall Γ Δ (σ : Subs Γ Δ) (a : Tm Δ U),
    (El a)[σ] = El a[σ]ᵗ.
Proof.
  reflexivity.
Qed.

Lemma Elπ :
  forall Γ (a : Tm Γ U) (b : Tm Γ (Π (El a) U)),
    El (π a b) = Π (El a) (El (app b[p]ᵗ q)).
Proof.
  reflexivity.
Qed.
```

This model is called the standard model.

More generally type theory can serve as a basis to give a semantics to another type theory by encoding it as we shall see in Chapter 9 (Syntax and formalisation of type theory). Another very interesting way of giving models is using so-called syntactic models or translations as we will see in Chapter 10 (Translations). The advantage of those is that the involved meta-theory can be very weak even if we interpret one theory in a much stronger one.

Syntax and formalisation of type theory 9

An interesting fact about type theory (and perhaps one of its main selling points) is that it is a suitable framework in which to reason about type theory. That being said, representing type theory in itself is not entirely straightforward, and some care must be taken. There are actually several choices to be made when representing type theory and they are not all equivalent or with the same pros and cons. I will detail some of them, spending more time on those I ended up choosing and will try to motivate my choice.

In this chapter I will refer to work done in conjunction with Andrej Bauer and Philipp Haselwarter called `formal-type-theory` [6].

[6]: Bauer et al. (2016), *Formalising Type Theory in a modular way for translations between type theories*

9.1 Representation of syntax

I will first focus on the syntactical side of type theory. The first important choice being how to represent variables.

How to deal with variables

When writing programs or expressions with binders on paper or on the computer we will usually use *names*, identifiers for variables like in $\lambda x.\lambda y.x\ y$, the subexpression x refers to the variable bound by the outermost λ, while y refers to the variable bound by the innermost one. The names are not fundamental in what the term represents: $\lambda z.\lambda w.z\ w$ represents *exactly* the same term. Here I α-renamed x to z and y to w. This defines the notion of α-equality or α-equivalence.

α-renaming is introduced in Chapter 3 (Simple type theory).

$$\lambda x.\lambda y.x\ y =_\alpha \lambda z.\lambda w.z\ w$$

However, variable names should only be thought of as an abstraction to represent such terms and not a part of the syntax in itself.

Thinking in terms of variables name can lead to unpleasant examples where α-renaming might become a necessity. For instance, $\lambda x.\lambda x.x$ is perfectly valid but is easier to read when renamed to $\lambda y.\lambda x.x$. This process is called *shadowing*, when several variables bear the same name in scope, it is the innermost that takes precedence. This principle is crucial for compositionality.

Even more problems arise when considering substitutions (after all, that is what variables are for: to be substituted). If you consider the term $t := x\ (\lambda x.x)$ we have two occurrences of the name x but they do *not* represent the same variable, the first x is *free* in the term, while the second is *bound* by the only λ. Now when substituting x for term

u in t, one has to be careful not to replace the bound variable x. The expected result is

$$t[x \leftarrow u] = u\ (\lambda x.x)$$

This used to be called *capture-avoiding* substitutions, but I will call them substitutions, because the operation yielding $u\ (\lambda x.u)$ is unlikely to correspond to what we want.

Several solutions have been proposed to this “problem” like nominal sets [68], Higher Order Abstract Syntax (HOAS) [69] and de Bruijn indices or levels [70]. I think de Bruijn indices are exactly what we want when dealing with λ-terms as they carry the right amount of information. The idea is to use natural numbers instead of names to indicate how many binders to traverse before reaching the one introducing the variable.

[68]: Pitts (2001), ‘Nominal logic: A first order theory of names and binding’
[69]: Pfenning et al. (1988), ‘Higher-order abstract syntax’
[70]: De Bruijn (1978), ‘Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings’

$$\begin{array}{lcl} \lambda x.\ \lambda y.\ \lambda z.\ z & \longrightarrow & \lambda\ \lambda\ \lambda\ \underline{0} \\ \lambda x.\ \lambda y.\ \lambda z.\ y & \longrightarrow & \lambda\ \lambda\ \lambda\ \underline{1} \\ \lambda x.\ \lambda y.\ \lambda z.\ x & \longrightarrow & \lambda\ \lambda\ \lambda\ \underline{2} \end{array}$$

In this setting the same variable can be represented in different ways:

$$\lambda x.\ x\ (\lambda y.\ x\ y)$$

becomes

$$\lambda\ \underline{0}\ (\lambda\ \underline{1}\ \underline{0})$$

so that x is now written $\underline{0}$ and $\underline{1}$ depending on whether it is referenced under the second λ or not. The following diagram should make things more explicit.

$$\begin{tikzcd} \lambda & \underline{0} \arrow[l, bend right] & (\lambda & \underline{1} \arrow[lll, bend left] & \underline{0}) \arrow[ll, bend right] \end{tikzcd}$$

Using this representation, both $\lambda x.x$ and $\lambda y.y$ are written

$$\lambda\ \underline{0}$$

so that α-equality is purely syntactic equality. α-renaming only remains a problem for pretty-printing. This also solves the problem of substitutions potentially capturing free variables: the term $t := x\ (\lambda x.\ x)$ of before is now $\underline{n}\ (\lambda\ \underline{0})$ where n is some number which should point to somewhere in the context (same as the x it replaces).

Notice however that using de Bruijn indices, weakening—*i.e.* putting a term into an extended context—will now affect the term itself. If you consider the following weakening, adding one variable z in the middle of the context, does not affect the term using names.

$$x : A, y : B \vdash x\ y\ (\lambda u.\ x\ y\ u) \rightsquigarrow x : A, z : C, y : B \vdash x\ y\ (\lambda u.\ x\ y\ u)$$

I assign types to the variables in the scope to make clear where the new variable is inserted in the nameless case.

In the context of de Bruijn indices this becomes

$$A, B \vdash \underline{1}\ \underline{0}\ (\lambda\ \underline{2}\ \underline{1}\ \underline{0}) \rightsquigarrow A, C, B \vdash \underline{2}\ \underline{0}\ (\lambda\ \underline{3}\ \underline{1}\ \underline{0})$$

Substitutions

Another choice that is close to the representation of variable is that of substitutions. There are several ways to represent a substitution in itself, but I think the main question is whether to make them *explicit* or not, *i.e.* part of the syntax or not.

With explicit substitutions $t[\sigma]$ is a term in itself and things like evaluation of substitutions come as reduction rules:

$$x[\sigma] \rightarrow \sigma(x)$$

The question of how you represent substitutions is more crucial in this case, and there are several ways to do so, the first being introduced in [71].

[71]: Abadi et al. (1991), 'Explicit substitutions'

In `formal-type-theory` where we formalised syntax of type theory in Coq using explicit substitutions we settled on the following constructions (I will give them using typing rules to make their behaviour explicit):

$$\frac{\Gamma \vdash u : A}{\{u\}_A : \Gamma \to \Gamma, A} \qquad \frac{\Gamma \vdash A}{\mathsf{w}_A : \Gamma, A \to \Gamma} \qquad \frac{\sigma : \Gamma \to \Delta \qquad \Delta \vdash A}{(\sigma|A) : \Gamma, A[\sigma] \to \Delta, A}$$

$$\frac{\vdash \Gamma}{\mathsf{id} : \Gamma \to \Gamma} \qquad \frac{\sigma : \Gamma \to \Delta \qquad \theta : \Delta \to \Xi}{\theta \circ \sigma : \Gamma \to \Xi} \qquad \frac{\vdash \Gamma}{\triangleright : \Gamma \to \bullet}$$

With computation rules such as

$$\begin{array}{rcl}
\underline{0}[\{u\}_A] & \rightarrow & u \\
(\underline{n+1})[\{u\}_A] & \rightarrow & \underline{n} \\
\underline{n}[\mathsf{w}_A] & \rightarrow & \underline{n+1} \\
\underline{0}[(\sigma|A)] & \rightarrow & \underline{0} \\
(\underline{n+1})[(\sigma|A)] & \rightarrow & \underline{n}[\sigma][\mathsf{w}_{A[\sigma]}] \\
 & \dots &
\end{array}$$

However, I find the other option of having substitutions as a meta-operation, outside of the syntax, more natural. It also helps in keeping the syntax and rules to a minimum while turning the substitutions notions above into definitions and properties. For instance weakening becomes a lemma and not something postulated with the constructor w. I will not make a strong case for either choice however as they both have their own interest. Note that in MetaCoq we go with meta-level substitutions.

The weakening lemma can be stated as admissibility of the following rule

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B}{\Gamma, B \vdash \uparrow t : \uparrow A}$$

where $\uparrow t$ is the term t with all its free variables shifted by one.

Annotations

I already touched on this in Chapter 8 (Models of type theory), especially in conclusion of the cardinal model of type theory, but I will once again talk about annotations in a broader setting.

The fist question regards annotations of λ-abstractions with the domain of the function, that is

$$\lambda(x : A).\ t \quad \text{vs} \quad \lambda x.\ t$$

This opposes the so-called Church-style (annotation of the domain) to Curry-style (no annotation whatsoever). There is even a third option of including the codomain $\lambda(x : A).B.\ t$ that seems necessary for ETT according to the cardinal model (it is necessary to restrict β-reduction). Forgetting the latter there is already a pivot between Church- and Curry-style in that plays a determinant role in uniqueness of type. Indeed one cannot hope to have uniqueness of type given that $\lambda x.\ x$ has type $A \longrightarrow A$ for any A. The domain annotation is the minimal information required to get uniqueness of type for λ-abstractions. This minimal information is equivalent to the minimal information required for inferring the type. This information is minimal in the sense that any more annotations can be recovered from a theory without them.

Reminder: Uniqueness of type

A type theory has uniqueness of type when $\Gamma \vdash t : A$ and $\Gamma \vdash t : B$ imply $A \equiv B$.

Universes and types

Another choice to make is whether to separate types and terms. They can be kept separate by putting them in different syntactic classes. One of the great advantages of dependent type theory over simple type theory is that types and terms can be expressed using the same language. That said, keeping them separate might also make sense.

In case we want to keep them distinct, we have to duplicate concepts a bit: substitution, weakening, and even typing: we have judgments $\Gamma \vdash A$ for types and $\Gamma \vdash t : A$ for terms. One of the advantages of this, is that no universes are required to say that *e.g.* Π-types are in the theory:

$$\frac{\Gamma \vdash A \qquad \Gamma, A \vdash B}{\Gamma \vdash \Pi\ A\ B}$$

This difference become clearer when considering the two main notion of universes: namely Tarski and Russell universes.

Tarski universes. I already gave a brief presentation of Tarski universes when looking at categorical models of type theory in Chapter 8 (Models of type theory). A Tarski universe is a type of *codes* of types which are terms representing types. They can de decoded to types using the El (element) type constructor.

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathsf{U}} \qquad\qquad \frac{\Gamma \vdash a : \mathsf{U}}{\Gamma \vdash \mathsf{El}\ a}$$

If we only had the above constructors, our universes would only contain type variables. In order to populate a universe, we introduce codes for the types we are interested in having. For instance here are the codes for Π-types.

$$\frac{\Gamma \vdash a : \mathsf{U} \qquad \Gamma \vdash b : \mathsf{El}\ a \to \mathsf{U}}{\Gamma \vdash \pi(a, b) : \mathsf{U}}$$

To relate them to actual Π-types, we simply add computation rules to El.

$$\mathsf{El}(\pi(a, b)) \rightsquigarrow \Pi(x : \mathsf{El}\ a).\ \mathsf{El}\ (b\ x)$$

Arguably we do not even need that, and to avoid duplication we could simply keep the Els everywhere

$$\frac{\Gamma \vdash a : \mathsf{U} \qquad \Gamma, x : \mathsf{El}\ a \vdash b : \mathsf{U} \qquad \Gamma, x : \mathsf{El}\ a \vdash t : \mathsf{El}\ (b\ x)}{\Gamma \vdash \lambda(x : a).\ t : \mathsf{El}\ (\pi(a, b))}$$

$$\frac{\Gamma \vdash a : \mathsf{U} \qquad \Gamma, x : \mathsf{El}\ a \vdash b : \mathsf{U} \qquad \Gamma \vdash f : \mathsf{El}\ (\pi(a, b)) \qquad \Gamma \vdash u : \mathsf{El}\ a}{\Gamma \vdash f\ u : \mathsf{El}\ (b\ u)}$$

The $\lambda(x : a).\ t$ is not a typo. I enforce here λ to use codes in their syntax since one should not be able to abstract over a type that is not some El a.

I still prefer having a notion of Π-types that is independent of universes and codes so that their behaviour is defined once and for all for every universe.

For instance, having Π-types as a general notion allows us to have Π-types that do not live in any universes but are still valid types.

Russell universes. The other option, the one used in Coq, Agda and most type-theory-based proof assistants, is to use Russell universes. In this setting types are just terms that inhabit a universe:

$$\Gamma \vdash A : \mathsf{Type}$$

As such we do not need something like El to bridge between the two worlds. In a sense, this is a special case of Tarski universes where codes are types and El is the identity function.

With Russell universes Π-types are typed as follows:

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma, A \vdash B : \mathsf{Type}}{\Gamma \vdash \Pi\ A\ B : \mathsf{Type}}$$

When formalising universes, there is a trade-off between the concise Russell universes and the compartmentalised Tarski universes. On the interface with a user however, I think universes are best when they are the least invasive, which works better in a Russell setting. This does not prevent the system from having Tarski universes under the hood.

Formalisation of syntax

Now that I have laid out a few design choices, I will talk a bit about how to formalise the syntax of a type theory. As I already mentioned I worked with two such representations, the one in MetaCoq and the one in `formal-type-theory`, both in Coq.

In MetaCoq we use an inductive type to represent terms of which I give an excerpt below:

```
Inductive term :=
| tRel (n : nat)
| tSort (u : Universe.t)
| tProd (na : name) (A B : term)
| tLambda (na : name) (A t : term)
| tLetIn (na : name) (b B t : term)
| tApp (u v : term)
| tConst (k : kername) (ui : Instance.t)
| tInd (ind : inductive) (ui : Instance.t)
| tConstruct (ind : inductive) (n : nat) (ui : Instance.t)
| tCase
    (indn : inductive * nat)
    (p c : term)
    (brs : list (nat * term))
| tProj (p : projection) (c : term)
| tFix (mfix : mfixpoint term) (idx : nat).
```

- `tRel n` represents the n-th de Bruijn index $\underline{n}$;
- `tSort u` is a universe, as described in the **`Universe`** module;
- `tProd na A B` represents a Π-type (informally $\Pi(n : A).B$), herein `na` is a `name`: even though we use de Bruijn indices, we still have naming annotations for pretty-printing that are irrelevant for typing;
- `tLambda na A t` represents a λ-abstraction $(\lambda(n : A).t)$;
- `tLetIn na b B t` is a let-binding (let $n : B := b$ in t);
- `tApp u v` is application of `u` to `v`;
- the other constructors deal with constants, inductive types and their constructors, pattern-matching and fixed-points.

In MetaCoq we try to stick as close as possible to the Coq implementation, whereas in `formal-type-theory` we used a different approach, trying to be modular on the syntax. Instead of using an inductive type, we use a record with fields for the types of types, terms, etc. as well as the different syntactic constructs. Here is also an excerpt of it.

```
Record Syntax := {
  context      : Type ;
  type         : Type ;
  term         : Type ;
  substitution : Type ;

  ctxempty  : context ;
  ctxextend : context -> type -> context ;

  Prod  : type -> type -> type ;
  Subst : type -> substitution -> type ;
  Uni   : level -> type ;
  El    : level -> term -> type ;

  var     : nat -> term ;
  lam     : type -> type -> term -> term ;
  app     : term -> type -> type -> term -> term ;
  subst   : term -> substitution -> term ;
  uniProd : level -> level -> term -> term -> term ;

  sbzero     : type -> term -> substitution ;
  sbweak     : type -> substitution ;
  sbshift    : type -> substitution -> substitution ;
  sbid       : substitution ;
  sbcomp     : substitution -> substitution -> substitution ;
  sbterminal : substitution
}.
```

We also use de Bruijn indices, but most other choices I presented above are not made: it may seem like we use Tarski universes, separate terms and types, use explicit substitutions and fully-annotated terms; however, these are not enforced because these are not *constructors*. Nothing prevents us from providing an instance where `term` and `type` are the same and `El` is the identity (hence Russell universes), functions that ignore the annotations for `app` and `lam`, or meta-level functions for the substitutions. For instance, here is a version with explicit substitutions, but the rest as I mentioned:

```
Inductive context : Type :=
| ctxempty : context
| ctxextend : context -> term -> context

with term : Type :=
(* Types *)
| Prod : term -> term -> term
| Uni : syntax.level -> term
(* Terms *)
| var : nat -> term
| lam : term -> term -> term
| app : term -> term -> term
| subst : term -> substitution -> term

with substitution : Type :=
| sbzero : term -> term -> substitution
| sbweak : term -> substitution
| sbshift : term -> substitution -> substitution
| sbid : substitution
| sbcomp : substitution -> substitution -> substitution
| sbterminal : substitution
.

Definition S : Syntax := {|
  context      := context ;
  type         := term ;
  term         := term ;
  substitution := substitution ;

  ctxempty  := ctxempty ;
  ctxextend := ctxextend ;

  Prod   := Prod ;
  Subst  := subst ;
  Uni    := Uni ;
  El i T := T ;

  var n            := var n ;
  lam A B t        := lam A t ;
  app u A B v      := app u v ;
  subst u sbs      := subst u sbs ;
  uniProd i j A B := Prod A B ;

  sbzero     := sbzero ;
  sbweak     := sbweak ;
  sbshift    := sbshift ;
  sbid       := sbid ;
  sbcomp     := sbcomp ;
  sbterminal := sbterminal
|}.
```

There is however an important consequence of such a presentation: the syntax is no longer inductive, so there is a priori no injectivity of constructors and we might well define all those types to be `unit`, identifying all those constructs (*e.g.* `lam A B t = app u A B v`). This can make it hard to reason about the syntax, and a lot of properties simply do not hold because you cannot distinguish two different constructs. As such I am not sure this is the best way to go. In the remainder of this thesis, I will focus more on approaches like that of MetaCoq.

9.2 Representation of typing

Now that we have a notion of syntax in mind, we can move on to representing typing derivations—as well as the notion of conversion. Once again, we are faced with many choices.

Paranoia of typing rules

If you have a look at the typing rule of the reflexivity constructor of equality:

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

you can see that we require u to have type A without asking for A to be a type itself. So perhaps we should consider the following rule instead:

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

This time, the question can be whether the context Γ itself makes sense.

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

With Andrej Bauer and Philipp Haselwarter in [6] we describe the two ends of this spectrum as either *economic* or *paranoid* versions of the same typing rule. There is a real question as to which should be chosen. When building derivations it is nicer to be in an economic setting to avoid big and redundant derivations, however when building something *from* a derivation, the paranoid version gives more hypotheses to work on. Leaving aside the practical side, there is of course the question of whether they are equivalent or not.

[6]: Bauer et al. (2016), *Formalising Type Theory in a modular way for translations between type theories*

In `formal-type-theory`, we formalise a proof that paranoid and economic type theories are equivalent provided rules pertaining to variables and constants require that the context is well-formed even in the economic version:

$$\frac{\vdash \Gamma \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}$$

The proof relies on what we called sanity properties (and that I called validity in Chapter 7 (Desirable properties of type theories)): when $\Gamma \vdash t : A$ then $\Gamma \vdash A$ and when $\Gamma \vdash A$ then $\vdash \Gamma$.

The right approach to this I think is to put as much information as required to make the proof of sanity/validity *easy* and then use this proof to prove that the economic versions are admissible. Empirically, it seems that requiring all the premises needed to do type inference on the term is the best practice. To infer the type of $\mathsf{refl}_A\ u$ one needs to check that u has type A, which usually relies on the well-typedness of A, meaning I would use the second rule:

Note that this approach is assuming you do not really care about the derivations that are produced. If you want to compute on them, having big derivations can be a problem and as such it might be worth it to put in the extra work to show them equivalent to their paranoid versions while keeping the data as small as possible.

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

It is also easy to see how proving $\Gamma \vdash u =_A u$ will be direct from the hypotheses.

Conversion

Conversion can also come in different flavours. In Coq, conversion is derived from reduction (as a congruence closure) but is also extended to take η-expansion into account.

More recently it also had to be modified to include marks to deal with definitionally proof-irrelevant universe `SProp`.

$$\frac{u \rightarrow w \qquad w \equiv v}{u \equiv v} \qquad \frac{u \equiv w \qquad v \rightarrow w}{u \equiv v} \qquad \frac{u =_{\alpha\eta} v}{u \equiv v}$$

Since it is untyped, we have to make sure the type is well-formed in the conversion rule:

$$\frac{\Gamma \vdash t : A \qquad A \equiv B \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash t : B}$$

In Agda on the other hand, conversion is *typed*, that is it is a judgment of the form

$$\Gamma \vdash u \equiv v : A$$

This has a serious advantage in that it allows for rules like

$$\frac{}{\Gamma \vdash u \equiv v : \mathsf{unit}}$$

stating that two terms of the unit type are convertible which is not feasible in an untyped setting. In general it simplifies all type-based conversion rules. As you can see with the rule above, $\Gamma \vdash u \equiv v : A$ need not imply $\Gamma \vdash u : A$ or $\Gamma \vdash v : A$. It could however, it is again a design choice. If we do not require it then the conversion rule is similar

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \equiv B : \mathsf{Type} \qquad \Gamma \vdash B : \mathsf{Type}}{\Gamma \vdash t : B}$$

Typed conversion also has shortcomings. For instance, proving injectivity of Π-types can become harder: it can be proven by either showing that transitivity of conversion can be eliminated (*i.e.* shown admissible), or using a model construction for instance.

Finally, there is a third notion of explicit conversion which stores the conversion proof in the term:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash H : A \equiv B \qquad \Gamma \vdash B}{\Gamma \vdash t^H : B}$$

Conversion is then part of the syntax as a new term constructor.

All three notions of conversion are proven equivalent in the case of PTSs in [33].

[33]: Doorn et al. (2013), 'Explicit convertibility proofs in pure type systems'

How intricate need the concepts be?

If we go back to the notion of typed equality, I mentioned we could enforce typedness by having rules closer to

$$\frac{\Gamma \vdash u : \mathsf{unit} \qquad \Gamma \vdash v : \mathsf{unit}}{\Gamma \vdash u \equiv v : \mathsf{unit}}$$

There is a drawback to this however: conversion now mentions typing and typing mentions conversion, they have to be mutually defined.

In Coq this would look like

```
Inductive typing (Γ : context) : term -> term -> Type :=
| typing_lam :
    forall Γ t A B,
      typing (Γ, A) t B ->
      typing Γ (tLambda A t) (tProd A B)

(* ... *)

| typing_conv :
    forall Γ t A B s,
      typing Γ t A ->
      conv Γ A B (tSort s) ->
      typing Γ t B

with conv (Γ : context) : term -> term -> term -> Type :=
| conv_unit :
    forall Γ u v,
      typing Γ u tUnit ->
      typing Γ v tUnit ->
      conv Γ u v tUnit

(* ... *).
```

In contrast, in MetaCoq where conversion is untyped, we first define conversion and then typing on top of it.

We can push things even further and define syntax at the same time as typing as well. This is called well-typed syntax, and is presented in [51] using QITs. The idea is to define contexts, types, terms and substitutions (very similar to categorical models, see Chapter 8 (Models of type theory)) mutually, in a typed way:

[51]: Altenkirch et al. (2016), 'Type theory in type theory using quotient inductive types'

```
data Con : Set
data Ty  : Con → Set
data Tms : Con → Con  → Set
data Tm  : ∀ Γ → Ty Γ → Set
```

Contexts are defined with the empty context and the extension as usual, except now they are well-formed by construction.

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

For types we will stick to Π-types, but we will also have a constructor for (explicit) substitutions.

```
data Ty where
  _[_]T : ∀ {Γ Δ} → Ty Δ → Tms Γ Δ → Ty Γ
  Π     : ∀ {Γ} → (A : Ty Γ) (B : Ty (Γ , A)) → Ty Γ
```

Substitutions which are given like so.

```
data Tms where
  ε   : ∀ {Γ} → Tms Γ •
  _,_ : ∀ {Γ Δ A} → (δ : Tms Γ Δ) → Tm Γ (A[δ]T) → Tms Γ (Δ , A)
  id  : ∀ {Γ} → Tms Γ Γ
  _∘_ : ∀ {Γ Δ Ξ} → Tms Δ Ξ → Tms Γ Δ → Tms Γ Ξ
  π₁  : ∀ {Γ A} → Tms (Γ , A) Γ
```

Finally for terms we have something like this, pretty standard except the application is not the usual[1] and is rather the inverse operation to λ-abstraction.

1: In fact I would not call it application, as nothing is applied yet in **app**. It is a key element to getting it however, usual application of `(u : Tm Γ (Π A B))` and `(v : Tm Γ A)` is given by `(app u)[id, u]t`.

```
data Tm where
  _[_]t : ∀ {Γ Δ A} → Tm Δ A → (δ : Tms Γ Δ) → Tm Γ (A[δ]T)
  π₂    : ∀ {Γ A} → Tm (Γ , A) (A[π₁]T)
  app   : ∀ {Γ A B} → Tm Γ (Π A B) → Tm (Γ , A) B
  lam   : ∀ {Γ A B} → Tm (Γ , A) B → Tm Γ (Π A B)
```

Now for this to work we need to have conversion rules as well. There are two ways to do it. The first, possible in Agda, is to also define those mutually with conversion and then rely on explicit conversion:

```
data ConvTy : (Γ : Con) (A B : Ty Γ) → Set
data ConvTm : (Γ : Con) (A : Ty Γ) (u v : Tm Γ A) → Set

-- ...

data Tm where
  -- ...
  coe : ∀ {Γ A B} → ConvTy Γ A B → Tm Γ A → Tm Γ B
```

The conversion expressed this way is really tedious as we have to deal with `coe` itself in the conversion, as well as extend conversion to context besides the usual type and term conversion.

The solution with Quotient Inductive Types—as the name suggests—is instead to quotient the syntax by the equalities it should verify. For types those would be things like

Note that ≡ corresponds to the propositional equality of Agda.

```
  [id]T : ∀ {Γ A} → A[id]T ≡ A
  [][]T : ∀ {Γ Δ Ξ A} → {σ : Tms Γ Δ} {δ : Tms Δ Ξ} →
          (A[δ]T)[σ]T ≡ A[δ ∘ σ]T
```

and for terms we would have, among other things, the laws for β- and η-equality.

```
  Πβ : ∀ {Γ A B} → {t : Tm (Γ, A) B}  → app (lam t) ≡ t
  Πη : ∀ {Γ A B} → {t : Tm Γ (Π A B)} → lam (app t) ≡ t
```

Congruence laws simply come from the fact that equality is already a congruence. Note that QITs are not yet available in Agda.

I think however that well-typed *syntax* is a misnomer because the data we are constructing represents derivations rather than terms. It is very interesting but does not supersede the good old syntax with a separate typing derivation construction. For one, it allows us to manipulate syntax in an untyped way, and later show that this manipulation is type-preserving. It is important in order to have program translations compute relatively efficiently as we will see in Chapter 10 (Translations) and becomes crucial when derivations are *big*[2] since terms (as in untyped syntax) are usually much smaller. Another point is that it does not really make sense to write a type-checker for a well-typed syntax, something that I expose in Part 'A verified type-checker for Coq, in Coq' on page 128.

2: This is the case in the translation I show in Part 'Elimination of Reflection'

It could however make sense for a type-checker to produce well-typed syntax.

Translations 10

Syntactic translations are a special case of program transformations suited for type theory, as well as a method of choice to get models of type theories. This is better studied in Simon Boulier's thesis [1, 67] but in order to keep this document self contained, I will do my best to give a meaningful excerpt here.

[1]: Boulier et al. (2017), 'The Next 700 Syntactical Models of Type Theory'
[67]: Boulier (2018), 'Extending type theory with syntactic models'

10.1 Syntactic translations

Definition

A *syntactic* translation, as the name suggests, operates on the syntax of terms. It has a source theory $\mathcal{S}$ and a target theory $\mathcal{T}$ with possibly different syntax and typing rules—though usually $\mathcal{S}$ is an extension of $\mathcal{T}$ (more on that later).

A translation is then given as two functions:

- $[.]$ taking $\mathcal{S}$ terms to $\mathcal{T}$ terms;
- $[\![.]\!]$ taking $\mathcal{S}$ types to $\mathcal{T}$ types.

And $[\![.]\!]$ is—often—canonically extended to contexts:

$$\begin{array}{lcl} [\![\bullet]\!] & := & \bullet \\ [\![\Gamma, x : A]\!] & := & [\![\Gamma]\!], x : [\![A]\!] \end{array}$$

It is not always the case that the $[\![\bullet]\!]$ is also the empty context, and sometimes one variable is translated to several, but in a lot of cases it can still be encoded that way.

Usually, $[\![.]\!]$ is built using $[.]$ and sometimes corresponds exactly to $[.]$. The translation is often done in an homomorphic way.

There are two main properties expected of a translation: type preservation and preservation of falsehood.

Definition 10.1.1 (Type preservation) *A translation $[.]$, $[\![.]\!]$ is type preserving when $\Gamma \vdash t : A$ implies $[\![\Gamma]\!] \vdash [t] : [\![A]\!]$.*

Definition 10.1.2 (Preservation of falsehood) *A translation $[.]$, $[\![.]\!]$ from $\mathcal{S}$ to $\mathcal{T}$ is said to preserve falsehood when the translation of the empty type implies falsehood in the target:*

$$\vdash_{\mathcal{T}} [\![\bot_{\mathcal{S}}]\!] \to \bot_{\mathcal{T}}$$

If a translation satisfies both those properties, we can conclude consistency of $\mathcal{S}$ relatively to $\mathcal{T}$.

Theorem 10.1.1 (Relative consistency) *If there is a translation from $\mathcal{S}$ to $\mathcal{T}$ that preserves typing and falsehood, then the consistency of $\mathcal{T}$ implies that of $\mathcal{S}$.*

Proof. Assume a translation $[.], [\![.]\!]$ from $\mathcal{S}$ to $\mathcal{T}$ that preserves typing and falsehood. We will prove the contraposition and assume inconsistency in $\mathcal{S}$, that is the existence of a term t such that

$$\vdash_{\mathcal{S}} t : \bot_{\mathcal{S}}$$

By preservation of typing, we get

$$\vdash_{\mathcal{T}} [t] : [\![\bot_{\mathcal{S}}]\!]$$

Moreover, since the translation preserves falsehood, there is a term f such that $\vdash_{\mathcal{T}} f : [\![\bot_{\mathcal{S}}]\!] \to \bot_{\mathcal{T}}$. Thus, we conclude, using application that

$$\vdash_{\mathcal{T}} f\ t : \bot_{\mathcal{T}}$$

in other words that $\mathcal{T}$ is also inconsistent. □

Relative consistency is all the more interesting when the source theory is more complex than the target and even more so when $\mathcal{T}$ is a theory that we trust.

Type preservation can be achieved by proving intermediary properties: substitutivity, preservation of reduction and of conversion.

Definition 10.1.3 (Substitutivity) *A translation* $[.], [\![.]\!]$ *is substitutive when*

$$\begin{array}{rcl} [t\{x \leftarrow u\}] & := & [t]\{x \leftarrow [u]\} \\ [\![A\{x \leftarrow u\}]\!] & := & [\![A]\!]\{x \leftarrow [u]\} \end{array}$$

Here I write $t\{x \leftarrow u\}$ for t where x is substituted by u, rather than $t[x \leftarrow u]$ to avoid confusion with the translation operator also written with square brackets.

Definition 10.1.4 (Preservation of reduction) *A translation* $[.], [\![.]\!]$ *is said to preserve reduction when* $u \rightarrowtriangle v$ *implies* $[u] \rightarrowtriangle [v]$ *(and* $[\![u]\!] \rightarrowtriangle [\![v]\!]$*).*

This can be relaxed by only asking $[u] \rightarrowtriangle^\star [v]$ or $[u] \rightarrowtriangle^+ [v]$.

Definition 10.1.5 (Preservation of conversion) *A translation* $[.], [\![.]\!]$ *is said to preserve conversion when* $u \equiv v$ *implies* $[u] \equiv [v]$ *(and* $[\![u]\!] \equiv [\![v]\!]$*).*

Example: the × bool translation

A simple yet non-trivial example of translation (or class of translations) is the so-called × bool translation, again from [1, 67].

[1]: Boulier et al. (2017), 'The Next 700 Syntactical Models of Type Theory'
[67]: Boulier (2018), 'Extending type theory with syntactic models'

As I said earlier, the source is going to be an extension of the target. I will keep the target pretty basic, as a simple instance of ITT with a hierarchy of universes $\square_i$ (for $i \in \mathbb{N}$) with the following rules:

Refer to Subsection 6.1 (Intensional Type Theory) on page 45 to see what theory we extend here.

- $(\square_i, \square_j) \in \mathsf{Ax}$ for $i \le j$;
- $(\square_i, \square_j, \square_k) \in \mathsf{R}$ for $i, j \le k$.

Otherwise it has to feature a boolean type bool—with the usual true and false—product types $A \times B$ as special cases[1] of Σ-types, and equality.

1: See Section 5.1 (Inductive types and pattern-matching) on page 30.

$$\frac{}{\Gamma \vdash \mathsf{bool} : \square_0} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

The source is going to be the target extended with a new principle corresponding to the negation of functional extensionality, that is the existence of two functions that are extensionally equal, but not equal themselves.

$$\frac{}{\Gamma \vdash \mathsf{notfunext} : \Sigma\ A\ B\ (f\ g : A \to B).\ (\forall x.\ f\ x = g\ x) \times f \neq g}$$

Inequality $x \neq y$ is defined as the negation of equality, that is the type of maps to the empty type: $x = y \to \bot$.

I will write NotFunExt for the type of notfunext in the following.

The idea is to inhabit this extra principle in the target which does not feature it, or rather to inhabit its translation $[\![\mathsf{NotFunExt}]\!]$. In order to distinguish functions of the target, while keeping them observationally equal, we translate them by adding a boolean label to them. Basically, $\lambda x.x$ is sent to $(\lambda x.x, \mathsf{true})$, the true telling us the it might have come from the translation of a function from $\mathcal{S}$. Any function coming with false instead will not be a translated term.

The translation is defined as follows:

$[x]$	$:=$	x
$[\lambda(x : A).\ t]$	$:=$	$(\lambda(x : [\![A]\!]).\ [t], \mathsf{true})$
$[t\ u]$	$:=$	$[t].1\ [u]$
$[(u, v)]$	$:=$	$([u], [v])$
$[p.1]$	$:=$	$[p].1$
$[p.2]$	$:=$	$[p].2$
$[\mathsf{true}]$	$:=$	true
$[\mathsf{false}]$	$:=$	false
$[\mathsf{if}\ t\ \mathsf{return}\ b.P\ \mathsf{then}\ u\ \mathsf{else}\ v]$	$:=$	$\mathsf{if}\ [t]\ \mathsf{return}\ b.[\![P]\!]\ \mathsf{then}\ [u]\ \mathsf{else}\ [v]$
$[\mathsf{refl}_A\ u]$	$:=$	$\mathsf{refl}_{[\![A]\!]}\ [u]$
$[\mathsf{J}(A, u, x.e.P, w, v, p)]$	$:=$	$\mathsf{J}([\![A]\!], [u], x.e.[\![P]\!], [w], [v], [p])$
$[\square_i]$	$:=$	$\square_i$
$[\Pi(x : A).\ B]$	$:=$	$(\Pi(x : [\![A]\!]).\ [\![B]\!]) \times \mathsf{bool}$
$[\Sigma(x : A).\ B]$	$:=$	$\Sigma(x : [\![A]\!]).\ [\![B]\!]$
$[\mathsf{bool}]$	$:=$	bool
$[u =_A v]$	$:=$	$[u] =_{[\![A]\!]} [v]$
$[\![A]\!]$	$:=$	$[A]$

Notice how $[\![.]\!]$ is just defined as $[.]$.

I actually did not provide the full translation here since I omitted the definition of [notfunext]. Writing down the term is rather tedious and boring, so instead I will give an argument as to why it exists: you can provide any type for A and B, for instance $\square_0$ and then for f and g we give $(\lambda(x : \square_0).\ x, \mathsf{true})$ and $(\lambda(x : \square_0).\ x, \mathsf{false})$ that will be pointwise equal (once projected) but different nonetheless.

This translation satisfies all the properties I mentioned in Subsection 'Definition', including preservation of typing and of falsehood. As such it shows that the negation of funext is consistent with ITT. On the other hand, it can be shown that funext is also consistent, and as such independent from ITT. For details, you should again refer to [1, 67] which gives a more comprehensive treatment of this and other translations.

Reminder: Independence

A proposition P is *independent* from a theory $\mathcal{T}$ when both $\mathcal{T} + P$ and $\mathcal{T} + \neg P$ are consistent relatively to $\mathcal{T}$.

[1]: Boulier et al. (2017), 'The Next 700 Syntactical Models of Type Theory'
[67]: Boulier (2018), 'Extending type theory with syntactic models'

10.2 Other translations

As I mentioned earlier, it is not always the case that contexts are translated homomorphically. For instance, in the binary parametricity translation [72] a variable $a : A$ in the context is sent to three variables:

[72]: Bernardy et al. (2012), 'Proofs for free: Parametricity for dependent types'

$$a_0 : [\![A]\!]_0, a_1 : [\![A]\!]_1, a_\varepsilon : [\![A]\!]\ a_0\ a_1$$

but that can still be encoded as a unique variable

$$a : \Sigma\,(x : [\![A]\!]_0)\,(y : [\![A]\!]_1).\ [\![A]\!]\ x\ y$$

Sometimes, the empty context itself is not translated to the empty context as is the case in the forcing translation [73] and such an encoding becomes impossible. Even worse, in this document, in Part 'Elimination of Reflection', I am going to present a translation that is not even syntactic! Indeed, sometimes the translation can only be conducted at the derivation level: derivations in $\mathcal{S}$ are sent to derivations in $\mathcal{T}$. It often reflects the fact that terms in $\mathcal{S}$ are not enough to recover the derivation when it exists[2]. We could however imagine performing the translation on some intermediary between the judgment and its derivation, containing just the right amount of information. This brings us back to the discussion on the notion of proof.

[73]: Jaber et al. (2016), 'The definitional side of the forcing'

2: In other words, type-checking is undecidable.

In those cases, the definitions and properties have to be adapted. This usually has to be done on a case-by-case basis as for now there is no general framework to deal with all those kinds of translations.

10.3 Conservative extensions

Translations are a nice way to obtain models of type theories, in particular when showing that a certain principle extending a theory can be given an interpretation and even computational content.

I did not dwell on this but as the new principle is given a definition in the target, we can *pull out* its computational behaviour from there while respecting preservation of reduction.

We can even be more restrictive on translations if we want to show that the new principles of $\mathcal{T}$ are not adding new *truths*, essentially saying that no new theorems in $\mathcal{S}$ become provable when we use the tools of $\mathcal{T}$. There is an appropriate notion that is not specific to translations or even type theory which is that of *conservative extension*.

Definition 10.3.1 (Conservative extension) *An extension $\mathcal{T}_2$ of a theory $\mathcal{T}_1$ is conservative if every theorem of $\mathcal{T}_2$ that can be stated in $\mathcal{T}_1$ is already a theorem of $\mathcal{T}_1$.*

An extension of a theory is a theory that proves at least as many theorems as the one it extends.

In the context of type theories it can be reformulated as follows.

Definition 10.3.2 (Conservative extension of a type theory) *An extension $\mathcal{T}_2$ of a type theory $\mathcal{T}_1$ is conservative if for every judgement $\vdash_2 t : A$ such that $\vdash_1 A$ (i.e. A is a type in $\mathcal{T}_1$), there exists t' such that $\vdash_1 t' : A$.*

We will call a translation *conservative* when it allows to exhibit that the source theory $\mathcal{S}$ is a conservative extension of the target $\mathcal{T}$, typically when every type of $\mathcal{T}$ (*i.e.* $\vdash_{\mathcal{T}} A$)—which makes sense in $\mathcal{S}$ since it extends $\mathcal{T}$—is sent to itself ($[\![A]\!] = A$) or at least implies itself:

$$\vdash_{\mathcal{T}} [\![A]\!] \to A$$

This is a kind of generalisation of preservation of falsehood.

The $\times$ bool translation is not conservative because

$$\vdash [\![\mathsf{NotFunExt}]\!] \not\to \mathsf{NotFunExt}$$

and more generally ITT extended with NotFunExt is not conservative over ITT. I will present a conservative translation in Part 'Elimination of Reflection'.

A simpler example could be that of encodings: say you want to add the booleans bool to a type theory that already features a unit type unit and disjunctions $A + B$, booleans can be encoded as unit + unit so they are not strictly necessary but having them as first-class objects can still be convenient. A translation corresponding to the encoding is conservative and as such we can keep working on the meta-theory of a minimal theory (without the booleans).

$$\begin{array}{lcl}
[x] & := & x \\
[\lambda(x:A).\ t] & := & \lambda(x:[\![A]\!]).\ [t] \\
[t\ u] & := & [t]\ [u] \\
[\mathsf{true}] & := & \mathsf{inl}\ () \\
[\mathsf{false}] & := & \mathsf{inr}\ () \\
[\mathsf{if}\ t\ \mathsf{return}\ b.P\ \mathsf{then}\ u\ \mathsf{else}\ v] & := & \begin{array}{l} \mathsf{match}\ [t]\ \mathsf{return}\ b.[\![P]\!]\ \mathsf{with} \\ \mid \mathsf{inl}\ () \implies [u] \\ \mid \mathsf{inr}\ () \implies [v] \\ \mathsf{end} \end{array} \\
[\mathsf{inl}\ t] & := & \mathsf{inl}\ [t] \\
[\mathsf{inr}\ t] & := & \mathsf{inr}\ [t] \\
\left[\begin{array}{l} \mathsf{match}\ t\ \mathsf{return}\ p.P\ \mathsf{with} \\ \mid \mathsf{inl}\ u \implies a \\ \mid \mathsf{inr}\ v \implies b \\ \mathsf{end} \end{array}\right] & := & \begin{array}{l} \mathsf{match}\ [t]\ \mathsf{return}\ p.[\![P]\!]\ \mathsf{with} \\ \mid \mathsf{inl}\ u \implies [a] \\ \mid \mathsf{inr}\ v \implies [b] \\ \mathsf{end} \end{array} \\
[\square_i] & := & \square_i \\
[\Pi(x:A).\ B] & := & \Pi(x:[\![A]\!]).\ [\![B]\!] \\
[\mathsf{bool}] & := & \mathsf{unit} + \mathsf{unit} \\
 & & \\
[\![A]\!] & := & [A]
\end{array}$$

It is conservative because the translation behaves as the identity on the target.

Once again this translation preserves typing, and is conservative (in particular it preserves falsehood). So it offers a *computational* evidence that booleans are just a convenience provided you already have the unit type and sums. In that sense, a translation can—and perhaps should—be seen as *compilation* or as a compilation phase. Indeed compilation transforms a typically complex program into something more primitive, usually closer to a machine language like assembly. In the example above, we went from a system supporting booleans to a simpler one without them.

In the next part, I am going to expose a translation from ETT to ITT and WTT that is not syntactic but which is conservative.

Elimination of Reflection

What I mean by elimination of reflection 11

I presented earlier ETT and its defining reflection rule. The next few chapters are going to be dedicated to its elimination from type theory: that is, how to make a translation from ETT to type theories that do not feature reflection.

Reminder: Reflection rule

$$\frac{\Gamma \vdash_{\mathsf{x}} e : u =_A v}{\Gamma \vdash_{\mathsf{x}} u \equiv v : A}$$

See Definition 6.1.1

This work gave rise to a publication [3] that focused on translating ETT to ITT. However, with Simon Boulier we worked on another version translating directly to WTT which I am going to present here as well.

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

11.1 Nature of the translation

Syntactic translations are not possible

First of all we have to wonder about what kind of translation is possible. I presented in Chapter 10 (Translations) the notion of syntactic translation. Unfortunately it is not possible to devise a syntactic translation to eliminate reflection from type theory.

Assume we have such a translation given by $[\![.]\!]$ and $[.]$ such that whenever $\Gamma \vdash_{\mathsf{x}} t : A$ we have $[\![\Gamma]\!] \vdash [t] : [\![A]\!]$ in the target type theory. Now let us say we have an inconsistent context in ETT: $\Gamma_\perp$ (one can for instance assume $0 = 1$ or $\forall A, A$), in such a context anything can have any type because conversion has become trivial.

$$\dfrac{\dfrac{\vdots}{\Gamma_\perp \vdash_{\mathsf{x}} 0 : \mathbb{N}} \qquad \dfrac{\dfrac{\vdots}{\Gamma_\perp \vdash_{\mathsf{x}} \_ : \mathbb{N} = \perp}}{\Gamma_\perp \vdash_{\mathsf{x}} \mathbb{N} \equiv \perp : \mathsf{Type}}}{\Gamma_\perp \vdash_{\mathsf{x}} 0 : \perp}$$

Since $\Gamma_\perp$ is inconsistent, anything can be proved from it, including the equality between the two types $\mathbb{N}$ and $\perp$. We then use reflection and conversion.

It thus follows that in the target we have $[\![\Gamma_\perp]\!] \vdash [0] : [\![\perp]\!]$. Similarly we would have $[\![\Gamma_\perp]\!] \vdash [0] : [\![\mathbb{N}]\!]$. This means that both $\perp$ and $\mathbb{N}$ should be translated to similar things (to convertible types in case the target theory has uniqueness of type), without being able to exploit the knowledge that $\Gamma_\perp$ is inconsistent because of the syntactical nature of the translation.

If moreover $[\![\bullet]\!] = \bullet$, knowing that we also have $\vdash_{\mathsf{x}} 0 : \mathbb{N}$, we get $\vdash [0] : [\![\mathbb{N}]\!]$ which means that $[0]$ is a closed term. By preservation of falsehood we get a closed term of type $\perp$ in the target which by strengthening is a proof of $\perp$ in the empty context.

Reminder: Preservation of falsehood

A translation preserves falsehood when there exists t such that

$$\vdash t : [\![\perp]\!] \to \perp$$

Definition: Strengthening

A theory has strengthening if for $\Gamma \vdash t : A$ with t and A closed, then $\vdash t : A$ also holds

This "proof" makes a lot of assumptions so it cannot conclude in a more general setting. For instance strengthening is a strong property to have, it does not hold in ETT but it does in ITT and WTT, as such I believe my assumptions are not that restrictive. One of the reasons such a translation is not possible is that it would translate terms, types and contexts independently when it cannot as ETT terms do not contain any hints with respect to the uses of reflection.

This is actually related to the fact that type-checking is undecidable in ETT. Indeed an algorithm capable of deciding whether $\Gamma \vdash 0 : \bot$ would have to be able to decide whether the context Γ is consistent or not which is itself undecidable.

Our translation(s)

If we go back to the notion of proof, it becomes apparent that syntactic translations do not work because we would not be translating proofs but only partial ones; in other words, terms in ETT are not proofs because they are insufficient to recover a typing derivation[1]. Thus comes the question of what is a *suitable proof* in ETT. There is probably an intermediate structure between the term and the full derivation that fits this role in the form of a term together with explicit casts, however, in the setting of this thesis, we will still use complete typing derivations in ETT as *proofs*.

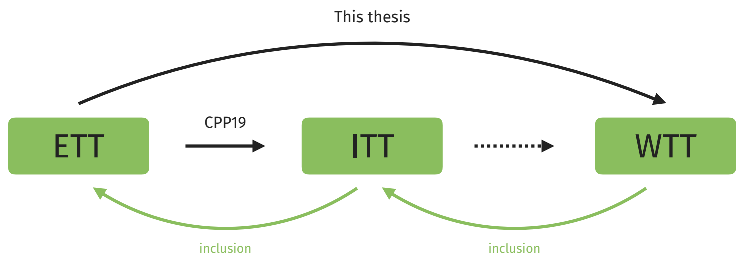
1: Because type-checking is undecidable.

Many problems stem from this approach unfortunately. Since we are translating derivations there is no guarantee that the same term t in $\Gamma \vdash_x t : A$ and $\Delta \vdash_x t : B$ will be translated twice to the same term, this actually goes even for two different derivations of the same judgement $\Gamma \vdash_x t : A$. This seems like a big obstacle to compositionality and would be irredeemable without extra care regarding how individual judgements are translated.

We solve these problems by relating translations of a term (respectively type and context) to the term itself, *syntactically*.

11.2 Target(s) of the translation

Strictly speaking, elimination of reflection should be a translation from a certain type theory T extended with reflection to T itself. To fit this framework, we would provide a translation from ETT to ITT. With Simon Boulier we discovered however that it is possible to do something even stronger and go directly to a much weaker theory where the notion of conversion is removed, WTT.



From a translation of ETT to WTT we get an indirect one from ETT to ITT as well as one from ITT to WTT.

In both cases we exploit the fact that ETT extends ITT which in turn extends WTT.

Note that in both cases, we are not dealing with arbitrary notions of ITT and WTT but ones extended with some principles on equality that will be described in Section 11.4 (Extensionality principles on equality). I will summarise the definitions for ETT, ITT and WTT I use for this translation in Chapter 12 (Framework).

11.3 Goal of the translation

Why would we want to eliminate reflection? The first interest is that it justifies that adding the reflection rule preserves consistency. The main take-away however comes from the fact that we can show that ETT is conservative over both ITT and WTT, meaning that in order to prove a statement in one of those, it is enough to prove it using the reflection rule.

Reminder: Conservativity (roughly)

A theory $\mathcal{S}$ is said to be *conservative* over a theory $\mathcal{T}$ when every statement in $\mathcal{T}$ that is provable in $\mathcal{S}$ is also provable in $\mathcal{T}$.

This can have practical use when proving theorems in a proof assistant like Coq which does not have reflection. If the translation is constructive, it gives rise to an algorithm to transform a proof using reflection into a proof without.

The deduced ITT to WTT even teaches us that computation is more of a commodity than a necessity as proofs can be transformed not to exploit any computational behaviour.

11.4 Extensionality principles on equality

As we said earlier, the ITT and WTT we consider are actually extended with extensional principles on equality. The main principles we require are UIP and functional extensionality. These two principles are valid statements of ITT and WTT which are provable in ETT, to show ETT is a conservative extension of the target, they must be provable in it; as it is not the case we need to extend the target with those principles.

Reminder: UIP

$$\Pi\ x\ y\ (e\ e' : x = y).\ e = e'$$

Reminder: Funext

$$\Pi\ f\ g.\ (\Pi x.\ f\ x = g\ x) \to f = g$$

For UIP it can be shown equivalent to Streicher's axiom K

$$\mathrm{K} \quad : \quad \Pi(A : s).\ \Pi(x : A).\ \Pi(e : x = x).\ e = \mathsf{refl}_x$$

where s is a sort, using the elimination on the identity type. K is provable in ETT by considering the type

$$\Pi(A : s).\ \Pi(x\ y : A).\ \Pi(e : x = y).\ e = \mathsf{refl}_x$$

which is well typed (using the reflection rule to show that e has type $x = x$) and which can be inhabited by elimination of the identity type.

In the same way, funext is provable in ETT as shown below.

$$\begin{array}{lll} & \Pi(x : A).\ f\ x = g\ x & \\ \to & x : A \vdash_{\mathsf{x}} f\ x \equiv g\ x & \text{by reflection} \\ \to & (\lambda(x : A).f\ x) \equiv (\lambda(x : A).g\ x) & \text{by congruence of } \equiv \\ \to & f \equiv g & \text{by } \eta\text{-law} \\ \to & f = g & \end{array}$$

Therefore, applying our translation to the proofs of those theorems in ETT gives corresponding proofs of the same theorems in the target.

As I said, UIP is independent from ITT, as first shown by Hofmann and Streicher using the groupoid model [74], which has recently been extended in the setting of univalent type theory using the simplicial or cubical models [28, 75].

[74]: Hofmann et al. (1998), 'The Groupoid Interpretation of Type Theory'

[75]: Kapulkin et al. (2012), 'The simplicial model of univalent foundations'

[28]: Bezem et al. (2013), 'A Model of Type Theory in Cubical Sets'

Similarly, funext is independent from ITT, this fact is folklore but has recently been formalised by Boulier *et al.* using a simple syntactical translation [1].

[1]: Boulier et al. (2017), 'The Next 700 Syntactical Models of Type Theory'

As previously said in Subsection 2 (Weak Type Theory) of Chapter 6, WTT has to be extended with more principles in the vein of functional extensionality, like extensionality of Π-types for similar reasons.

11.5 Basic idea of the translation

The basic idea behind the translation is to interpret conversion using the internal notion of equality, *i.e.* the identity type. The naive approach would thus be to use transport to simulate the conversion rule.

Reminder: Transport

Transport turns an equality $e : A = B$ into a map $A \to B$.

This means however that for two terms $a : A$ and $b : B$ such that $A \equiv B$ and $u \equiv v$ in ETT, their translations become comparable in the target only up-to the equality between the two types. The type $[u] =_{[A]} [v]$ does not necessarily make sense anymore since $[v]$ might not be of type $[A]$.

To solve this problem we instead use a notion of heterogeneous equality. This is also the approach followed by Oury [76] although we decided on a slightly different presentation of heterogeneous equality to avoid the axioms involved with the use of JMeq.

Reminder: Heterogenous equality

$t \;{}_T{\cong}_U\; u$, equality between $t : T$ and $u : U$ is defined as $\Sigma(p : T = U).\, p_*\, t = u$.

[76]: Oury (2005), 'Extensionality in the calculus of constructions'

During the translation, the same term occurring twice can be translated in two different manners, if the corresponding typing derivations are different. Even the types of the two different translated terms may be different. However, we have the strong property that any two translations of the same term only differ in places where transports of proofs of equality have been injected.

To keep track of this property, we introduce the relation $t \sim t'$ between two terms of the target, of possibly different types[2]. The crux of the proof of the translation is to guarantee that for every two terms t_1 and t_2 such that $\Gamma \vdash t_1 : T_1, \Gamma \vdash t_2 : T_2$ and $t_1 \sim t_2$, there exists p such that $\Gamma \vdash p : t_1 \;{}_{T_1}{\cong}_{T_2}\; t_2$.

2: The relation is fully syntactic.

During the proof, variables of different but (propositionally) equal types are introduced and the context cannot be maintained to be the same for both t_1 and t_2. Therefore, the translation needs to keep track of this duplication of variables, plus a proof that they are heterogeneously equal. This mechanism is similar to what happens in the (relational) internal parametricity translation in ITT introduced by [72] and recently rephrased in the setting of MetaCoq [77]. Namely, a context is not translated as a telescope of variables, but as a telescope of triples

[72]: Bernardy et al. (2012), 'Proofs for free: Parametricity for dependent types'

[77]: Anand et al. (2018), 'Towards Certified Meta-Programming with Typed Template-Coq'

consisting of two variables plus a witness that they are in the parametric relation: $x : A$ is (roughly) sent to $x_1 : A_1, x_2 : A_2, x_\varepsilon : A_\varepsilon\ x_1\ x_2$. In our setting, this amounts to considering telescopes of triples consisting of two variables plus a witness that they are heterogeneously equal. We can express this by considering the following dependent sums:

$$\mathsf{Pack}\ A_1\ A_2 := \Sigma(x_1 : A_1).\, \Sigma(x_2 : A_2).\, x_1\ {}_{A_1}{\cong}_{A_2}\ x_2.$$

This presentation inspired by the parametricity translation is crucial in order to get an effective translation, because it is necessary to keep track of the evolution of contexts when doing the translation on open terms. This ingredient is missing in Oury's work [76], which prevents him from deducing an effective (*i.e.* constructive and computable) translation from his theorem. The construction of this relation is discussed in Chapter 13 (Relating translated expressions).

[76]: Oury (2005), 'Extensionality in the calculus of constructions'

Framework 12

Now that the problem is stated, I will clearly define the type theories I will use as source and target. This is a bit redundant with Chapter 6 (Flavours of type theory) but I think it is worthwhile to make it clear what the translation is operating on.

12.1 Syntax(es)

To make things simple I will consider the syntax of ITT and WTT as extensions of the syntax of ETT. Actually, WTT will also extend ITT.

Syntax of ETT

First, comes the syntax of ETT.

$$
\begin{array}{lcl}
s & \in & \mathcal{S} \\
T, A, B, t, u, v & ::= & x \mid \lambda(x:A).B.t \mid t \mathbin{@_{x:A.B}} u \\
 & \mid & \langle u; v \rangle_{x:A.B} \mid \pi_1^{x:A.B}\ p \mid \pi_2^{x:A.B}\ p \\
 & \mid & \mathsf{refl}_A\ u \mid \mathsf{J}(A, u, x.e.P, w, v, p) \\
 & \mid & \mathsf{ax}(n) \\
 & \mid & s \mid \Pi(x:A).\ B \mid \Sigma(x:A).B \mid u =_A v \\
\Gamma, \Delta & ::= & \bullet \mid \Gamma, x : A \\
\Sigma & ::= & \bullet \mid \Sigma, n : A
\end{array}
$$

Sorts. Sorts are kept abstract (as represented by the generic $\mathcal{S}$) but are not unrestricted. With the sorts should come the sort of a sort (or successor sort) $s+1$, the sort of a Π-type $\mathsf{pi}(s_1, s_2)$ where s_1 and s_2 are the sorts of the domain and codomain respectively, and likewise for each constructor. These are functions, so the underlying PTS is functional. Additionally, we ask that equality of sorts is decidable and that the successor function is injective:

$$s_1 + 1 = s_2 + 1 \longrightarrow s_1 = s_2.$$

Keeping the sorts abstract means that the proof can be instantiated with a lot of different hierarchies, as long as they do not feature cumulativity unfortunately. This will help us apply our result in the homotopy framework.

Definition: Functional PTS

A PTS is *functional* when Ax and R are functional, *i. e.* for each s there is at most one s' such that $(s, s') \in \mathsf{Ax}$ and given s and s' there is at most one s'' such that $(s, s', s'') \in \mathsf{R}$.

Annotations. Although it may look like a technical detail, the use of annotation is more fundamental in ETT than it is in ITT/WTT where it is irrelevant and does not affect the theory. This is actually one of the main differences between our work (and that of Martin Hofmann [78] who has a similar presentation) and the work of Nicolas Oury [76].

[78]: Hofmann (1995), 'Conservativity of equality reflection over intensional type theory'

[76]: Oury (2005), 'Extensionality in the calculus of constructions'

Indeed, using the cardinal model, it is possible to see that the equality $\mathsf{nat} \to \mathsf{nat} = \mathsf{nat} \to \mathsf{bool}$ is independent from the theory, it is thus possible to assume it (as an axiom, or for those that would still not be convinced, simply under a λ that would introduce this equality). In that context, the identity map $\lambda(x : \mathsf{nat}).\ x$ can be given the type $\mathsf{nat} \to \mathsf{bool}$ and we thus type $(\lambda(x : \mathsf{nat}).\ x)\ 0 : \mathsf{bool}$. Moreover, the β-reduction of the non-annotated system used by Oury concludes that this expression reduces to 0, but cannot be given the type bool (as we said, the equality $\mathsf{nat} \to \mathsf{nat} = \mathsf{nat} \to \mathsf{bool}$ is independent from the theory, so the context is consistent). This means we lack subject reduction in this case (or unique typing, depending on how we see the issue). Our presentation has a blocked β-reduction limited to matching annotations: $(\lambda(x : A).B.\ t)\ @_{x:A.B}\ u = t[x \leftarrow u]$, from which subject reduction and unique typing follow.

See Subsection 'Cardinal model' of Chapter 8 (Models of type theory) for more on this.

Although subtle, this difference is responsible for Oury's need for an extra axiom. Indeed, to treat the case of equality of applications in his proof, he needs to assume the congruence rule for heterogeneous equality of applications, which is not provable when formulated with JMeq. Thanks to annotations and our notion of heterogeneous equality, we can prove this congruence rule for applications.

JMAPP
$$\frac{f_1\ {}_{\forall(x:U_1).V_1}\!\cong_{\forall(x:U_2).V_2}\ f_2 \qquad u_1\ {}_{U_1}\!\cong_{U_2}\ u_2}{f_1\ u_1\ {}_{V_1[x\leftarrow u_1]}\!\cong_{V_2[x\leftarrow u_2]}\ f_2\ u_2}$$

Axioms. Another interesting bit is the presence of the $\mathsf{ax}(n)$ term, as well as that of the signature Σ. Indeed, in order for our ETT to be a bit extensible, we add a signature Σ of axioms $n : A$ that can be referenced using $\mathsf{ax}(n)$ but cannot be bound (they are global). For a theory like Coq, having just axioms is a bit weak, but in ETT this is enough to simulate constants and even things such as inductive types: any computation rule can be stated propositionally and still be definitional thanks to reflection.

For instance to simulate the constant

$$n : A := t$$

we can use the following signature

$$n : A,\ n_{def} : n = t$$

Similarly, booleans can be encoded as follows.

$$\begin{array}{lcl}
B & : & s, \\
t & : & B, \\
f & : & B, \\
\mathit{if} & : & \Pi(P : B \to s).\ P\ t \to P\ f \to \Pi(b : B).\ P\ b, \\
\mathit{if}_t & : & \Pi P\ u\ v.\ \mathit{if}\ P\ u\ v\ t = u, \\
\mathit{if}_f & : & \Pi P\ u\ v.\ \mathit{if}\ P\ u\ v\ f = v
\end{array}$$

Note that the sort s is fixed in the eliminator so it still is not that convenient, but it will serve for a few examples.

Syntax of ITT and WTT

ITT is not really far from ETT, we only extend the syntax of terms with funext and UIP:

$$T, A, B, t, u, v \ ::= \ \dots \ | \ \mathsf{funext}(x : A.B, f, g, e) \ | \ \mathsf{uip}(A, u, v, p, q)$$

We build WTT over ITT by adding computation rules and some missing congruence rules (in the likeness of funext). Indeed the elimination of equality provided by J or by Leibniz' principle does not account for equalities under binders; conversion however does and as such, in WTT, we need to add this to the theory. In ITT it is enough to use the congruence of conversion and funext to recover all of those, unfortunately in WTT I could not yet find a way to factorise them. As such we add to the syntax equality constructors for each binder.

$$\begin{array}{rl} T, A, B, t, u, v \ ::= & \dots \ | \ \beta(t, u) \\ & | \ \beta_{\pi_1}(u, v) \ | \ \beta_{\pi_2}(u, v) \\ & | \ \beta_{\mathsf{J}}(u, x.e.P, w) \\ & | \ \overline{\lambda}(x : A).e_B.e_t \ | \ e_u \ \overline{@}_{x:A.e_B} \ v \\ & | \ \overline{\langle} u; e_v \overline{\rangle}_{x:A.e_B} \ | \ \overline{\pi}_1^{x:A.e_B} \ e_p \ | \ \overline{\pi}_2^{x:A.e_B} \ e_p \\ & | \ \overline{\mathsf{J}}(A, u, x.e.e_P, e_w, v, p) \\ & | \ \overline{\Pi}(x : A).e_B \ | \ \overline{\Sigma}(x : A).e_B \end{array}$$

The first rules (with β) are the computation rules. The other are congruence rules going under binders. They might be easier to understand with their typing rules in Subsection 'Typing of WTT'.

Note most of these congruence symbols would no longer be necessary if we were to type things such as J or Σ-types using Π-types so that Π and λ are the only binders. Computation rules would still be required however.

Can ETT still be called a conservative extension?

As I said earlier one of the goals is to show that ETT is a conservative extension of both ITT and WTT. However this cannot strictly be the case if ETT is not first an *extension* of those.

Reminder: Conservative extension

An extension $\mathcal{T}_2$ of a type theory $\mathcal{T}_1$ is *conservative* if for every judgement $\vdash_2 t : A$ such that $\vdash_1 A$ (*i.e.* A is a type in $\mathcal{T}_1$), there exists t' such that $\vdash_1 t' : A$.

This is not really an issue because the terms we add to the syntax of ITT and WTT can be encoded in ETT as definitions. The notion of conservativity can thus be adapted to work in that case: let us write ι the translation that is homomorphically the identity but replaces extra symbols like funext and UIP with their respective proofs in ETT, ETT is conservative over ITT/WTT in the sense that, for every $\vdash A$ such that $\vdash_{\mathsf{x}} t : \iota(A)$, there exists a term t' such that $\vdash t' : A$.

12.2 Typing rules

Similarly to the syntax I will present first the common rules of all involved theories and then detail the specificities of each.

Note that I will differentiate ETT judgements from those of the target by using x as a subscript to the turnstile: $\vdash_{\mathsf{x}}$.

Common typing rules

For all systems, the typing rules are the same for the most part. Conversion and the conversion rule will be the main point of divergence, besides the new terms in ITT and WTT of course.

Well-formedness of contexts.

$$\frac{}{\vdash \bullet} \qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, x : A}(x \notin \Gamma)$$

Types.

The sort constructors here are those mentioned in Section 12.1 (Syntax(es)), though not all were given explicit names.

$$\frac{\vdash \Gamma}{\Gamma \vdash s : s+1} \qquad \frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi(x : A).\ B : \mathsf{pi}(s_1, s_2)}$$

$$\frac{\Gamma \vdash A : s_1 \qquad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Sigma(x : A).\, B : \mathsf{sig}(s_1, s_2)} \qquad \frac{\Gamma \vdash A : s \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : A}{\Gamma \vdash u =_A v : \mathsf{eq}(s)}$$

λ-calculus terms.

$$\frac{\vdash \Gamma \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).B.t : \Pi(x : A).\ B}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma \vdash t : \Pi(x : A).\ B \qquad \Gamma \vdash u : A}{\Gamma \vdash t \ @_{x:A.B}\ u : B[x \leftarrow u]}$$

$$\frac{\Gamma \vdash u : A \qquad \Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s' \qquad \Gamma \vdash v : B[x \leftarrow u]}{\Gamma \vdash \langle u; v \rangle_{x:A.B} : \Sigma(x : A).\, B}$$

$$\frac{\Gamma \vdash p : \Sigma(x : A).\, B}{\Gamma \vdash \pi_1^{x:A.B}\ p : A} \qquad \frac{\Gamma \vdash p : \Sigma(x : A).\, B}{\Gamma \vdash \pi_2^{x:A.B}\ p : B[x \leftarrow \pi_1^{x:A.B}\ p]}$$

Equality terms.

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

$$\frac{\begin{array}{c}\Gamma \vdash A : s \qquad \Gamma \vdash u, v : A \qquad \Gamma, x : A, e : u =_A x \vdash P : s' \\ \Gamma \vdash p : u =_A v \qquad \Gamma \vdash w : P[x \leftarrow u, e \leftarrow \mathsf{refl}_A\ u]\end{array}}{\Gamma \vdash \mathsf{J}(A, u, x.e.P, w, v, p) : P[x \leftarrow v, e \leftarrow p]}$$

Axioms. Although I will omit the global context (or signature) Σ most of the time (it is global after all), you can still imagine that it is present. The axiom typing rule is the only interaction with it. In particular we assume that it is well-formed.

Well-formedness of global context Σ is defined similarly to that of regular contexts.

$$\frac{(n : A) \in \Sigma}{\Gamma \vdash \mathsf{ax}(n) : A}$$

Typing of ETT

As already mentioned several times, the key feature of ETT is its reflection rule, allowing us to go from propositional equality to conversion. This means we need to define conversion in this setting. Here, I will define conversion without relying on reduction, but instead in a typed way.

See Chapter 9 (Syntax and formalisation of type theory).

Computation.

$$\frac{\Gamma \vdash_{\mathsf{x}} A : s \qquad \Gamma, x : A \vdash_{\mathsf{x}} B : s' \qquad \Gamma, x : A \vdash_{\mathsf{x}} t : B \qquad \Gamma \vdash_{\mathsf{x}} u : A}{\Gamma \vdash_{\mathsf{x}} (\lambda(x : A).B.t) @_{x:A.B}\ u \equiv t[x \leftarrow u] : B[x \leftarrow u]}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{x}} A : s \qquad \Gamma \vdash_{\mathsf{x}} u : A \\ \Gamma, x : A, e : u =_A x \vdash_{\mathsf{x}} P : s' \qquad \Gamma \vdash_{\mathsf{x}} w : P[x \leftarrow u, e \leftarrow \mathsf{refl}_A\ u]\end{array}}{\Gamma \vdash_{\mathsf{x}} \mathsf{J}(A, u, x.e.P, w, u, \mathsf{refl}_A\ u) \equiv w : P[x \leftarrow u, e \leftarrow \mathsf{refl}_A\ u]}$$

$$\frac{\Gamma \vdash_{\mathsf{x}} A : s \qquad \Gamma \vdash_{\mathsf{x}} u : A \qquad \Gamma, x : A \vdash_{\mathsf{x}} B : s' \qquad \Gamma \vdash_{\mathsf{x}} v : B[x \leftarrow u]}{\Gamma \vdash_{\mathsf{x}} \pi_1^{x:A.B}\ \langle u; v\rangle_{x:A.B} \equiv u : A}$$

$$\frac{\Gamma \vdash_{\mathsf{x}} A : s \qquad \Gamma \vdash_{\mathsf{x}} u : A \qquad \Gamma, x : A \vdash_{\mathsf{x}} B : s' \qquad \Gamma \vdash_{\mathsf{x}} v : B[x \leftarrow u]}{\Gamma \vdash_{\mathsf{x}} \pi_2^{x:A.B}\ \langle u; v\rangle_{x:A.B} \equiv v : B[x \leftarrow u]}$$

Conversion.

$$\frac{\Gamma \vdash_{\mathsf{x}} t_1 \equiv t_2 : T_1 \qquad \Gamma \vdash_{\mathsf{x}} T_1 \equiv T_2}{\Gamma \vdash_{\mathsf{x}} t_1 \equiv t_2 : T_2} \qquad \frac{\Gamma \vdash_{\mathsf{x}} u : A \qquad \Gamma \vdash_{\mathsf{x}} A \equiv B : s}{\Gamma \vdash_{\mathsf{x}} u : B}$$

Notice that I use conversion and not cumulativity as I need uniqueness of types to proceed.

Reflection.

$$\frac{\Gamma \vdash_{\mathsf{x}} e : u =_A v}{\Gamma \vdash_{\mathsf{x}} u \equiv v : A}$$

I omit the congruence rules as they are quite heavy and boring. Here is for instance the congruence rule for application:

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{x}} A_1 \equiv A_2 : s \qquad \Gamma, x : A_1 \vdash_{\mathsf{x}} B_1 \equiv B_2 : s' \\ \Gamma \vdash_{\mathsf{x}} t_1 \equiv t_2 : \Pi(x : A_1).\ B_1 \qquad \Gamma \vdash_{\mathsf{x}} u_1 \equiv u_2 : A_1\end{array}}{\Gamma \vdash_{\mathsf{x}} t_1 @_{x:A_1.B_1} u_1 \equiv t_2 @_{x:A_2.B_2} u_2 : B_1[x \leftarrow u_1]}$$

Typing of ITT

ITT comes with a conversion deduced from reduction (of which we take the congruent closure). Because of this, the conversion rule will be slightly different to account for the untyped style of conversion.

Reduction.

$$(\lambda(x:A).B.t)\ @_{x:A'.B'}\ u \rightarrow t[x \leftarrow u]$$

$$\mathsf{J}(A, u, x.e.P, w, v, \mathsf{refl}_{A'}\ u') \rightarrow w \qquad \pi_1^{x:A.B}\ \langle u; v\rangle_{x:A'.B'} \rightarrow u$$

$$\pi_2^{x:A.B}\ \langle u; v\rangle_{x:A'.B'} \rightarrow v$$

Notice that in the typed conversion for J, v and u' were both u, and A' was A. This will still be the case for well-typed terms, but the reduction rule should not have to decide conversion to apply. I do something similar for the other computation rules, including β-reduction which only has to be blocked by annotations in ETT.

$\rightarrow$ is also extended to reduce in subterms with rules like

$$\frac{t \rightarrow t'}{\lambda(x:A).B.t \rightarrow \lambda(x:A).B.t'}$$

Conversion. As I said, conversion is simply the reflexive symmetric transitive closure of reduction.

$$\frac{}{u \equiv u} \qquad \frac{u \rightarrow w \qquad w \equiv v}{u \equiv v} \qquad \frac{u \equiv w \qquad v \rightarrow w}{u \equiv v}$$

Granted, this presentation does not *immediately* corresponds to reflexive symmetric transitive closure, but it is equivalent.

The idea is that two terms are convertible if and only if they reduce to the same term. This property only follows from confluence and termination of the reduction however (see Chapter 7 (Desirable properties of type theories)).

Typing.

$$\frac{\Gamma \vdash t : A \qquad A \equiv B \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B} \qquad \frac{\Gamma \vdash e_1, e_2 : u =_A v}{\Gamma \vdash \mathsf{uip}(A, u, v, e_1, e_2) : e_1 = e_2}$$

$$\frac{\Gamma \vdash f, g : \Pi(x:A).\ B \qquad \Gamma \vdash e : \Pi(x:A).\ f\ @_{x:A.B}\ x =_B g\ @_{x:A.B}\ x}{\Gamma \vdash \mathsf{funext}(x:A.B, f, g, e) : f = g}$$

Typing of WTT

For WTT, there is no notion of reduction or conversion, hence the absence of a conversion rule. This leaves us with the typing rules of the extra symbols we added. Two of them are in common with ITT but I will repeat them nonetheless for clarity.

$$\frac{\Gamma \vdash e_1, e_2 : u =_A v}{\Gamma \vdash \mathsf{uip}(A, u, v, e_1, e_2) : e_1 = e_2}$$

$$\frac{\Gamma \vdash f, g : \Pi(x : A).\ B \qquad \Gamma \vdash e : \Pi(x : A).\ f \mathbin{@_{x:A.B}} x =_B g \mathbin{@_{x:A.B}} x}{\Gamma \vdash \mathsf{funext}(x : A.B, f, g, e) : f = g}$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash \beta(t, u) : (\lambda(x : A).B.t) \mathbin{@_{x:A.B}} u = t[x \leftarrow u]}$$

$$\frac{\Gamma, x : A \vdash B : s \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : B[x \leftarrow u]}{\Gamma \vdash \beta_{\pi_1}(u, v) : \pi_1^{x:A.B}\ \langle u; v\rangle_{x:A.B} = u}$$

$$\frac{\Gamma, x : A \vdash B : s \qquad \Gamma \vdash u : A \qquad \Gamma \vdash v : B[x \leftarrow u]}{\Gamma \vdash \beta_{\pi_2}(u, v) : \pi_2^{x:A.B}\ \langle u; v\rangle_{x:A.B} = v}$$

$$\frac{\begin{array}{c}\Gamma \vdash u : A \\ \Gamma, x : A, e : u = x \vdash P : s \qquad \Gamma \vdash w : P[x \leftarrow u, e \leftarrow \mathsf{refl}_A\ u]\end{array}}{\Gamma \vdash \beta_{\mathsf{J}}(u, x.e.P, w) : \mathsf{J}(A, u, x.e.P, w, u, \mathsf{refl}_A\ u) = w}$$

$$\frac{\Gamma, x : A \vdash e_B : B_1 = B_2 \qquad \Gamma, x : A \vdash e_t : t_1 \ {}_{B_1}\!\cong_{B_2}\ t_2}{\Gamma \vdash \overline{\lambda}(x : A).e_B.e_t : \lambda(x : A).B_1.\ t_1 \ {}_{\Pi(x:A).B_1}\!\cong_{\Pi(x:A).B_2}\ \lambda(x : A).B_2.\ t_2}$$

$$\frac{\begin{array}{c}\Gamma, x : A \vdash e_B : B_1 = B_2 \\ \Gamma \vdash e_u : u_1 \ {}_{\Pi(x:A).B_1}\!\cong_{\Pi(x:A).B_2}\ u_2 \qquad \Gamma \vdash v : A\end{array}}{\Gamma \vdash e_u \mathbin{\overline{@}_{x:A.e_B}} v : u_1 \mathbin{@_{x:A.B_1}} v \ {}_{B_1[x \leftarrow v]}\!\cong_{B_2[x \leftarrow v]}\ u_2 \mathbin{@_{x:A.B_2}} v}$$

$$\frac{\begin{array}{c}\Gamma \vdash u : A \\ \Gamma, x : A \vdash e_B : B_1 = B_2 \qquad \Gamma \vdash e_v : v_1 \ {}_{B_1[x \leftarrow u]}\!\cong_{B_2[x \leftarrow u]}\ v_2\end{array}}{\Gamma \vdash \overline{\langle} u; e_v \overline{\rangle}_{x:A.e_B} : \langle u; v_1\rangle_{x:A.B_1} \ {}_{\Sigma(x:A).B_1}\!\cong_{\Sigma(x:A).B_2}\ \langle u; v_2\rangle_{x:A.B_2}}$$

$$\frac{\Gamma, x : A \vdash e_B : B_1 = B_2 \qquad \Gamma \vdash e_p : p_1 \ {}_{\Sigma(x:A).B_1}\!\cong_{\Sigma(x:A).B_2}\ p_2}{\Gamma \vdash \overline{\pi}_1^{x:A.e_B}\ e_p : \pi_1^{x:A.B_1}\ p_1 = \pi_1^{x:A.B_2}\ p_2}$$

$$\frac{\Gamma, x : A \vdash e_B : B_1 = B_2 \qquad \Gamma \vdash e_p : p_1 \ {}_{\Sigma(x:A).B_1}\!\cong_{\Sigma(x:A).B_2}\ p_2}{\Gamma \vdash \overline{\pi}_2^{x:A.e_B}\ e_p : \pi_2^{x:A.B_1}\ p_1 \ {}_{B_1[x \leftarrow \pi_1^{x:A.B_1}\ p_1]}\!\cong_{B_2[x \leftarrow \pi_1^{x:A.B_2}\ p_2]}\ \pi_2^{x:A.B_2}\ p_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash u : A \qquad \Gamma, x : A, e : u = x \vdash e_P : P_1 = P_2 \\ \Gamma \vdash e_w : w_1 \ {}_{P_1[x \leftarrow u, e \leftarrow \mathsf{refl}\ u]}\!\cong_{P_2[x \leftarrow u, e \leftarrow \mathsf{refl}\ u]}\ w_2 \qquad \Gamma \vdash v : A \qquad \Gamma \vdash p : u = v\end{array}}{\Gamma \vdash \overline{\mathsf{J}}(A, u, x.e.e_P, e_w, v, p) : \mathsf{J}(A, u, x.e.P_1, w_1, v, p) \cong \mathsf{J}(A, u, x.e.P_2, w_2, v, p)}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B_1 = B_2}{\Gamma \vdash \overline{\Pi}(x : A).e_B : \Pi(x : A).B_1 = \Pi(x : A).B_2}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B_1 = B_2}{\Gamma \vdash \overline{\Sigma}(x : A).e_B : \Sigma(x : A).B_1 = \Sigma(x : A).B_2}$$

You can notice that I am cheating a little bit here. Indeed, I am already using heterogeneous equality ($\cong$) in the typing rules. If you would indulge me, you can imagine that the definition for heterogeneous equality is *inlined* in the typing rules. That way it is a much more convenient read (and it is already a bit rough).

Remark 12.2.1 Most of these rules have to do with congruence of heterogeneous equality under binders. We can in fact deduce more comprehensive congruence rules. For instance, for application we can inhabit the type

$$u_1 \mathbin{@}_{x:A_1.B_1} v_1 \;{}_{B_1[x \leftarrow v_1]}{\cong}_{B_2[x \leftarrow v_2]}\; u_2 \mathbin{@}_{x:A_2.B_2} v_2$$

from

$$\begin{array}{l}
\Gamma \vdash e_A : A_1 = A_2 \\
\Gamma, p : \mathsf{Pack}\ A_1\ A_2 \vdash e_B : B_1[x \leftarrow \pi_1\ p] = B_2[x \leftarrow \pi_1\ \pi_2\ p] \\
\Gamma \vdash e_u : u_1 \;{}_{\Pi(x:A_1).B_1}{\cong}_{\Pi(x:A_2).B_2}\; u_2 \\
\Gamma \vdash e_v : v_1 \;{}_{A_1}{\cong}_{A_2}\; v_2.
\end{array}$$

This is achieved by relying on the definition of heterogeneous equality and the J eliminator.

We can now dive into the proof itself.

Relating translated expressions 13

We want to define a relation on terms that equates two terms that are the same up to transport. This begs the question of what notion of transport is going to be used. Transport can be defined from elimination of equality as in Chapter 5. However, in order not to confuse the transports added by the translation with the transports that were already present in the source, we consider p_*—*i.e.* the transports added by the translation—as part of the syntax in the reasoning. It will be unfolded to its definition only after the complete translation is performed. This idea is not novel as Hofmann already had a Subst operator that was part of his ITT (written $\mathrm{TT_I}$ in his paper [78]).

[78]: Hofmann (1995), 'Conservativity of equality reflection over intensional type theory'

13.1 Relating terms and their translation

We first define the (purely syntactic) relation $\sqsubset$ between ETT terms and target terms by saying the translated term must be a decoration of the first term by transports. Its purpose is to state how close to the original term its translation is.

As you can see, this relation does not talk about ITT or WTT specific terms.

$$\frac{t_1 \sqsubset t_2}{t_1 \sqsubset p_*\ t_2} \qquad \frac{}{x \sqsubset x} \qquad \frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2}{\Pi(x : A_1).\ B_1 \sqsubset \Pi(x : A_2).\ B_2}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2}{\Sigma(x : A_1).\, B_1 \sqsubset \Sigma(x : A_2).\, B_2} \qquad \frac{A_1 \sqsubset A_2 \qquad u_1 \sqsubset u_2 \qquad v_1 \sqsubset v_2}{u_1 =_{A_1} v_1 \sqsubset u_2 =_{A_2} v_2}$$

$$\frac{}{\mathsf{ax}(n) \sqsubset \mathsf{ax}(n)} \qquad \frac{}{s \sqsubset s} \qquad \frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad t_1 \sqsubset t_2}{\lambda(x : A_1).B_1.t_1 \sqsubset \lambda(x : A_2).B_2.t_2}$$

$$\frac{t_1 \sqsubset t_2 \qquad A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad u_1 \sqsubset u_2}{t_1 \mathbin{@}_{x:A_1.B_1} u_1 \sqsubset t_2 \mathbin{@}_{x:A_2.B_2} u_2}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad t_1 \sqsubset t_2 \qquad u_1 \sqsubset u_2}{\langle t_1; u_1\rangle_{x:A_1.B_1} \sqsubset \langle t_2; u_2\rangle_{x:A_2.B_2}}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad p_1 \sqsubset p_2}{\pi_1^{x:A_1.B_1}\ p_1 \sqsubset \pi_1^{x:A_2.B_1}\ p_2}$$

$$\frac{A_1 \sqsubset A_2 \qquad B_1 \sqsubset B_2 \qquad p_1 \sqsubset p_2}{\pi_2^{x:A_1.B_1}\ p_1 \sqsubset \pi_2^{x:A_2.B_2}\ p_2} \qquad \frac{A_1 \sqsubset A_2 \qquad u_1 \sqsubset u_2}{\mathsf{refl}_{A_1}\ u_1 \sqsubset \mathsf{refl}_{A_2}\ u_2}$$

$$\frac{u_1 \sqsubset u_2 \qquad P_1 \sqsubset P_2 \qquad \begin{array}{c} A_1 \sqsubset A_2 \\ w_1 \sqsubset w_2 \end{array} \qquad v_1 \sqsubset v_2 \qquad p_1 \sqsubset p_2}{\mathsf{J}(A_1, u_1, x.e.P_1, w_1, v_1, p_1) \sqsubset \mathsf{J}(A_2, u_2, x.e.P_2, w_2, v_2, p_2)}$$

From this relation we build a new one ($\sim$) between translated terms. $u \sim v$ basically means that u and v are both decorations of the same term.

$$\sim \coloneqq \sqsupset^{+} . \sqsubset^{+}$$

In order to better reason about it however, we actually define it inductively again.

$$\frac{t_1 \sim t_2}{p_* \, t_1 \sim t_2} \qquad \frac{t_1 \sim t_2}{t_1 \sim p_* \, t_2} \qquad \frac{A_1 \sim A_2 \qquad B_1 \sim B_2}{\Pi(x : A_1).\ B_1 \sim \Pi(x : A_2).\ B_2}$$

$$\frac{A_1 \sim A_2 \qquad B_1 \sim B_2}{\Sigma(x : A_1).\, B_1 \sim \Sigma(x : A_2).\, B_2} \qquad \frac{A_1 \sim A_2 \qquad u_1 \sim u_2 \qquad v_1 \sim v_2}{u_1 =_{A_1} v_1 \sim u_2 =_{A_2} v_2}$$

$$\frac{}{\mathsf{ax}(n) \sim \mathsf{ax}(n)} \qquad \frac{}{s \sim s} \qquad \frac{A_1 \sim A_2 \qquad B_1 \sim B_2 \qquad t_1 \sim t_2}{\lambda(x : A_1).B_1.t_1 \sim \lambda(x : A_2).B_2.t_2}$$

$$\frac{t_1 \sim t_2 \qquad A_1 \sim A_2 \qquad B_1 \sim B_2 \qquad u_1 \sim u_2}{t_1 \mathbin{@_{x:A_1.B_1}} u_1 \sim t_2 \mathbin{@_{x:A_2.B_2}} u_2}$$

$$\frac{A_1 \sim A_2 \qquad B_1 \sim B_2 \qquad t_1 \sim t_2 \qquad u_1 \sim u_2}{\langle t_1; u_1 \rangle_{x:A_1.B_1} \sim \langle t_2; u_2 \rangle_{x:A_2.B_2}}$$

$$\frac{A_1 \sim A_2 \qquad B_1 \sim B_2 \qquad p_1 \sim p_2}{\pi_1^{x:A_1.B_1}\ p_1 \sim \pi_1^{x:A_2.B_1}\ p_2} \qquad \frac{A_1 \sim A_2 \qquad B_1 \sim B_2 \qquad p_1 \sim p_2}{\pi_2^{x:A_1.B_1}\ p_1 \sim \pi_2^{x:A_2.B_2}\ p_2}$$

$$\frac{A_1 \sim A_2 \qquad u_1 \sim u_2}{\mathsf{refl}_{A_1}\ u_1 \sim \mathsf{refl}_{A_2}\ u_2}$$

$$\frac{\begin{array}{c} A_1 \sim A_2 \\ u_1 \sim u_2 \qquad P_1 \sim P_2 \qquad w_1 \sim w_2 \qquad v_1 \sim v_2 \qquad p_1 \sim p_2 \end{array}}{\mathsf{J}(A_1, u_1, x.e.P_1, w_1, v_1, p_1) \sim \mathsf{J}(A_2, u_2, x.e.P_2, w_2, v_2, p_2)}$$

Once more, constructions specific to ITT or WTT are not related with $\sim$: these will only appear in the equalities along which we transport (the p in the highlighted rules).

13.2 Properties of the relation

As I just remarked, the $\sim$ relation is not reflexive; but only in that respect does it fall short of being an equivalence relation.

Lemma 13.2.1 ($\sim$ is a partial equivalence relation) *$\sim$ is symmetric and transitive.*

The goal is to prove that two terms in this relation, that are well-typed in the target type theory, are heterogeneously equal.

Heterogeneous equality, as seen in Subsection 'Heterogenous equality' of Chapter 5, is reflexive, symmetric and transitive. Thanks to UIP, heterogeneous equality collapses to regular equality when taken on the same type on both sides.

Remark 13.2.1 As a corollary, $\cong$ on types corresponds to equality. Indeed when we have $\Gamma \vdash e : A \;{}_{s}{\cong}_{s'}\; B$ we have that $s = s'$, which implies that s and s' have the same sort and thus are syntactically the same (by an inversion argument).

For heterogeneous equality to be a congruence however we have to rely on funext in ITT, and on the several equality constructors we introduce in WTT like $\overline{\lambda}$, $\overline{\Pi}$, etc.

Before we can prove the fundamental lemma stating that two terms in relation are heterogeneously equal, we need to consider another construction. As explained earlier, when proving the property by induction on terms, we introduce variables in the context that are equal only up to heterogeneous equality. This phenomenon is similar to what happens in the parametricity translation [72]. Our fundamental lemma on the decoration relation $\sim$ assumes two related terms of potentially different types T_1 and T_2 to produce an heterogeneous equality between them. For induction to go through under binders (e.g. for dependent products and abstractions), we hence need to consider the two terms under different, but heterogeneously equal contexts. Therefore, the context we produce will not only be a telescope of variables, but rather a telescope of triples consisting of two variables of possibly different types, and a witness that they are heterogeneously equal. To make this precise, we define the following macro:

[72]: Bernardy et al. (2012), 'Proofs for free: Parametricity for dependent types'

$$\mathsf{Pack}\ A_1\ A_2 := \Sigma(x : A_1).\, \Sigma(y : A_2).\, x \cong y$$

together with its projections

$$\mathsf{Proj}_1\ p := \pi_1\ p \qquad \mathsf{Proj}_2\ p := \pi_1\ \pi_2\ p \qquad \mathsf{Proj}_\mathsf{e}\ p := \pi_2\ \pi_2\ p.$$

We can then extend this notion canonically to contexts of the same length that are well formed using the same sorts:

That is $\Gamma_1 := x_1 : A_1, \ldots, x_n : A_n, \Gamma_2 := y_1 : B_1, \ldots, y_n : B_n$ where there exists s_i such that $A_i : s_i$ and $B_i : s_i$ for each i.

$$\begin{array}{l}\mathsf{Pack}\ (\Gamma_1, x : A_1)\ (\Gamma_2, x : A_2) := \\ (\mathsf{Pack}\ \Gamma_1\ \Gamma_2), x : \mathsf{Pack}\ (A_1[\gamma_1])\ (A_2[\gamma_2]) \\ \\ \mathsf{Pack}\ \bullet\ \bullet := \bullet.\end{array}$$

When we pack contexts, we also need to apply the correct projections for the types in that context to still make sense. Assuming two contexts Γ_1 and Γ_2 of the same length, we can define left and right substitutions:

$$\begin{array}{ll}\gamma_1 & := [x \leftarrow \mathsf{Proj}_1\ x \mid (x : \_) \in \Gamma_1] \\ \gamma_2 & := [x \leftarrow \mathsf{Proj}_2\ x \mid (x : \_) \in \Gamma_2].\end{array}$$

These substitutions implement lifting of terms to packed contexts:

we have $\Gamma, \mathsf{Pack}\ \Gamma_1\ \Gamma_2 \vdash t[\gamma_1] : A[\gamma_1]$ whenever $\Gamma, \Gamma_1 \vdash t : A$ and $\Gamma, \mathsf{Pack}\ \Gamma_1\ \Gamma_2 \vdash t[\gamma_2] : A[\gamma_2]$ whenever $\Gamma, \Gamma_2 \vdash t : A$.

For readability, when Γ_1 and Γ_2 are understood we will write Γ_p for $\mathsf{Pack}\ \Gamma_1\ \Gamma_2$.

Implicitly, whenever we use the notation $\mathsf{Pack}\ \Gamma_1\ \Gamma_2$ it means that the two contexts are of the same length and well-formed with the same sorts. Note that I will also use $\vdash$ by opposition to $\vdash_\mathsf{x}$ in judgements to indicate that these judgements are in the target, *i.e.* ITT or WTT. We can now tackle the fundamental lemma.

Lemma 13.2.2 (Fundamental lemma) *Let t_1 and t_2 be two terms such that $t_1 \sim t_2$. For all contexts Γ, Γ_1 and Γ_2 we can construct another term p such that whenever $\Gamma, \Gamma_1 \vdash t_1 : T_1$ and $\Gamma, \Gamma_2 \vdash t_2 : T_2$ we have $\Gamma, \mathsf{Pack}\ \Gamma_1\ \Gamma_2 \vdash p : t_1[\gamma_1]\ {}_{T_1[\gamma_1]}{\cong}_{T_2[\gamma_2]}\ t_2[\gamma_2]$.*

Proof. The proof is by induction on the derivation of $t_1 \sim t_2$. We show the three most interesting cases:

- Var

$$\frac{}{x \sim x}$$

If x belongs to Γ, we apply reflexivity—together with unique typing—to conclude. Otherwise, $\mathsf{Proj}_\mathsf{e}\ x$ has the expected type (since $x[\gamma_1] \equiv \mathsf{Proj}_1\ x$ and $x[\gamma_2] \equiv \mathsf{Proj}_2\ x$).

- Application

$$\frac{t_1 \sim t_2 \qquad A_1 \sim A_2 \qquad B_1 \sim B_2 \qquad u_1 \sim u_2}{t_1 \mathbin{@_{x:A_1.B_1}} u_1 \sim t_2 \mathbin{@_{x:A_2.B_2}} u_2}$$

We have $\Gamma, \Gamma_1 \vdash t_1 \mathbin{@_{x:A_1.B_1}} u_1 : T_1$ and $\Gamma, \Gamma_2 \vdash t_2 \mathbin{@_{x:A_2.B_2}} u_2 : T_2$ which means by inversion[1] that the subterms are well-typed. We apply the induction hypothesis and then conclude.

- TransportLeft

$$\frac{t_1 \sim t_2}{p_*\ t_1 \sim t_2}$$

We have $\Gamma, \Gamma_1 \vdash p_*\ t_1 : T_1$ and $\Gamma, \Gamma_2 \vdash t_2 : T_2$. By inversion[2] we have $\Gamma, \Gamma_1 \vdash p : T_1' = T_1$ and $\Gamma, \Gamma_1 \vdash t_1 : T_1'$. By induction hypothesis we have e such that $\Gamma, \Gamma_\mathsf{p} \vdash e : t_1[\gamma_1] \cong t_2[\gamma_2]$. From transitivity and symmetry we only need to provide a proof of $t_1[\gamma_1] \cong p[\gamma_1]_*\ t_1[\gamma_1]$ which is inhabited by $\langle p[\gamma_1]; \mathsf{refl}\ (p[\gamma_1]_*\ t_1[\gamma_1])\rangle_{\_.\_}$.

□

1: It is important to note that we are not doing induction on any typing derivation here so we do not have to argue that inversion indeed produces smaller derivations.

2: Inversion gives us some T_1' and T_1'' such that $p : T_1' = T_1''$ and $t : T_1'$ and $T_1'' \equiv T_1$ meaning we can give the type $T_1' = T_1$ to p by conversion.

We can also prove that $\sim$ preserves substitution.

Lemma 13.2.3 *If $t_1 \sim t_2$ and $u_1 \sim u_2$ then $t_1[x \leftarrow u_1] \sim t_2[x \leftarrow u_2]$.*

Proof. We proceed by induction on the derivation of $t_1 \sim t_2$. □

The fundamental lemma, as the name suggests, is the main ingredient to proving the translation correct (and actually building the translation). Now that we have it, we can proceed with the translation.

Translation from ETT to ITT and WTT 14

14.1 The translation

We now define the translations (let us stress the plural here) of an extensional judgment. First, we extend $\sqsubset$ canonically to contexts.

$\Gamma \sqsubset \overline{\Gamma}$ holds when they bind the same variables and the types are in relation for $\sqsubset$.

Before defining the translation, we define a set $[\![\Gamma \vdash_{\mathsf{x}} t : A]\!]$ of typing judgments in the target associated to a typing judgment $\Gamma \vdash_{\mathsf{x}} t : A$ in ETT. The idea is that this set describes all the possible translations that lead to the expected property. When $\overline{\Gamma} \vdash \overline{t} : \overline{A} \in [\![\Gamma \vdash_{\mathsf{x}} t : A]\!]$, we say that $\overline{\Gamma} \vdash \overline{t} : \overline{A}$ realises $\Gamma \vdash_{\mathsf{x}} t : A$. The translation will be given by showing that this set is inhabited whenever the judgment in ETT is derivable, by induction on the derivation.

This is a predicate rather than a set. The formalisation is clearer on that fact.

Note that $\overline{t}$ is just a variant like t', the bar is not an operation.

Definition 14.1.1 (Characterisation of possible translations)

- *For any $\vdash_{\mathsf{x}} \Gamma$ we define $[\![\vdash_{\mathsf{x}} \Gamma]\!]$ as a set of valid judgments (in the target) such that $\vdash \overline{\Gamma} \in [\![\vdash_{\mathsf{x}} \Gamma]\!]$ if and only if $\Gamma \sqsubset \overline{\Gamma}$.*
- *Similarly, $\overline{\Gamma} \vdash \overline{t} : \overline{A} \in [\![\Gamma \vdash_{\mathsf{x}} t : A]\!]$ iff $\vdash \overline{\Gamma} \in [\![\vdash_{\mathsf{x}} \Gamma]\!]$ and $A \sqsubset \overline{A}$ and $t \sqsubset \overline{t}$.*

In order to better master the shape of the produced realiser, we state the following lemma which says that we can chose the translated term such that the type has the same head type constructor as the type it realises. This is important for instance for the case of an application, where we do not know *a priori* if the translated function has a dependent product type, which is required to be able to use the typing rule for application.

Lemma 14.1.1 *We can always choose types $\overline{T}$ that have the same head constructor as T.*

Proof. Assume we have $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_{\mathsf{x}} t : T]\!]$. By definition of $\sqsubset$, $T \sqsubset \overline{T}$ means that $\overline{T}$ is shaped $p_* \; q_* \; \dots \; r_* \; \overline{T}'$ with $\overline{T}'$ having the same head constructor as T. By inversion, the subterms are typable, including $\overline{T}'$. Actually, from inversion, we even get that the type of $\overline{T}'$ is a universe. Then, using Lemma 13.2.2 and Lemma 5.3.1, we get $\overline{\Gamma} \vdash e : \overline{T} = \overline{T}'$. We conclude with $\overline{\Gamma} \vdash e_* \; \overline{t} : \overline{T}' \in [\![\Gamma \vdash_{\mathsf{x}} t : T]\!]$. □

Finally, in order for the induction to go through, we need to know that when we have a realiser of a derivation $\Gamma \vdash_{\mathsf{x}} t : T$, we can pick an arbitrary other type realising $\Gamma \vdash_{\mathsf{x}} T$ and still get a new derivation realising $\Gamma \vdash_{\mathsf{x}} t : T$ with that type. This is important for instance for the case of an application, where the type of the domain of the translated function may differ from the type of the translated argument. So we need to be able to change it *a posteriori*.

Lemma 14.1.2 *When we have* $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_{\mathsf{x}} t : T]\!]$ *and* $\overline{\Gamma} \vdash \overline{T}' \in [\![\Gamma \vdash_{\mathsf{x}} T]\!]$ *then we also have* $\overline{\Gamma} \vdash \overline{t}' : \overline{T}' \in [\![\Gamma \vdash_{\mathsf{x}} t : T]\!]$ *for some* $\overline{t}'$.

Proof. By definition we have $T \sqsubset \overline{T}$ and $T \sqsubset \overline{T}'$ and thus $\overline{T} \sim \overline{T}'$. By Lemma 13.2.2 (in the case $\Gamma_1 \equiv \Gamma_2 \equiv \bullet$) we get $\overline{\Gamma} \vdash p : \overline{T} \cong \overline{T}'$ for some p. By Lemma 5.3.1 (and Lemma 14.1.1 to give universes as types to $\overline{T}$ and $\overline{T}'$) we can assume $\overline{\Gamma} \vdash p : \overline{T} = \overline{T}'$. Then $\overline{\Gamma} \vdash p_* \, \overline{t} : \overline{T}'$ is still a translation since $\sqsubset$ ignores transports. □

We can now define the translation. This is done by mutual induction on context well-formedness, typing and conversion derivations. Indeed, in order to be able to produce a realiser by induction, we need to show that every conversion in ETT is translated as an heterogeneous equality.

Theorem 14.1.3 (Translation)

- *If* $\vdash_{\mathsf{x}} \Gamma$ *then there exists* $\vdash \overline{\Gamma} \in [\![\vdash_{\mathsf{x}} \Gamma]\!]$,
- *If* $\Gamma \vdash_{\mathsf{x}} t : T$ *then for any* $\vdash \overline{\Gamma} \in [\![\vdash_{\mathsf{x}} \Gamma]\!]$ *there exist* $\overline{t}$ *and* $\overline{T}$ *such that* $\overline{\Gamma} \vdash \overline{t} : \overline{T} \in [\![\Gamma \vdash_{\mathsf{x}} t : T]\!]$,
- *If* $\Gamma \vdash_{\mathsf{x}} u \equiv v : A$ *then for any* $\vdash \overline{\Gamma} \in [\![\vdash_{\mathsf{x}} \Gamma]\!]$ *there exist* $A \sqsubset \overline{A}, A \sqsubset \overline{A}', u \sqsubset \overline{u}, v \sqsubset \overline{v}$ *and* $\overline{e}$ *such that* $\overline{\Gamma} \vdash \overline{e} : \overline{u} \;{}_{\overline{A}}{\cong}_{\overline{A}'}\; \overline{v}$.

Proof. We prove the theorem by induction on the derivation in ETT. We only show the two most interesting cases of application and conversion besides the reflection case.

- Application

$$\frac{\Gamma, x : A \vdash_{\mathsf{x}} B : s' \qquad \begin{array}{c}\Gamma \vdash_{\mathsf{x}} A : s \\ \Gamma \vdash_{\mathsf{x}} t : \Pi(x : A).\ B\end{array} \qquad \Gamma \vdash_{\mathsf{x}} u : A}{\Gamma \vdash_{\mathsf{x}} t \mathbin{@_{x:A.B}} u : B[x \leftarrow u]}$$

  Using IH together with Lemma 14.1.1 and Lemma 14.1.2 we get $\overline{\Gamma} \vdash \overline{A} : s$ and $\overline{\Gamma}, x : \overline{A} \vdash \overline{B} : s'$ and $\overline{\Gamma} \vdash \overline{t} : \Pi(x : \overline{A}).\ \overline{B}$ and $\overline{\Gamma} \vdash \overline{u} : \overline{A}$ meaning we can conclude $\overline{\Gamma} \vdash \overline{t} \mathbin{@_{x:\overline{A}.\overline{B}}} \overline{u} : \overline{B}[x \leftarrow \overline{u}] \in [\![\Gamma \vdash_{\mathsf{x}} t \mathbin{@_{x:A.B}} u : B[x \leftarrow u]]\!]$.
- Conversion

$$\frac{\Gamma \vdash_{\mathsf{x}} u : A \qquad \Gamma \vdash_{\mathsf{x}} A \equiv B}{\Gamma \vdash_{\mathsf{x}} u : B}$$

  By IH and Lemma 5.3.1 we have $\overline{\Gamma} \vdash \overline{e} : \overline{A} = \overline{B}$ which implies $\overline{\Gamma} \vdash \overline{A} \in [\![\Gamma \vdash_{\mathsf{x}} A]\!]$ by inversion, thus, from Lemma 14.1.2 and IH we get $\overline{\Gamma} \vdash \overline{u} : \overline{A}$, yielding $\overline{\Gamma} \vdash \overline{e}_* \, \overline{u} : \overline{B} \in [\![\Gamma \vdash_{\mathsf{x}} u : B]\!]$.
- Reflection

$$\frac{\Gamma \vdash_{\mathsf{x}} e : u =_A v}{\Gamma \vdash_{\mathsf{x}} u \equiv v : A}$$

  By IH and Lemma 14.1.1 we have $\overline{\Gamma} \vdash \overline{e} : \overline{u} =_{\overline{A}} \overline{v}$ which is almost what we want as a conclusion: we have an homogenous equality when we want an heterogeneous one. We conclude by taking reflexivity for the types and $\overline{e}$ for the equality between terms.

□

14.2 Meta-theoretical consequences

We can check that all ETT theorems whose type are typable in ITT have proofs in ITT as well. This also holds for WTT.

The reason I did not differentiate the two target theories is simply because there was no need. The proof is exactly the same, indeed, the proof never uses the conversion in the target since every conversion from ETT is translated to an equality, without ever relying on conversion (other than to produce equalities corresponding to computation rules in ITT, which are axiomatised in WTT). The two formalisations are proof enough of this.

Corollary 14.2.1 (Conservativity) *If* $\vdash_{\mathsf{x}} t : \iota(T)$ *and* $\vdash T$ *then there exist* $\bar{t}$ *such that* $\vdash \bar{t} : T \in [\![\vdash_{\mathsf{x}} t : \iota(T)]\!]$.

Proof. Since $\vdash \bullet \in [\![\vdash_{\mathsf{x}} \bullet]\!]$, by Theorem 14.1.3, there exists $\bar{t}$ and $\bar{T}$ such that $\vdash \bar{t} : \bar{T} \in [\![\vdash_{\mathsf{x}} t : \iota(T)]\!]$ But as $\vdash T$, we have $\vdash T \in [\![\vdash_{\mathsf{x}} \iota(T)]\!]$, and, using Lemma 14.1.2, we obtain $\vdash \bar{t} : T \in [\![\vdash_{\mathsf{x}} t : \iota(T)]\!]$. □

Corollary 14.2.2 (Relative consistency) *Assuming the target (ITT or WTT depending on you taste) is consistent, there is no term* t *such that* $\vdash_{\mathsf{x}} t : \Pi(A : \square_0).\ A$.

Proof. Assume such a t exists. We have $\vdash \Pi(A : \square_0).\ A$, so by Corollary 14.2.1 there exists $\bar{t}$ such that $\vdash \bar{t} : \Pi(A : \square_0).\ A$ which contradicts the assumed consistency of the target. □

14.3 Optimisations

Up until now, I remained silent about one thing: the size of the translated terms. Indeed, the translated term is a decoration of the initial one by transports which appear in many locations. For example, at each application we use a transport by Lemma 14.1.1 to ensure that the term in function position is given a function type. In most cases—in particular when translating ITT terms back to ITT terms—this produces unnecessary transports (often by reflexivity) that we wish to avoid.

In order to limit the size explosion, in the above we use a different version of transport, namely $\mathsf{transport}'$ such that

$$\begin{aligned}\mathsf{transport}'_{A_1,A_2}(p,t) &= t && \text{when } A_1 =_\alpha A_2\\ &= p_* t && \text{otherwise.}\end{aligned}$$

The idea is that we avoid *trivially* unnecessary transports (we do not deal with β-conversion for instance). We extend this technique to the different constructors of equality (symmetry, transitivity, …) so that they reduce to reflexivity whenever possible. Take transitivity for instance:

$$\begin{aligned} \mathsf{transitivity}'(\mathsf{refl}\ u, q) &= q \\ \mathsf{transitivity}'(p, \mathsf{refl}\ u) &= p \\ \mathsf{transitivity}'(p, q) &= \mathsf{transitivity}(p, q). \end{aligned}$$

We show these *defined terms* enjoy the same typing rules as their counterparts and use them instead. In practice it is enough to recover the exact same term when it is typed in ITT. In ITT, this can even be pushed a bit further to test some conversions, but in WTT it has to stay syntactical.

Note that these optimisations still make sense in WTT even though transport does not compute, we are just replacing a proof by another if you will.

Note that these optimisations serve no purpose at all to one interested only in the proof, it only becomes interesting when exploiting the constructiveness of the formalised proof to get translated terms and the corresponding derivations from a derivation in ETT.

14.4 Translation of axioms

After introducing them, I barely mentioned axioms $\mathsf{ax}(n)$. The idea is that when dealing with the translation of $\Gamma \vdash t : A$ which can contain some mentions to the global environment Σ, we assume we are given some $\overline{\Sigma} \in [\![\Sigma]\!]$ a global context of translations of the types of Σ. Then, $\mathsf{ax}(n)$ is simply translated to itself.

This is very similar to how we treat local contexts Γ. Once that we know how to translate a judgment, given a translation of the global context, it suffices to apply this to the types of Σ which all live in smaller global contexts to finally reach a point where the global context is empty and thus with a trivial translation. This is merely a formality.

In the target, the axioms need no longer be axioms usually because they can be given content, but I find it easier to replace them by some definitions after the translation is done rather than trying to do it all at once.

Reflection and homotopy | 15

Homotopy and reflection seem contradictory at first since UIP often comes with reflection[1] and is negated by univalence, but as I have exposed earlier, using 2-level Type Theories we can make two equalities—one with UIP and one which is homotopic—cohabit consistently.

1: UIP can be proved from reflection and the J eliminator of equality.

Refer to Chapter 6 (Flavours of type theory) for more on 2TTs.

Now, in Chapter 12 (Framework) I showed how the proof was (almost) agnostic with respect to universes, thanks to abstract universe constructors, one of which was (crucially) the sort of an identity type.

This is the key to instantiate our translation into one going from HTS to 2TT or 2-level Weak Type Theory (2WTT). We can recover those theories in our setting by taking F_i and U_i as respectively the fibrant and strict universes (for $i \in \mathbb{N}$), along with the following PTS rules:

$$\begin{array}{llll}
(\mathsf{F}_i, \mathsf{F}_{i+1}) & \in \mathsf{Ax} & (\mathsf{U}_i, \mathsf{U}_{i+1}) & \in \mathsf{Ax} \\
(\mathsf{F}_i, \mathsf{F}_j, \mathsf{F}_{\mathsf{max}(i,j)}) & \in \mathsf{R} & (\mathsf{F}_i, \mathsf{U}_j, \mathsf{U}_{\mathsf{max}(i,j)}) & \in \mathsf{R} \\
(\mathsf{U}_i, \mathsf{F}_j, \mathsf{U}_{\mathsf{max}(i,j)}) & \in \mathsf{R} & (\mathsf{U}_i, \mathsf{U}_j, \mathsf{U}_{\mathsf{max}(i,j)}) & \in \mathsf{R}
\end{array}$$

and the fact that the sort of the (strict) identity type on $A : s$ is the *strictified* version of s, *i.e.* U_i for $s = \mathsf{U}_i$ or $s = \mathsf{F}_i$. The fibrant equality can be recovered using axioms as exposed in Section 12.1 (Syntax(es)).

Of course, just extending the proof to two equalities is a possibility as well. As one can imagine it does not change the proof much.

Note that this is just one of many presentations of 2-level Type Theories: instead of being type-based, fibration is sometimes dealt with using a specific judgment and this is not covered by our formalisation.

Type-based approaches are usually much better suited to translations (because translations are type-preserving).

In short, the translation from HTS to 2TT or 2WTT is *exactly* the same as the one from ETT to ITT or WTT that I present in this thesis. This fact is factorised through our formalisation.

Formalisation of the translation 16

The formalisation is inspired from that of Coq in MetaCoq. It is actually defined *besides* MetaCoq to allow for some interoperability, bringing about fairly realistic examples. This provides evidence that the translation is constructive *and* computes! Note that we also rely on the Equations [20, 21] plugin to derive nice dependent induction principles.

[20]: Sozeau (2010), 'Equations: A Dependent Pattern-Matching Compiler'
[21]: Sozeau et al. (2019), 'Equations reloaded: high-level dependently-typed functional programming and proving in Coq'

Our formalisation takes full advantage of its easy interfacing with MetaCoq: we define two theories, namely ETT and ITT (or WTT), but the target features a lot of syntactic sugar by having things such as transport, heterogeneous equality and packing as part of the syntax. The operations regarding these constructors—in particular the tedious ones—are written in Coq and then quoted to finally be *realised* in a translation from ITT to MetaCoq. For WTT the story is a bit different because there is no *weak* Coq or MetaCoq so for now there is no translation to a purer WTT though this is work that we started but put on hold by coming to realise the target would not be that much simpler.

Some work for the future...

Interoperability with MetaCoq. The translation we define from ITT to MetaCoq is not proven correct, but it is not really important as it can just be seen as a feature to observe the produced terms in a nicer setting. In any case, MetaCoq does not yet provide a complete formalisation of CIC rules, as guard checking of recursive definitions and strict positivity of inductive type declarations are not formalised yet.

We will discuss about MetaCoq and its theory in greater detail in Part 'A verified type-checker for Coq, in Coq'.

Our formalised theorems however do not depend on MetaCoq itself and as such there is no need to *trust* the plugin or the formalisation of Coq inside it.

We also provide a translation from MetaCoq (and thus Coq!) to ETT that we will describe more extensively with the examples in Section 16.3 (ETT-flavoured Coq: examples).

16.1 Quick overview of the formalisation

The formalisation can be found at github.com/TheoWinterhalter/ett-to-itt and github.com/TheoWinterhalter/ett-to-itt/tree/weak for the translations from ETT to ITT and from ETT to WTT respectively. Let me describe quickly the formalisation in the case of ETT to ITT, the other formalisation is pretty similar.

The file SAst.v contains the definition of the (common) abstract syntax of ETT and ITT in the form of an inductive definition with de Bruijn indices for variables. Sorts are defined separately in Sorts.v and we will address them later in Section 16.2 (About universes).

In Chapter 12 (Framework) I showed two different syntaxes but to make things simpler I use a common type of terms, only some terms will only have a typing rule in ITT.

```
Inductive sterm : Type :=
| sRel (n : nat)
| sSort (s : sort)
| sProd (nx : name) (A B : sterm)
| sLambda (nx : name) (A B t : sterm)
| sApp (u : sterm) (nx : name) (A B v : sterm)
| sEq (A u v : sterm)
| sRefl (A u : sterm)
| (* ... *) .
```

The files ITyping.v and XTyping.v define respectively the typing judgments for ITT and ETT. The first is defined with first a notion of reduction from which is deduced conversion, while the latter has typed conversion and typing defined using mutual inductive types. Then, most of the files are focused on the meta-theory of ITT and can be ignored by readers who do not need to see yet another proof of subject reduction.

In the formalisation of the translation to WTT, these meta-theory files are kept to a minimum!

The most interesting files are obviously those where the fundamental lemma and the translation are formalised: FundamentalLemma.v and Translation.v. For instance, here is the main theorem, as stated in our formalisation:

```
Theorem complete_translation {Σ} :
  type_glob Σ ->
  (forall {Γ t A} (h : Σ ;;; Γ |-x t : A)
     {Γ'} (hΓ : Σ |--i Γ' ∈ ⟦ Γ ⟧),
      Σ A' t', Σ ;;;; Γ' ⊢ [t'] : A' ∈ ⟦ Γ ⊢ [t] : A ⟧) *
  (forall {Γ u v A} (h : Σ ;;; Γ |-x u ≡ v : A)
     {Γ'} (hΓ : Σ |--i Γ' ∈ ⟦ Γ ⟧),
      Σ A' A'' u' v' p',
        eqtrans Σ Γ A u v Γ' A' A'' u' v' p').
```

Herein `type_glob Σ` refers to the fact that the global context is well-typed—thing which we mainly ignored in the paper translation. The fact that the theorem holds in Coq ensures we can actually compute a translated term and type out of a derivation in ETT.

16.2 About universes

As I mentioned earlier in Chapter 12 (Framework), sorts are treated abstractly in the translation. In the formalisation sorts are defined using the following class:

```
Class Sorts.notion := {
  sort : Type ;
  succ : sort -> sort ;
  prod_sort : sort -> sort -> sort ;
  sum_sort : sort -> sort -> sort ;
  eq_sort : sort -> sort ;
  eq_dec : forall s z : sort, {s = z} + {s <> z} ;
  succ_inj : forall s z, succ s = succ z -> s = z
}.
```

From the notion of sorts, we require functions to get the sort of a sort, the sort of a product from the sorts of its arguments, and the sort of an identity type. We also require some measure of decidable equality and injectivity on those. From this we ensure the PTS is *functional* (in particular without cumulativity) so that we have unique typing.

We are using classes here because they come with some automation in Coq which allows us to assume globally a notion of sort without having to specify it at each use.

This allows us to instantiate this by a lot of different notions like a natural number based hierarchy of Type_i and even an extension of it with a universe Prop of propositions as in CIC. We also provide an instance corresponding to 2TT as explained in Chapter 15 (Reflection and homotopy).

In order to deal with examples in a simpler manner (making up for our lack of universe polymorphism and cumulativity) by interacting with Coq (thanks to MetaCoq), one of the instances we provide comes with only one universe Type and the inconsistent typing rule Type : Type.

For inconsistency of Type *in* Type, see Subsection 'Universes in Coq'.

16.3 ETT-flavoured **Coq**: examples

In this section I demonstrate how our translation can bring extensionality to the world of Coq in action. The examples can be found in plugin_demo.v. Again, since we do not have *weak* Coq or MetaCoq, these are in the case of the translation to ITT only.

First, a pedestrian approach. I would like to begin by showing how one can write an example step by step before we show how it can be instrumented and automated as a plugin. For this I use a self-contained example without any inductive types or recursion, illustrating a very simple case of reflection. The term we want to translate is the identity coercion:

$$\lambda\ A\ B\ e\ x.\ x : \Pi\ A\ B.\ A = B \to A \to B$$

which relies on the equality $e : A = B$ and reflection to convert $x : A$ to $x : B$. Of course, this definition is not accepted in Coq because this conversion is not valid in ITT.

```
Fail Definition pseudoid (A B : Type) (e : A = B) (x : A) : B
  := x.
```

However, we still want to be able to write it *in some way*, in order to avoid manipulating de Bruijn indices directly. For this, we use a little trick by first defining a Coq axiom to represent an ill-typed term:

When I say ill-typed term, I mean a subterm that is typed but of the wrong type.

```
Axiom candidate : forall A B (t : A), B.
```

`candidate A B t` is a candidate `t` of type `A` to inhabit type `B` in the fashion of OCaml's `Obj.magic`. We complete this by adding a notation that is reminiscent to Agda's hole mechanism.

```
Notation "'{!' t '!}'" := (candidate _ _ t).
```

We can now write the ETT function within Coq.

```
Definition pseudoid (A B : Type) (e : A = B) (x : A) : B :=
  {! x !}.
```

We can then quote the term and its type to MetaCoq thanks to the `Quote Definition` command provided by the plugin.

```
Quote Definition pseudoid_term :=
  ltac:(let t := eval compute in pseudoid in exact t).
Quote Definition pseudoid_type :=
  ltac:(let T := type of pseudoid in exact T).
```

The syntax is a bit heavy because of a call to LTac used to quote the normal form of the term.

The terms that we get are now MetaCoq terms, representing Coq syntax. We need to put them in ETT, meaning adding the annotations, and also removing the `candidate` axiom. This is the purpose of the `fullquote` function that we provide in our formalisation.

`fullquote` is given a *big* number corresponding to the the number of recursive calls it is allowed to do. This *fuel* technique allows us to circumvent the termination checker of Coq.

```
Definition pretm_pseudoid :=
  Eval lazy in
  fullquote (2^18) Σ [] pseudoid_term empty empty nomap.

Definition tm_pseudoid :=
  Eval lazy in
  match pretm_pseudoid with
  | Success t => t
  | Error _ => sRel 0
  end.


Definition prety_pseudoid :=
  Eval lazy in
  fullquote (2^18) Σ [] pseudoid_type empty empty nomap.

Definition ty_pseudoid :=
  Eval lazy in
  match prety_pseudoid with
  | Success t => t
  | Error _ => sRel 0
  end.
```

`tm_pseudoid` and `ty_pseudoid` correspond respectively to the ETT representation of `pseudoid` and its type. We then produce, using our home-brewed LTac type-checking tactic, the corresponding ETT typing derivation.

```
Lemma type_pseudoid : Σi ;;; [] |-x tm_pseudoid : ty_pseudoid.
Proof.
  unfold tm_pseudoid, ty_pseudoid.
  ettcheck. cbn.
  eapply reflection with (e := sRel 1).
  ettcheck.
Defined.
```

Notice the use of `reflection` which is a constructor of the inductive type of typing derivations in ETT. Its type is given by

```
reflection :
  forall A u v e,
    Σ ;;; Γ |-x e : sEq A u v ->
    Σ ;;; Γ |-x u ≡ v : A
```

and corresponds to a use of the reflection rule. We can then translate this derivation, obtain the translated term and then convert it to MetaCoq.

```
Definition itt_pseudoid : sterm :=
  Eval lazy in
  let '(_ ; t ; _) :=
    type_translation type_pseudoid istrans_nil
  in t.

Definition tc_pseudoid : tsl_result term :=
  Eval lazy in
  tsl_rec (2 ^ 18) Σ [] itt_pseudoid empty.
```

Once we have it, we *unquote* the term to obtain a Coq term (notice that the only use of reflection has been replaced by a transport).

```
fun (A B : Type) (e : A = B) (x : A) => transport e x
     : forall A B : Type, A = B -> A -> B
```

Making a Plugin with MetaCoq. All of this work is pretty systematic. Fortunately for us, MetaCoq also features a monad to reify Coq commands which we can use to *program* the translation steps. As such we have written a complete procedure, relying on type-checkers we wrote for ITT and ETT, which can generate equality obligations.

Thanks to this, the user does not have to know about the details of implementation of the translation, and can stay within the Coq ecosystem.

For instance, our previous example now becomes:

```
Definition pseudoid (A B : Type) (e : A = B) (x : A) : B :=
  {! x !}.

Run TemplateProgram (Translate ε "pseudoid").
```

ε is the empty translation context, see the next example to understand the need for a translation context

This produces a Coq term `pseudoid'` corresponding to the translation. Notice how the user does not even have to provide any proof of equality or derivations of any sort. The derivation part is handled by our own type-checker while the obligation part is solved automatically by the Coq obligation mechanism.

About inductive types. As we promised, our translation is able to handle inductive types. For this consider the inductive type of vectors (or length-indexed lists) below, together with a simple definition (we will remain in ITT for simplicity).

```
Inductive vec A : nat -> Type :=
| vnil : vec A 0
| vcons : A -> forall n, vec A n -> vec A (S n).

Arguments vnil {_}.
Arguments vcons {_} _ _ _.

Definition vv := vcons 1 _ vnil.
```

This time, in order to apply the translation we need to extend the translation context with `nat` and `vec`.

```
Run TemplateProgram (
  Θ <- TranslateConstant ε "nat" ;;
  Θ <- TranslateConstant Θ "vec" ;;
  Translate Θ "vv"
).
```

The `_ <- _ ;; _` notation corresponds to a monadic bind.

The command `TranslateConstant` enriches the current translation context with the types of the inductive type and of its constructors. The translation context then also contains associative tables between our own representation of constants and those of Coq. Unsurprisingly[1], the translated Coq term is the same as the original term.

1: It is thanks to all the effort that has gone into optimising the translation

Reversal of vectors. Next, we tackle a motivating example: reversal on vectors. Indeed, implementing this operation the same way it can be done on lists ends up in the following conversion problem:

```
Fail Definition vrev {A n m} (v : vec A n) (acc : vec A m)
: vec A (n + m) :=
  vec_rect
    A (fun n _ => forall m, vec A m -> vec A (n + m))
    (fun m acc => acc)
    (fun a n _ rv m acc => rv _ (vcons a m acc))
    n v m acc.
```

The recursive call returns a vector of length `n + S m` where the context expects one of length `S n + m`. In ITT, these types are not convertible. This example is thus a perfect fit for ETT where we can use the fact that these two expressions always compute to the same thing when instantiated with concrete numbers.

```
Definition vrev {A n m} (v : vec A n) (acc : vec A m)
: vec A (n + m) :=
  vec_rect
    A (fun n _ => forall m, vec A m -> vec A (n + m))
    (fun m acc => acc)
    (fun a n _ rv m acc => {! rv _ (vcons a m acc) !})
    n v m acc.

Run TemplateProgram (
  Θ <- TranslateConstant ε "nat" ;;
  Θ <- TranslateConstant Θ "vec" ;;
  Θ <- TranslateConstant Θ "Nat.add" ;;
  Θ <- TranslateConstant Θ "vec_rect" ;;
  Translate Θ "vrev"
).
```

This generates four obligations that are all solved automatically. One of them contains a proof of `S n + m = n + S m` while the remaining three correspond to the computation rules of addition (as mentioned before, `add` is simply a constant and does not compute in our representation, hence the need for equalities). The returned term is the following, with only one transport remaining (remember our interpretation map removes unnecessary transports).

```
fun (A : Type) (n m : nat) (v : vec A n) (acc : vec A m) =>
vec_rect A
  (fun n _ => forall m, vec A m -> vec A (n + m))
  (fun m acc => acc)
  (fun a n₀ v₀ rv m₀ acc₀ =>
    transport
      (vrev_obligation_3 A n m v acc a n₀ v₀ rv m₀ acc₀)
      (rv (S m₀) (vcons a m₀ acc₀))
  ) n v m acc
: forall A n m, vec A n -> vec A m -> vec A (n + m)
```

16.4 Towards an interface between **Andromeda 1** and **Coq**

Andromeda 1 [34] is a proof assistant implementing ETT in a sense that is really close to our formalisation. Aside from a concise nucleus consisting in a very basic type theory, Andromeda 1 features an interface in which the user can declare constants with given types. Definitions

[34]: Bauer et al. (2016), *The 'Andromeda' prover*

and computational behaviour are defined using constants which inhabit equalities.

Here is for instance the definition of natural numbers and their eliminator with computation rules.

```
constant nat : Type
constant O : nat
constant S : nat -> nat

constant natrec :
  ∏ (P : nat -> Type),
    P O ->
    (∏ (n : nat), P n -> P (S n)) ->
    ∏ (n : nat), P n.

constant natrec_O :
  ∏ P Pz Ps, natrec P Pz Ps O ≡ Pz.

constant natrec_S :
  ∏ P Pz Ps n, natrec P Pz Ps (S n) ≡ Ps n (natrec P Pz Ps n).
```

This is essentially what we do in our formalisation. Furthermore, their theory relies on Type : Type, meaning, our modular handling of universes can accommodate for this as well.

All in all, it should be possible in the future to use our translation (or a similar one) to produce Coq terms out of Andromeda 1 developments. Andromeda 1 would be generating the typing derivations for us, which is particularly interesting because Andromeda 1's system is much more practical than the small type-checker I wrote in Coq.

16.5 Composition with other translations

This translation also enables the formalisation of translations that target ETT rather than ITT and still get mechanised proofs of (relative) consistency by composition with this ETT to ITT translation. This could also be used to implement plugins based on the composition of translations. In particular, supposing we have a theory which forms a subset of ETT and whose conversion is decidable. Using this translation, we could formalise it as an embedded domain-specific type theory and provide an automatic translation of well-typed terms into witnesses in Coq. This would make it possible to extend conversion with the theory of lists for example.

The ability to chain translations would provide a simple way to justify the consistency of CoqMT [79] for example, seeing it as an extensional type theory where reflection is restricted to equalities on a specific domain whose theory is decidable.

[79]: Jouannaud et al. (2017), 'Coq without Type Casts: A Complete Proof of Coq Modulo Theory'

Limitations and related works 17

17.1 Limitations and axioms

Currently, the representation of terms and derivations and the computational content of the proof only allow us to deal with the translation of relatively small terms but I hope to improve this in the future. As we have seen, the actual translation involves the computational content of lemmata of inversion, substitution, weakening and equational reasoning and thus cannot be presented as a simple recursive definition on derivations.

I would like to use a minimal representation of derivations that does not contain everything that can be inferred automatically for one.

As I already mentioned, the axioms K and funext are both necessary in ITT if we want the translation to be conservative as they are provable in ETT [78]. However, one might still be concerned about having axioms as they can for instance hinder canonicity of the system. In that respect, K is not really a restriction since it preserves canonicity, when taken as a definition rather than as an axiom. The best proof of that is probably Agda itself which natively features K—in fact, one needs to explicitly deactivate it with a flag if one wishes to work without. In Agda one writes K as follows:

[78]: Hofmann (1995), 'Conservativity of equality reflection over intensional type theory'

```
K : {A : Set} {x : A} (p : x ≡ x) → p ≡ refl
K refl = refl
```

The case of funext is trickier. It should be possible to realise the axiom by composing our translation with a setoid interpretation [80] which validates it, or by going into a system featuring it, for instance by implementing Observational Type Theory [26] like EPIGRAM [81].

[80]: Altenkirch (1999), 'Extensional equality in intensional type theory'

[26]: Altenkirch et al. (2007), 'Observational equality, now!'

[81]: McBride (2004), 'Epigram: Practical programming with dependent types'

However, these two axioms are not used to define the translation itself, but only to witness UIP and functional extensionality in the translation to Coq. The translation itself only relies on one axiom, called `conv_trans_AXIOM`, stating that conversion of ITT is transitive. The translation to WTT is totally axiom-free however!

The proof of transitivity in ITT basically sums up to the confluence of the reduction rules which is out of scope for this thesis and has recently been formalised in Agda [82] (in a simpler setting with only one universe). MetaCoq also features a proof of confluence for Coq [4]. Regardless, this axiom inhabits a proposition (the type of conversion is in `Prop`) and is thus irrelevant for computation. Actually no information about the derivation leaks to the production of the ITT term.

[82]: Abel et al. (2017), 'Decidability of Conversion for Type Theory in Type Theory'

[4]: Sozeau et al. (2020), 'Coq Coq correct! verification of type checking and erasure for Coq, in Coq'

On a different note, the `candidate` axiom allows us to derive `False` but is merely used to write ill-typed terms in Coq. The translated term will never make us of it and one can always check if a term is relying on unsafe assumptions thanks to the `Print Assumptions` command.

Finally, it is important to note that, while this work does provide a way to translate from ITT to WTT, one must still assume UIP, funext and several equality axioms even though they are not necessary in ITT itself.

17.2 Related works

The seminal works on the precise connection between ETT and ITT go back to [83] and [78, 84]. In particular, the work of Hofmann provides a categorical answer to the question of consistency and conservativity of ETT over ITT with UIP and funext. Ten years later, Oury [76, 85] provided a translation from ETT to ITT with UIP and funext and other axioms (mainly due to technical difficulties). Although a first step towards a move from categorical semantics to a syntactic translation, his work does not stress any constructive aspect of the proof and shows that there merely exist translations in ITT of a typed term in ETT.

[83]: Streicher (1993), *Investigations into intensional type theory*
[78]: Hofmann (1995), 'Conservativity of equality reflection over intensional type theory'
[84]: Hofmann (1997), *Extensional constructs in intensional type theory*
[76]: Oury (2005), 'Extensionality in the calculus of constructions'
[85]: Oury (2006), 'Egalité et filtrage avec types dépendants dans le calcul des constructions inductives'

Doorn *et al.* [33] have later proposed and formalised a similar translation between a PTS with and without explicit conversion. This does not entail anything about ETT to ITT but we can find similarities in that there is a witness of conversion between any term and itself under an explicit conversion, which internalises irrelevance of explicit conversions. This morally corresponds to a *Uniqueness of Conversions* principle.

[33]: Doorn et al. (2013), 'Explicit convertibility proofs in pure type systems'

The Program [86] extension of Coq performs a related coercion insertion algorithm, between objects in subsets on the same carrier or in different instances of the same inductive family, assuming a proof-irrelevance axiom. Inserting coercions locally is not as general as the present translation from ETT to ITT which can insert transports in any context.

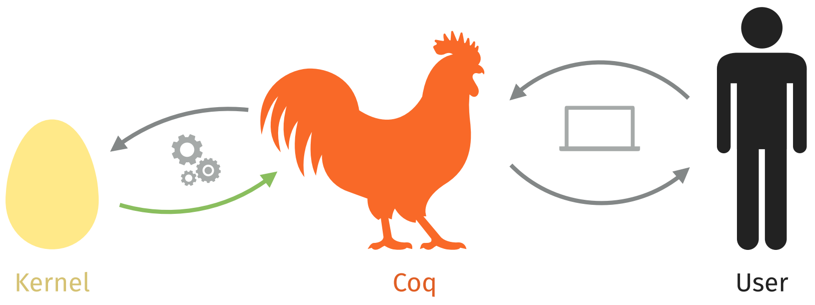
[86]: Sozeau (2007), 'Program-ing Finger Trees in Coq'

In both formalisations, I implemented type-checkers for the theories involved. In WTT, since the meta-theory is so simple, I was able to prove the type-checker to be sound and complete. I do not detail them in this thesis as I will dedicate Part 'A verified type-checker for Coq, in Coq' to the implementation and verification of a full-fledged type-checker of Coq in Coq.

A verified type-checker for Coq, in Coq

Overview 18

Coq is built around a well-delimited kernel that contains a representation of terms and performs type-checking for PCUIC. This kernel makes up a so-called Trusted Code Base: its OCaml implementation has to be *trusted* by the user to *trust* in Coq. This paradigm allows several *unsafe* features to be implemented outside of this kernel so that they do not have to be trusted. Tactics for instance will produce terms, but in the end, Coq's kernel will type-check those and will complain if the tactics produced ill-typed terms. This means the implementation of the tactic language and other *surface*-features are not critical. It is better if they do their job properly, but there is a safety net in case they do not.



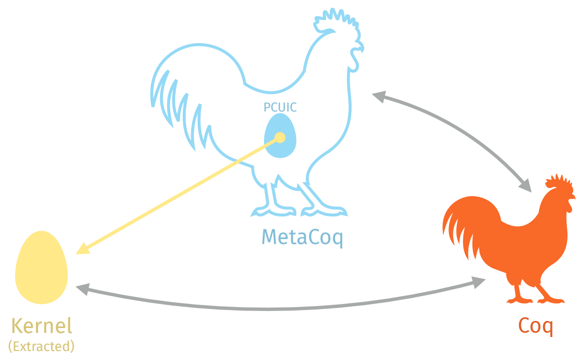
The green arrow is the only *trusted* one. The others can fail.

In this setting the kernel has to be kept as small as possible for inspection by humans. In particular experimental features are dangerous in there. Unfortunately, Coq's kernel is not free of mistakes, and one critical bug has been found roughly every year for the last two decades. Even then, the kernel presentation offers some damage control as it usually requires minor changes to fix the problem. In fact, those bugs are always related to the implementation rather than being consistency issues of PCUIC.

The interested reader can have a look at `https://github.com/coq/coq/blob/master/dev/doc/critical-bugs` where a list of critical bugs of Coq is maintained.

PCUIC on the other hand seems more worthy of trust. The idea of our project [4] represents a paradigm shift: we propose to replace the TCB by a Trusted Theory Base (TTB).

[4]: Sozeau et al. (2020), 'Coq Coq correct! verification of type checking and erasure for Coq, in Coq'



This time the idea is to rely on MetaCoq which contains a specification of PCUIC to implement a verified kernel, assuming that the theory is correct (hence the TTB). This TTB is also kept to a minimum, it only consists in assuming strong normalisation. The new kernel can be extracted to Coq using its extraction mechanism [87] and it no longer has to be trusted. Of course this requires trusting the extraction mechanism and even Coq itself in which PCUIC is formalised. The first point is mitigated by a verified extraction that is also part of the project. The latter is more fundamental but I would like to argue that this is *not* entirely detrimental to our goal: the verified implementation is still more detailed than the OCaml implementation, in particular all the invariants are made explicit; implementation and specification are now clearly separated.

[87]: Letouzey (2008), 'Coq Extraction, an Overview'

Note that we still have to rely on a TTB because of Gödel's incompleteness theorems which prevent us from proving consistency of Coq within Coq. This implies that we have to admit strong normalisation of the system, as it would lead us to canonicity, hence consistency. Thus, the assumption that the meta-theory enjoys the well-known assumed properties is the main shortcoming of our formalisation. If those did not hold, it would either raise a serious issue in Coq's theory, or a problem in the specification we made of it. For this last reason, the relatively small specification we have is welcome.

See Chapter 2 (Proof theory) for more on Gödel's incompleteness theorems.

See Chapter 7 (Desirable properties of type theories).

Besides that, the theory of PCUIC we specify and implement is a bit simpler than that of Coq: we use lists where arrays are involved and more importantly we do not feature modules and template polymorphism. The implementation also provides efficient universe constraint checking and guard condition checking, we chose to remain abstract over those and simply assume the existence of those in the specification: any implementation satisfying the specification can thus be plugged in. We use a rather naive universe checker for now and do not provide any guard checking at all. Finally, we do not handle η-equalities yet as I will explain in the next chapter where I present our specification of PCUIC.

See Chapter 6 (Flavours of type theory).

A specification of Coq 19

In Section 6.2 (Focus on the theory behind Coq) I already talked about the type theory of Coq we call PCUIC while in Chapter 9 (Syntax and formalisation of type theory) you caught a glimpse of its representation inside Coq itself. In this chapter I will recall a few things focusing on the points that were not discussed in those chapters like representation of fixed-points and inductive types, as well as typing.

All this is the subject of [4].

[4]: Sozeau et al. (2020), 'Coq Coq correct! verification of type checking and erasure for Coq, in Coq'

19.1 Syntax of PCUIC

In MetaCoq, the syntax of PCUIC is defined as the following inductive type.

```
Inductive term :=
| tRel (n : nat)
| tSort (u : Universe.t)
| tProd (na : name) (A B : term)
| tLambda (na : name) (A t : term)
| tLetIn (na : name) (b B t : term)
| tApp (u v : term)
| tConst (k : kername) (ui : Instance.t)
| tInd (ind : inductive) (ui : Instance.t)
| tConstruct (ind : inductive) (n : nat) (ui : Instance.t)
| tCase
    (indn : inductive * nat)
    (p c : term)
    (brs : list (nat * term))
| tProj (p : projection) (c : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat).
```

I will not present again variables, Π-types, etc. and will focus on the other constructors, for this I need to introduce the global environment.

Global environment. Just as for the translation presented in Part 'Elimination of Reflection', we have a global environment Σ containing the axioms, definitions and declarations of inductive types. Instead of using de Bruijn indices we use strings to refer to these declarations. In fact, `kername` is simply a notation for `string` which we use to lift ambiguity as to its purpose.

Universes. Besides the global environment we have a set of universes with their constraints. Indeed, although we usually present the hierarchy of universes in Coq using Type_i with $i \in \mathbb{N}$, the implementation is a bit different in that universes are not fixed but floating.

```
Type@{i} : Type@{j}
```

In Coq, Type_i is written `Type@{i}`.

holds in Coq as long as the constraint $i < j$ is satisfied but neither i nor j correspond to natural numbers. Satisfying the constraints means that there exists a valuation in the natural numbers preserving the constraints. We also have local universes and constraints for polymorphic definitions; universe instances (of type `Instance.t`) are there to substitute those for other universes. Aside from `Type`, we also represent `Set` and `Prop`. We do not deal with `SProp` yet.

Constants. `tConst k ui` represents a definition or axiom declared in the environment under name `k`. Constant declarations are inhabitants of the following record type.

```
Record constant_body := {
  cst_type : term ;
  cst_body : option term ;
  cst_universes : universes_decl
}.
```

A constant comes with its type and an optional definition. Constants without definitions are simply axioms. The `universes_decl` is there in case the constant happens to be universe polymorphic.

Inductive types. `tInd ind ui` represents an inductive type defined in the global context. The type `inductive` is

```
Record inductive : Set := mkInd {
  inductive_mind : kername ;
  inductive_ind : nat
}.
```

It is not just a `kername` because inductive types can be mutually defined: the kername points to the mutual block, `inductive_ind` points to one inductive in the block. Note that this term refers to the inductive type, without its parameters or indices. For vectors this would be

```
vec : Type -> nat -> Type
```

In the global environment an inductive declaration provides the following information.

```
Record mutual_inductive_body := {
  ind_finite    : recursivity_kind ;
  ind_npars     : nat ;
  ind_params    : context ;
  ind_bodies    : list one_inductive_body ;
  ind_universes : universes_decl ;
  ind_variance  : option (list Universes.Variance.t)
}.
```

These informations include the number of parameters, the type of these parameters in the form of a context and the universe declarations. `ind_variance` deals with variance of universes in the case of cumulative inductive types. The `recursivity_kind` determines what kind of inductive type we are building, this in fact includes the usual inductive types, but also record types and coinductive types. Finally, `ind_bodies` is a list of declarations for each of the inductive types in the mutual block.

```
Record one_inductive_body := {
  ind_name  : ident ;
  ind_type  : term ;
  ind_kelim : sort_family ;
  ind_ctors : list (ident * term * nat) ;
  ind_projs : list (ident * term)
}.
```

Each of those comes with its name and type. `ind_kelim` states in which universes the definition can be eliminated (in most cases inductive types living in `Prop` cannot be scrutinised to build terms in `Type`). Then we either have a list of constructors for (co)inductive types, or a list of projections for (negative) records.

Constructors. `tConstruct ind n ui` comes with `ind` to point to the right inductive type while `n` tells us which constructor it is. For instance it would be `0` for `true` and `1` for `false`. Each constructor is given by a term of type

```
ident * term * nat
```

as we saw in the previous paragraph. It consists of a name, a type and an arity.

Pattern-matching. `tCase (ind, n) p c brs` is the representation of the pattern-matching on `c` of inductive type `ind` with `n` parameters, return predicate `p` and branches `brs`. There is one branch for each constructor and it contains the arity of the constructor and the term corresponding to the branch.

Projections. `tProj p c` represents the projection `p` applied to term `c`.

```
Definition projection :=
  inductive * nat * nat.
```

A projection is described by the inductive (or rather record) to which it belongs, the number of parameters of the record, and the index of the projected argument. In the case of dependent pairs, the latter will be `0` for the first projection and `1` for the second.

Fixed-points. Finally we have `tFix mfix idx` representing fixed-points and `tCoFix mfix idx` for cofixed-points. Like inductive types, fixed-points can be defined mutually, as such `mfix` is a list of (mutual) definitions:

```
Definition mfixpoint term :=
  list (def term).
```

while `idx` tells which one we are referring to. Definitions are given as

```
Record def term := mkdef {
  dname : name ;
  dtype : term ;
  dbody : term ;
  rarg  : nat
}.
```

that is with a name, a type and a body (the definition itself). The `rarg` field points to the recursive argument of the fixed-point. For instance in

```
Fixpoint map {A B : Type} (f : A -> B) (l : list A) {struct l}
  : list B :=
  match l with
  | [] => []
  | x :: l => f x :: map f l
  end.
```

the `{struct l}` means that `l` is the recursive argument. That is all recursive calls must be structurally decreasing on `l`. It is part of the syntax because Coq uses it as a syntactic guard to avoid unfolding fixed-points indefinitely: only when the recursive argument is a constructor can the fixed-point be unfolded.

Local environments. Local environments or contexts are lists of declarations we write in *snoc* order:

- `[]` is the empty context;
- `Γ ,, vass na A` extends Γ with an *assumption* variable of type A, named na;
- `Γ ,, vdef na a A` extends Γ with a *local definition* a of type A, named na.

Now that the syntax is out of the way we can move to the semantics.

19.2 Semantics of PCUIC

There are two important properties defined on the syntax: reduction and typing. The former being necessary for the latter as it is the base for conversion.

Reduction

We define reduction as a relation on terms, however, unlike for STL or many systems, because of constants and let-bindings which can be unfolded to their definitions, the relation has to mention the global and local environment. Note also that the relation is not functional, that is that the reduction we define is not deterministic.

The inductive type of 1-step reduction is

```
Inductive red1 (Σ : global_env) (Γ : context) :
  term -> term -> Type
```

I will give the different constructors one by one, while explaining them, focusing on the computation rules.

β-reduction.

```
red_beta na t b a :
  red1 Σ Γ (tApp (tLambda na t b) a) (subst10 a b)
```

This is the usual β-reduction rule, `subst10 a b` is just the term `b` where the 0-th variable is substituted by `a`.

ζ-reduction.

```
red_zeta na b B t :
  red1 Σ Γ (tLetIn na b B t) (subst10 b t)
```

This rule is similar to β-reduction, it states that the expression

```
let x : B := b in t x
```

reduces to `t b`.

Local definition unfolding.

```
red_rel i body :
  option_map decl_body (nth_error Γ i) = Some (Some body) ->
  red1 Σ Γ (tRel i) (lift0 (S i) body)
```

If the context contains some local definition $x := t$ then the variable x *reduces* to t. Because the definition makes sense in a smaller context, it needs to be weakened, hence the `lift`.

ι-reduction.

```
red_iota ind pars c u args p brs :
  red1 Σ Γ
    (tCase (ind, pars) p (mkApps (tConstruct ind c u) args) brs)
    (iota_red pars c args brs)
```

`mkApps` is a shortcut to apply a term to a list of arguments.

Pattern-matching reduces when the term under scrutiny is a constructor (potentially applied to some arguments). `iota_red` is defined as follows

```
Definition iota_red npar c args brs :=
  mkApps
    (snd (List.nth c brs (0, tDummy)))
    (List.skipn npar args).
```

`List.nth` takes a default value, so `(0, tDummy)` can be safely ignored. `iota_red` basically picks the branch corresponding to the constructor giving it the arguments of the constructor that are not also parameters of the inductive type: `List.skipn npar args`.

For instance for `@cons A x l`, which is a long way of writing `x :: l`, we only provide `x` and `l` to the branch and not `A` because it is a parameter.

Fixed-point unfolding.

```
red_fix mfix idx args narg fn :
  unfold_fix mfix idx = Some (narg, fn) ->
  is_constructor narg args = true ->
  red1 Σ Γ (mkApps (tFix mfix idx) args) (mkApps fn args)
```

Here we can witness the syntactic guard I talked about when describing fixed-points: when a fixed-point is applied to arguments such that its recursive argument is an applied constructor, it can safely be unfolded. `unfold_fix mfix idx` returns the recursive argument `narg` and the body of the fixed-point `fn`.

Cofixed-point unfolding. There are two rules to deal with unfolding of cofixed-points: one when it is pattern-matched, the other when it is projected.

```
red_cofix_case ip p mfix idx args narg fn brs :
  unfold_cofix mfix idx = Some (narg, fn) ->
  red1 Σ Γ
    (tCase ip p (mkApps (tCoFix mfix idx) args) brs)
    (tCase ip p (mkApps fn args) brs)

red_cofix_proj p mfix idx args narg fn :
  unfold_cofix mfix idx = Some (narg, fn) ->
  red1 Σ Γ
    (tProj p (mkApps (tCoFix mfix idx) args))
    (tProj p (mkApps fn args))
```

The are very similar to fixed-points except this time there is no syntactic guard with respect to a recursive argument.

δ-reduction.

```
red_delta c decl body (isdecl : declared_constant Σ c decl) u :
  decl.(cst_body) = Some body ->
  red1 Σ Γ (tConst c u) (subst_instance_constr u body)
```

A constant can reduce to its definition in the global environment (if it has one, *i.e.* if it is not an axiom). Since the definition is potentially universe polymorphic, we instantiate its universes with the one the constant was used with.

Projection.

```
red_proj i pars narg args k u arg:
  nth_error args (pars + narg) = Some arg ->
  red1 Σ Γ
    (tProj (i, pars, narg) (mkApps (tConstruct i k u) args))
    arg
```

When a constructor of a record (that is the record is given with its fields) is projected, it reduces to the corresponding field.

Congruence rules. All remaining rules are congruence rules we have for instance the two congruence rules for `tLambda`:

```
| abs_red_l na M M' N :
    red1 Σ Γ M M' ->
    red1 Σ Γ (tLambda na M N) (tLambda na M' N)

| abs_red_r na M M' N :
    red1 Σ (Γ ,, vass na N) M M' ->
    red1 Σ Γ (tLambda na N M) (tLambda na N M')
```

They state that you can reduce either subterm.

Conversion

Unlike Agda, conversion in Coq is not typed but rather based primarily on reduction. Its definition is the following, corresponding basically to the reflexive, symmetric and transitive closure of reduction:

```
Inductive conv Σ Γ : term -> term -> Type :=
| conv_refl t u :
    eq_term (global_ext_constraints Σ) t u ->
    Σ ;;; Γ |- t = u

| conv_red_l t u v :
    red1 Σ Γ t v ->
    Σ ;;; Γ |- v = u ->
    Σ ;;; Γ |- t = u

| conv_red_r t u v :
    Σ ;;; Γ |- t = v ->
    red1 (fst Σ) Γ u v ->
    Σ ;;; Γ |- t = u

where ” Σ ;;; Γ |- t = u ” := (@conv _ Σ Γ t u) : type_scope.
```

Here Σ is a pair comprised of a global environment and the universe declarations.

In terms of inference rules this would be written

$$\frac{t =_\alpha u}{\Sigma;\Gamma \vdash t \equiv u} \qquad \frac{\Sigma;\Gamma \vdash t \to v \qquad \Sigma;\Gamma \vdash v \equiv u}{\Sigma;\Gamma \vdash t \equiv u}$$

$$\frac{\Sigma;\Gamma \vdash t \equiv v \qquad \Sigma;\Gamma \vdash u \to v}{\Sigma;\Gamma \vdash t \equiv u}$$

This is similar except that we do not merely use α-conversion in the reflexive case, we also equate the terms up to universes. Indeed sometimes, two universes may be syntactically different but still be the same like i and j with constraints $i \leq j$ and $j \leq i$.

This becomes more apparent in the cumulativity definition.

```
Inductive cumul Σ Γ : term -> term -> Type :=
| cumul_refl t u :
    leq_term (global_ext_constraints Σ) t u ->
    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) : type_scope.
```

$$\frac{t \preceq_\alpha u}{\Sigma;\Gamma \vdash t \preceq u} \qquad \frac{\Sigma;\Gamma \vdash t \to v \qquad \Sigma;\Gamma \vdash v \preceq u}{\Sigma;\Gamma \vdash t \preceq u}$$

$$\frac{\Sigma;\Gamma \vdash t \preceq v \qquad \Sigma;\Gamma \vdash u \to v}{\Sigma;\Gamma \vdash t \preceq u}$$

Of course this time it is not symmetric. We break symmetry by using `leq_term` to perform α-equality up to cumulativity of universes. In a sense this is what does all the work regarding cumulativity.

We define `eq_term` and `leq_term` at the same time using a more general definition.

```
Definition eq_term φ :=
  eq_term_upto_univ (eq_universe φ) (eq_universe φ).

Definition leq_term φ :=
  eq_term_upto_univ (eq_universe φ) (leq_universe φ).
```

φ represents the universe constraints. `eq_universe` and `leq_universe` are relations on universes with self-explanatory names.

The definition of `eq_term_upto_univ` is rather long so I will give an excerpt of it.

```
Inductive eq_term_upto_univ
  (Re Rle : Universe.t -> Universe.t -> Prop)
  : term -> term -> Type :=

| eq_Rel n  :
    eq_term_upto_univ Re Rle (tRel n) (tRel n)

| eq_Sort s s' :
    Rle s s' ->
    eq_term_upto_univ Re Rle (tSort s) (tSort s')

| eq_App t t' u u' :
    eq_term_upto_univ Re Rle t t' ->
    eq_term_upto_univ Re Re u u' ->
    eq_term_upto_univ Re Rle (tApp t u) (tApp t' u')

| eq_Const c u u' :
    R_universe_instance Re u u' ->
    eq_term_upto_univ Re Rle (tConst c u) (tConst c u')

| eq_Prod na na' a a' b b' :
    eq_term_upto_univ Re Re a a' ->
    eq_term_upto_univ Re Rle b b' ->
    eq_term_upto_univ Re Rle (tProd na a b) (tProd na' a' b')

(* ... *)
.
```

As you can see, when comparing sorts we simply compare the universes with `Rle`. Otherwise, we simply compare the subterms with the same relations. There is the exception of domains however, indeed they are in contravariant positions and as such they should be compared in reverse order. In Coq we simply use conversion rather that cumulativity in contravariant positions:

$$\frac{A =_\alpha A' \qquad B \leq_\alpha B'}{\Pi(x : A).B \leq_\alpha \Pi(x : A').B'}$$

We also compare universe instances for polymorphic constants.

About η. Coq supports η-expansion in the conversion. In a typed setting like Agda's conversion it is rather easy to add it. In Agda the conversion rule can be given as

$$\frac{}{\Gamma \vdash t \equiv \lambda x.\, t\ x : \Pi(x : A).B}$$

Reminder: η-expansion

η-expansion is defined as follows

$$t \rightarrow_\eta \lambda x.\, t\ x$$

or even in a more directed way as

$$\frac{\Gamma, x : A \vdash f\ x \equiv g\ x : B}{\Gamma \vdash f \equiv g : \Pi(x : A).B}$$

In an untyped setting the question is a bit more complex, we have several candidates but are struggling to find a formulation that lets us preserve good properties about conversion/cumulativity without going through too much trouble.

Typing

Again, typing is defined inductively.

```
Inductive typing (Σ : global_env_ext) (Γ : context)
  : term -> term -> Type
(* ... *)
where " Σ ;;; Γ |- t : T " := (typing Σ Γ t T) : type_scope.
```

I will go over the rules one by one.

Variables.

```
type_Rel n decl :
  All_local_env (lift_typing typing Σ) Γ ->
  nth_error Γ n = Some decl ->
  Σ ;;; Γ |- tRel n : lift0 (S n) decl.(decl_type)
```

Herein `All_local_env (lift_typing typing Σ) Γ` will later on be written `wf_local Σ Γ`, it corresponds to well-formedness of the local environment $\Sigma \vdash \Gamma$. This basically means that the types and local definitions in it are well-typed.

```
nth_error Γ n = Some decl
```

verifies that the variable `n` corresponds to a declaration in `Γ`. The type of the variable is thus the type stored in this declaration. As always it has to be *lifted*, *i.e.* weakened to account for the declarations that were added successively in `Γ`.

Sorts.

```
type_Sort l :
  wf_local Σ Γ ->
  LevelSet.In l (global_ext_levels Σ) ->
  Σ ;;; Γ |- tSort (Universe.make l) : tSort (Universe.super l)
```

For universes we once again require the local environment to be well-formed. The universe should also make sense in the global environment, in which case it is typed by its successor universe.

Π-types.

```
type_Prod na A B s1 s2 :
  Σ ;;; Γ |- A : tSort s1 ->
  Σ ;;; Γ ,, vass na A |- B : tSort s2 ->
  Σ ;;; Γ |- tProd na A B : tSort (sort_of_product s1 s2)
```

$\Pi(x : A).B$ is well-typed if A and B are well-typed too. The Π-type lives in another universe computed from the universes of A and B, if those are Type_i and Type_j this will be $\mathsf{Type}_{\max\ i\ j}$ but if the second is Prop, then the Π-type lives in Prop as well thanks to impredicativity.

λ-abstractions.

```
type_Lambda na A t s1 B :
  Σ ;;; Γ |- A : tSort s1 ->
  Σ ;;; Γ ,, vass na A |- t : B ->
  Σ ;;; Γ |- tLambda na A t : tProd na A B
```

let-bindings.

```
type_LetIn na b B t s1 A :
  Σ ;;; Γ |- B : tSort s1 ->
  Σ ;;; Γ |- b : B ->
  Σ ;;; Γ ,, vdef na b B |- t : A ->
  Σ ;;; Γ |- tLetIn na b B t : tLetIn na b B A
```

Here we see the interesting fact that let-bindings are typed by let-bindings themselves. That is because the type A also makes sense in `Γ ,, vdef na b B` that is an environment extended with a local definition.

Application.

```
type_App t na A B u :
  Σ ;;; Γ |- t : tProd na A B ->
  Σ ;;; Γ |- u : A ->
  Σ ;;; Γ |- tApp t u : B{0 := u}
```

Here `B{0 := u}` stands for B where the 0-th variable is substituted by u.

Constants.

```
type_Const k u :
  wf_local Σ Γ ->
  forall d (isdecl : declared_constant Σ.1 k d),
  consistent_instance_ext Σ d.(cst_universes) u ->
  Σ ;;; Γ |- tConst k u : subst_instance_constr u d.(cst_type)
```

The rule is a complicated way of saying that if the constant k is declared in the global environment and used with a consistent instance of universes, it has the type prescribed by Σ instantiated with those universes.

Inductive types.

```
type_Ind ind u :
  wf_local Σ Γ ->
  forall md id (isdecl : declared_inductive Σ.1 md ind id),
  consistent_instance_ext Σ md.(ind_universes) u ->
  Σ ;;; Γ |- tInd ind u : subst_instance_constr u id.(ind_type)
```

This rule is pretty similar to constants because without their constructors, inductive types are really just constants.

Constructors.

```
type_Construct ind i u :
  wf_local Σ Γ ->
  forall md id cd
    (isdecl : declared_constructor Σ.1 md id (ind, i) cd),
    consistent_instance_ext Σ md.(ind_universes) u ->
    Σ ;;; Γ |- tConstruct ind i u :
               type_of_constructor md cd (ind, i) u
```

A constructor is well-typed when the inductive it refers to is declared and the constructor itself is declared in it. `type_of_constructor` recovers the corresponding type and put it in context, substituting the universes.

Pattern-matching.

```
type_Case in u p c brs args :
  let ind := in.1 in
  let npar := in.2 in
  forall md id (isdecl : declared_inductive Σ.1 md ind id),
  md.(ind_npars) = npar ->
  let params := List.firstn npar args in
  forall ps pty,
  build_case_predicate_type ind md id params u ps = Some pty ->
  Σ ;;; Γ |- p : pty ->
  leb_sort_family (universe_family ps) id.(ind_kelim) ->
  Σ ;;; Γ |- c : mkApps (tInd ind u) args ->
  forall btys,
  map_option_out (build_branches_type ind md id params u p) =
  Some btys ->
  All2 (fun br bty =>
    (br.1 = bty.1) *
    (Σ ;;; Γ |- br.2 : bty.2) *
    (Σ ;;; Γ |- bty.2 : tSort ps)
  ) brs btys ->
  Σ ;;; Γ |- tCase in p c brs :
            mkApps p (skipn npar args ++ [c])
```

This rule is rather hairy. We verify that the inductive type given for the scrutinee is declared; we verify that the return predicate has the right type and is in a sort which can be eliminated to; we verify that the scrutinee is indeed typed in the inductive type and we recover the indices from its type, they will be used in the return type; finally we verify that each branch is well-typed using the other information.

Projection.

```
type_Proj p c u :
  forall md id pd
    (isdecl : declared_projection Σ.1 md id p pd) args,
    Σ ;;; Γ |- c : mkApps (tInd (fst (fst p)) u) args ->
    #|args| = ind_npars md ->
    let ty := snd pd in
    Σ ;;; Γ |- tProj p c :
    subst0 (c :: List.rev args) (subst_instance_constr u ty)
```

As usual we verify that the projection is declared and corresponds to a declared record type. The projected term must be of that record type, where universes are instantiated. The projection then has the type of the corresponding field with these universes.

Fixed-points.

```
type_Fix mfix n decl :
  fix_guard mfix ->
  nth_error mfix n = Some decl ->
  wf_local Σ Γ ->
  All (fun d => {s & Σ ;;; Γ |- d.(dtype) :  tSort s}) mfix ->
  All (fun d =>
    (Σ ;;; Γ ,,, fix_context mfix |-
      d.(dbody) :
      lift0 #|fix_context mfix| d.(dtype)
    ) *
    (isLambda d.(dbody) = true)
  ) mfix ->
  Σ ;;; Γ |- tFix mfix n : decl.(dtype)
```

A fixed-point is well-typed if all the types of the different mutual functions are sorted (*i.e.* are typed in a universe) and if the bodies have their ascribed types in the context extended with the mutual functions (the bodies can refer to other functions or do a recursive call to themselves). We also ask that each body is at least a λ-abstraction to make sure it takes at least one argument explicitly. The interesting point however is

```
fix_guard mfix
```

This is our own representation of the guard condition ensuring that the fixed-point is terminating. As the guard condition of Coq is rather complicated and perhaps ad-hoc we instead assume we have such a guard condition satisfying some properties and work with it in generality. For this we use axioms

```
Axiom fix_guard : mfixpoint term -> bool.

Axiom fix_guard_red1 :
  forall Σ Γ mfix mfix' idx,
    fix_guard mfix ->
    red1 Σ Γ (tFix mfix idx) (tFix mfix' idx) ->
    fix_guard mfix'.

(* ... *)
```

which has the drawback that they cannot be instantiated with actual guard conditions. We will probably change this at some point in the future.

Cofixed-points.

```
type_CoFix mfix n decl :
  cofix_guard mfix ->
  nth_error mfix n = Some decl ->
  wf_local Σ Γ ->
  All (fun d => {s & Σ ;;; Γ |- d.(dtype) :  tSort s}) mfix ->
  All (fun d =>
    Σ ;;; Γ ,,, fix_context mfix |-
      d.(dbody) :
      lift0 #|fix_context mfix| d.(dtype)
  ) mfix ->
  Σ ;;; Γ |- tCoFix mfix n : decl.(dtype)
```

Cofixed-points are pretty similar to fixed-points, with one exception. Similarly to fixed-points, not all cofixed-points are valid. Instead of termination we talk about *productivity* which essentially means that each observation—for instance accessing the head of a stream—on the cofixed-point should terminate. This is the purpose of `cofix_guard` that is defined using axioms, similarly to `fix_guard`.

Cumulativity.

```
type_Cumul t A B :
  Σ ;;; Γ |- t : A ->
  (isWfArity typing Σ Γ B + {s & Σ ;;; Γ |- B : tSort s}) ->
  Σ ;;; Γ |- A <= B ->
  Σ ;;; Γ |- t : B
```

The cumulativity rule is mostly unsurprising, if $t : A$ and $A \preceq B$ then $t : B$. As I have showed several times we require B to be well-formed because conversion is untyped. The way it is written here might be surprising however: we ask for the existence of a sort typing B *or* that B is a well-formed *arity*. This is an oddity of Coq coming from the presence of so-called *algebraic universes* consisting of things such as $i+1$ or $\max\ i\ j$. These universes should only appear on the right-hand side of the colon, *i.e.* in types. An arity is a well-formed quantification over a universe that may be algebraic. As those can appear in types, we allow them in the cumulativity rule. This peculiar aspect can be mostly ignored for the rest of this document, and well-formedness of types can be thought of as the usual.

Note that I did not talk about typing of environments, local and global, but they of course are also specified in the formalisation. For inductive type declarations there is another guard condition that we axiomatise:

```
Axiom ind_guard : mutual_inductive_body -> bool.
```

In Coq this is implemented as the strict positivity condition but we once again remain abstract and simply ask for some way of determining if an inductive type is well-founded. These axioms will become meaningful when we reach the question of strong normalisation in Chapter 20 (Meta-theoretical properties).

Meta-theoretical properties 20

Before we can formalise the type-checker, we need to develop some of the meta-theory. In particular the type-checker relies on subject reduction, confluence and strong normalisation of the reduction. As I already said in Chapter 18 (Overview), these properties can be part of the trusted base (and some of them like strong normalisation have to be in it, lest we prove consistency of Coq within Coq). Moreover they are not the subject of this work. I will thus focus on their statements and not their potential proofs, those are explained in [4] or will be in upcoming publications regarding the MetaCoq project.

Most of these properties are defined in Chapter 7 (Desirable properties of type theories).

[4]: Sozeau et al. (2020), 'Coq Coq correct! verification of type checking and erasure for Coq, in Coq'

Substitutivity and weakening. Weakening and substitution preserve typing, a fact which we prove rather early in the development. It is really easy to make mistakes while manipulating de Bruijn indices, so having those theorems (and not merely as assumed properties) increase the confidence we can have in our handling of de Bruijn indices.

The weakening theorem is rather simple:

```
Lemma weakening :
  forall Σ Γ Γ' (t : term) T,
    wf Σ ->
    wf_local Σ (Γ ,,, Γ') ->
    Σ ;;; Γ |- t : T ->
    Σ ;;; Γ ,,, Γ' |- lift0 #|Γ'| t : lift0 #|Γ'| T.
```

It asks for the global environment and the extended local environment to make sense as a precondition.

Substitution is slightly more complex

```
Lemma substitution :
  forall Σ Γ Γ' s Δ (t : term) T,
    wf Σ ->
    subslet Σ Γ s Γ' ->
    Σ ;;; Γ ,,, Γ' ,,, Δ |- t : T ->
    Σ ;;; Γ ,,, subst_context s 0 Δ |-
      subst s #|Δ| t : subst s #|Δ| T.
```

This time we are replacing a bunch of variables at once using so-called parallel substitutions. Not only the term and type, but also the context that was built on top of the substituted variables are substituted. Unlike weakening, the substitution itself needs to be well-typed, this is defined as follows.

```
Inductive subslet {cf:checker_flags} Σ (Γ : context)
  : list term -> context -> Type :=

| emptyslet :
    subslet Σ Γ [] []

| cons_let_ass Δ s na t T :
    subslet Σ Γ s Δ ->
    Σ ;;; Γ |- t : subst0 s T ->
    subslet Σ Γ (t :: s) (Δ ,, vass na T)
```

```
| cons_let_def Δ s na t T :
    subslet Σ Γ s Δ ->
    Σ ;;; Γ |- subst0 s t : subst0 s T ->
    subslet Σ Γ (subst0 s t :: s) (Δ ,, vdef na t T).
```

Substitution typing needs to account for local definitions (*i.e.* let-bindings) in the environment. In a more understandable language this becomes

$$\frac{}{\Sigma ; \Gamma \vdash \bullet : \bullet} \qquad \frac{\Sigma ; \Gamma \vdash \sigma : \Delta \qquad \Sigma ; \Gamma \vdash t : A[\sigma]}{\Sigma ; \Gamma \vdash \sigma, x \leftarrow t : \Delta, x : A}$$

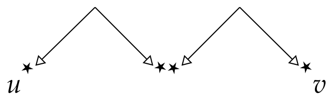
$$\frac{\Sigma ; \Gamma \vdash \sigma : \Delta \qquad \Sigma ; \Gamma \vdash t[\sigma] : A[\sigma]}{\Sigma ; \Gamma \vdash \sigma, x \leftarrow t[\sigma] : \Delta, x : A \coloneqq t}$$

This time I write $\Gamma \vdash \sigma : \Delta$ were earlier I wrote $\sigma : \Gamma \to \Delta$. This is because in this case σ is a list of terms.

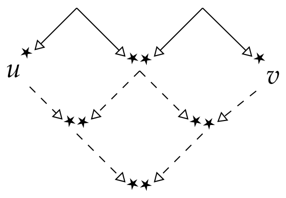
It can help to think that $\Delta \vdash A$ holds for the last two rules and that $\Delta \vdash t : A$ holds for the last one.

Notice how the term of the substitution is forced when it targets a definition, this is reassuring because a substitution should not overwrite definitions.

Confluence. Confluence is an important result that we have on Coq's theory. In particular it allows us to simplify greatly what it means to be convertible. Indeed, right now, conversion involves reduction going in both directions so that u and v can be convertible by something like the following picture.



Thanks to confluence this picture can be completed into



This means that we can only regard conversion as u and v reducing to the same term (up to names and universes).

Context conversion. This is a property that I did not mention before: context conversion states that you can replace a context in a judgment by another context where all types and definitions are convertible to those of the original context. This will often be used to replace $\Gamma, x : A$ by $\Gamma, x : A'$ with $A \equiv A'$ but is proven in full generality.

Validity. In PCUIC, validity differs from the usual because of arities.

```
Lemma validity_term :
  forall Σ Γ t T,
    wf Σ ->
    Σ ;;; Γ |- t : T ->
    isWfArity_or_Type Σ Γ T.
```

As you can see, T is guaranteed to either be a type (as in be typed in a universe) or a well-formed arity.

Subject reduction. Subject reduction was not proven at the time of the article [4] but is now proven.

[4]: Sozeau et al. (2020), 'Coq Coq correct! verification of type checking and erasure for Coq, in Coq'

```
Theorem subject_reduction :
  forall Σ Γ t A B,
    wf Σ ->
    Σ ;;; Γ |- t : A ->
    red Σ Γ A B ->
    Σ ;;; Γ |- t : B.
```

Principality. Because of cumulativity, PCUIC cannot have uniqueness of types, it however has principal types which we express as follows.

```
Theorem principal_typing :
  forall Σ Γ u A B,
    Σ ;;; Γ |- u : A ->
    Σ ;;; Γ |- u : B ->
    Σ C,
      Σ ;;; Γ |- C <= A ×
      Σ ;;; Γ |- C <= B ×
      Σ ;;; Γ |- u : C.
```

If a term u has two types, then there is a smaller type typing u. Similarly, this property is now proven.

Strong normalisation. There is no hope of proving strong normalisation of PCUIC inside Coq. As such it will be stated as an axiom.

The usual presentation of strong normalisation stating that there is no infinite reduction sequence is ill-suited in a constructive setting so we instead show that the opposite of reduction is well-founded. This opposite, which I call coreduction, is defined as follows:

An infinite decreasing sequence for the opposite of reduction is simply an infinite reduction sequence, so we are indeed talking about the same concept.

```
Inductive cored Σ Γ : term -> term -> Prop :=
| cored1 :
    forall u v,
      red1 Σ Γ u v ->
      cored Σ Γ v u

| cored_trans :
    forall u v w,
      cored Σ Γ v u ->
      red1 Σ Γ v w ->
      cored Σ Γ w u.
```

It is the transitive closure of the symmetric of 1-step reduction.

As I will explain in more depth in Chapter 21 (Well-founded induction and well-orders), we use accessibility predicates to state that a relation is well-founded constructively. As such the axiom of strong normalisation is

```
Axiom normalisation :
  forall Γ t,
    wf Σ ->
    wellformed Σ Γ t ->
    Acc (cored (fst Σ) Γ) t.
```

Herein `wellformed Σ Γ t` states that *t* is either a well-typed term or a well-formed arity. `wellformed` also lands in **`Prop`** so that it gets erased during extraction to OCaml.

Well-founded induction and well-orders 21

Before delving into the definition of the type checker which will rely on strong normalisation, we need to study well-founded induction and orders in more detail (remember that strong normalisation is defined in terms of well-foundedness of coreduction).

21.1 General setting

The *classical* definition of a well-founded order basically says that one cannot *go down* indefinitely in that order.

Definition 21.1.1 (Well-order (classically)) *An order $\prec$ is said to be well-founded when there is no infinitely decreasing sequence for that order, i.e. there is no sequence $(u_i)_{i \in \mathbb{N}}$ such that for all i, $u_{i+1} \prec u_i$.*

When $\prec$ is a well-order, every non-increasing sequence has to become stationary at some point. The prototypical example of this the canonical order on $\mathbb{N}$: if you take a natural number and start going down, at some point you have to stop lest you go below 0 and leave the realm of natural numbers. We say that $\mathbb{N}$ is a well-founded set in that case.

This is particularly useful to show that a process terminates. If after completing each task you are left with the completion of smaller tasks, you will eventually run out of tasks to complete. This can be used to justify the termination of an OCaml function like the ever-famous factorial function.

```
let rec fact n =
  if n < 1
  then 1
  else n * (fact (n - 1))
```

Even though the OCaml `int` type is not comprised only of natural numbers, the `n < 1` condition ensures that the recursive call only happens when $n \geq 1$.

The recursive call is always on a smaller natural number and at some point it will reach some $n < 1$ and return a value.

This kind of reasoning is the key to proofs of termination in general, but this presentation is not really suited to a constructive setting like Coq because of the negative formulation.

21.2 Constructive setting

In an intuitionistic or even constructive setting, the notion of well-foundedness is stated in a positive way: instead of saying that there is no infinitely decreasing sequence, we say that every element—for instance every natural number—is *reachable* in a finite number of steps from the base case(s).

Definition 21.2.1 (Accessibility) *A term t of type A is accessible for a relation $\prec$ on A when all smaller elements for $\prec$ are also accessible.*

In particular, a minimal element t—in the sense that there are no u such that $u \prec t$—is always accessible.

For instance, to show that 2 is accessible for $<$ we would need to show that 0 and 1 are accessible, and for 1 we would need to show that 0 is accessible. This leaves us with 0. It is accessible because there are no smaller natural numbers. 0 is the base case.

Definition 21.2.2 (Well-founded relation) *A relation $\prec$ on A is well-founded when all the inhabitants of A are accessible for $\prec$.*

Note that we define well-founded relations and not well-orders, this is more general and actually sufficient for us. A well-order is an order that forms a well-founded relation.

Again, this is the case that all natural numbers are accessible for $<$ and this can be shown by induction. From this we can do well-founded induction: if a property holds for t when it holds for smaller elements, it holds for every accessible element.

In Coq the accessibility predicate is defined as follows:

```
Inductive Acc (A : Type) (R : A -> A -> Prop) (x : A) : Prop :=
  Acc_intro : (forall y : A, R y x -> Acc R y) -> Acc R x
```

This corresponds to a translation of the definition above to the syntax of Coq. The induction principle produced by Coq for `Acc` yields well-founded induction.

```
Acc_ind :
  forall (A : Type) (R : A -> A -> Prop) (P : A -> Prop),
    (forall x : A,
      (forall y : A, R y x -> P y) ->
      P x
    ) ->
    forall x : A, Acc R x -> P x
```

It says that if for any given $x : A$ such that all $y : A$ smaller than x are accessible and verify the predicate P, x also verifies P, then all accessible elements verify P.

Let us say this time we want to define the Fibonacci sequence, the following definition is not accepted by Coq's termination checker.

Reminder: Fibonacci sequence

The Fibonacci sequence $(\mathcal{F}_n)_{n\in\mathbb{N}}$ is defined by $\mathcal{F}_0 = 0$, $\mathcal{F}_1 = 1$ and $\mathcal{F}_{n+2} = \mathcal{F}_{n+1} + \mathcal{F}_n$.

```
Fail Fixpoint fib n :=
  match n with
  | 0 => 1
  | 1 => 1
  | S (S n) => fib (S n) + fib n
  end.
```

It does not like the `S n` bit in the recursive call. I am cheating a bit because it can still be defined by telling Coq that `S n` here is indeed a subterm of the matched term.

```
Fixpoint fib n :=
  match n with
  | 0 => 1
  | 1 => 1
  | S ((S n) as m) => fib m + fib n
  end.
```

For the sake of the argument I will still write this function using well-founded induction.

If you are not convinced by this argument, just read the rest of the thesis to see well-founded induction in action.

```
Require Import Lia Arith.

Lemma wf_nat :
  well_founded lt.
Proof.
  intro n. induction n as [| n ih].
  - constructor. intros m h. lia.
  - constructor. intros m h.
    destruct (eq_nat_dec m n).
    + subst. assumption.
    + destruct ih as [ih]. apply ih. lia.
Qed.

Definition fib n :=
  Acc_rec (fun _ => nat)
    (fun n =>
      match n with
      | 0 => fun _ _ => 1
      | 1 => fun _ _ => 1
      | S (S m) =>
        fun acc_lt_n fib_lt_n =>
          fib_lt_n (S m) (Nat.lt_succ_diag_r (S m)) +
          fib_lt_n m
            (Nat.lt_trans _ _ _
              (Nat.lt_succ_diag_r m)
              (Nat.lt_succ_diag_r (S m)))
      end
    )
    (wf_nat n).
```

The first line loads the `lia` tactic which solves arithmetic goals.

`well_founded R` is defined as `forall x : A, Acc R x` for `R : A -> A -> Prop`. Also `n < m` is just a notation for `lt n m`.

The `fib` function we tried to define earlier is barely recognisable here. Fortunately for us there are tools to write well-founded inductions in a nicer way. One such tool is Equations.

```
From Equations Require Equations.

Equations fib (n : nat) : nat
  by wf n lt :=

  fib 0 := 1 ;
  fib 1 := 1 ;
  fib (S (S n)) := fib (S n) + fib n.
```

The syntax differs a bit from vanilla Coq but should be pretty self-explanatory. We write the function just the way we want it to be and specify that we are using well-founded recursion on n using the standard `lt` order with `by wf n lt`. Then it generates some proof obligations saying that the recursive calls are indeed made on smaller arguments for `lt`, but as they are pretty simple, it solves them for us. Moreover it uses the type-class mechanism to automatically find a proof that the order is indeed well-founded. Thanks to Equations, well-founded recursion is not too painful and pretty useful.

21.3 Lexicographic orders

I will dedicate this section to a special case of order that will be useful for the formalisation.

Simple lexicographic orders

Generally speaking, a lexicographic order is the combination of two orders to yield an order on pairs. The idea is similar to the alphabetical order: you first compare the first letter of each word, and if they match you proceed to the second, etc.

Definition 21.3.1 (Lexicographic order) *Given two types A and B and two orders $\prec_A$ and $\prec_B$ on them, the lexicographic order $\prec_{\text{lex}}$ on $A \times B$ is defined as*

$$(a, b) \prec_{\text{lex}} (a', b') \text{ when } \begin{cases} \text{either} & a \prec_A a' \\ \text{or} & a = a' \text{ and } b \prec_B b' \end{cases}$$

So, like the alphabetical order, when the two first elements are in the relation, we will not even look at the rest. This is pretty convenient, especially since lexicographic orders preserve well-foundedness.

They also remain orders.

Lemma 21.3.1 (Well-founded lexicographic orders) *If $\prec_A$ and $\prec_B$ are both well-founded, then the lexicographic order they induce is also well-founded.*

With lexicographic orders we can define the Ackermann function for instance.

```
Equations ack (x : nat * nat) : nat
  by wf x (lexprod _ _ lt lt) :=

  ack (0,    n)   := S n ;
  ack (S m, 0)    := ack (m, 1) ;
  ack (S m, S n) := ack (m, ack (S m, n)).
```

`lexprod` is the lexicographic order in Coq as provided by Equations itself.

Once again, Equations saves the day by providing us with a readable version of the function, while hiding the well-foundedness proof: the definition is accepted as-is with no more input required of the user.

In the context of dependent types, these might not prove sufficient, hence the need for dependent lexicographic orders.

Dependent lexicographic orders

In the same way that simple products $A \times B$ can be extended to their dependent counterpart, namely Σ-types: $\Sigma(x : A).B$, lexicographic orders have a dependent version. As before, we take an order $\prec_A$ on A, but as B depends on $x : A$, we need the second order to also depend on x. Fortunately for us, the only times we are interested in $\prec_B$ is when the first two components are the same ($a = a'$), meaning the same type $B[x \leftarrow a]$.

Definition 21.3.2 (Dependent lexicographic order) *Given a type A and a type B dependent on A, as well as an order $\prec_A$ on A and an order $\prec_B$ on B, dependent on A, we define their dependent lexico-*

graphic order on $\Sigma(x : A).B$, $\prec_{\text{lex}}$ as

$$\begin{array}{lll}\langle a,b\rangle \prec_{\text{lex}} \langle a',b'\rangle & \textit{when} & a \prec_A a' \\ \langle a,b\rangle \prec_{\text{lex}} \langle a,b'\rangle & \textit{when} & b \prec_{B[x\leftarrow a]} b'\end{array}$$

Notice in the second case that there is no a', it is twice a.

I find the Coq definition can shed more light than the paper one in this case. In it the dependencies appear in a more explicit manner so it can lift some confusion I hope.

```
Inductive dlexprod {A} {B : A -> Type}
  (leA : A -> A -> Prop) (leB : forall x, B x -> B x -> Prop)
  : sigT B -> sigT B -> Prop :=

| left_lex :
    forall x x' y y',
      leA x x' ->
      dlexprod leA leB (x;y) (x';y')

| right_lex :
    forall x y y',
      leB x y y' ->
      dlexprod leA leB (x;y) (x;y').
```

`sigT B` has implicit parameter `A` and corresponds to the Σ-type `Σ (x : A), B x`.

In Coq I write `(x;y)` for $\langle x,y\rangle$.

I use notations for the simple and dependent lexicographic orders:

```
Notation "R1 ⊗ R2" :=
  (lexprod R1 R2)
  (at level 20, right associativity).

Notation "x ⊩ R1 ⊗ R2" :=
  (dlexprod R1 (fun x => R2))
  (at level 20, right associativity).
```

Once again, the dependent lexicographic order of two well-founded orders is also well-founded. In Coq I show something even more precise, saying that a pair $\langle x,y\rangle$ is accessible if x is accessible and the second order is well-founded.

```
Lemma dlexprod_Acc :
  forall A B leA leB,
    (forall x, well_founded (leB x)) ->
    forall x y,
      Acc leA x ->
      Acc (@dlexprod A B leA leB) (x;y).
```

Lexicographic order modulo a bisimulation

With usual lexicographic orders, the second order is involved when the first components are *equal*. In some cases, this requirement is just too strong and we want to relax it a bit, by accepting them to be equal modulo another relation. Of course we cannot accept any relation, otherwise we would have no hope of showing the resulting order well-founded.

Again I will stick to the Coq definition because I find it better to convey information.

```
Inductive dlexmod {A} {B : A -> Type}
    (leA : A -> A -> Prop)
    (eA : A -> A -> Prop)
    (coe : forall x x', eA x x' -> B x -> B x')
    (leB : forall x, B x -> B x -> Prop)
  : sigT B -> sigT B -> Prop :=

| left_dlexmod :
    forall x x' y y',
      leA x x' ->
      dlexmod leA eA coe leB (x;y) (x';y')

| right_dlexmod :
    forall x x' y y' (e : eA x x'),
      leB x' (coe _ _ e y) y' ->
      dlexmod leA eA coe leB (x;y) (x';y').

Notation "x ⊨ e \ R1 'by' coe ⊗ R2" :=
  (dlexmod R1 e coe (fun x => R2))
  (at level 20, right associativity).
```

It is very similar to the dependent lexicographic order of the previous section, only this time we compare the second components when they are related by `eA`. Because of the dependency however, we have `y : B x` and `y' : B x'` while `leB` will want only one of either `x` or `x'`. For this reason we need to somehow relate `B x` and `B x'`; that is the purpose of the `coe` function we take as argument. It *transports* terms of `B x` to `B x'`; that way we compare `coe x x' e y` and `y'` for `leB`. The existence of such a coercion function is not automatic, hence the need to have it explicitly in the notation `x ⊨ e \ R1 by coe ⊗ R2`.

As you probably have guessed, dependent lexicographic order modulo a relation can also be shown to be well-founded, provided it involves well-orders and that the coercion function is well-behaved. Here is the lemma—whose statement is longer than its proof so I will break it down right after—in Coq.

```
Lemma dlexmod_Acc :
  forall A B (leA : A -> A -> Prop) (eA : A -> A -> Prop)
    (coe : forall x x', eA x x' -> B x -> B x')
    (leB : forall x : A, B x -> B x -> Prop)
    (sym : forall x y, eA x y -> eA y x)
    (trans : forall x y z, eA x y -> eA y z -> eA x z),
    (forall x, well_founded (leB x)) ->
    (forall x x' y, eA x x' -> leA y x' -> leA y x) ->
    (forall x,
      exists e : eA x x, forall y, coe _ _ e y = y
    ) ->
    (forall x x' y e,
      coe x x' (sym _ _ e) (coe _ _ e y) = y
    ) ->
    (forall x0 x1 x2 e1 e2 y,
      coe _ _ (trans x0 x1 x2 e1 e2) y =
      coe _ _ e2 (coe _ _ e1 y)
    ) ->
    (forall x x' e y y',
      leB _ y (coe x x' e y') ->
      leB _ (coe _ _ (sym _ _ e) y) y'
    ) ->
    forall x y,
      Acc leA x ->
      Acc (@dlexmod A B leA eA coe leB) (x ; y).
```

Admittedly this formulation is not really pretty, and comes as the minimal subset of requirement I found with which I could still prove well-foundedness. The first thing to notice, is the addition of two operators on the relation `eA`: I ask that it is symmetric and transitive as witnessed by `sym` and by `trans`. Then I list several properties that I require, I shall list them one by one:

(i) `leB` has to be well-founded;
(ii) objects in relation for `eA` can substituted in `leA` on the right;
(iii) `eA` should be reflexive and coercing using the proof of reflexivity is the identity;
(iv) `sym _ _ e` and `e` cancel each other out for `coe`;
(v) coercing a transitivity is the same as coercing twice;
(vi) if you have a coercion of the right of `leB` you can instead put its symmetric on the left.

There is probably a hidden notion here like an order isomorphism but this would require a more thorough investigation.

All these should feel natural if you have equality in mind. I did mention bisimulations however and did not even talk about it. Let me now address this point.

Definition 21.3.3 (Simulation) *Given a relation $\twoheadrightarrow$ on A, a relation $\sim$ on A is called a simulation if whenever $p \twoheadrightarrow q$ and $p \sim p'$ there exists q' such that $q \sim q'$ and $p' \twoheadrightarrow q'$.*

This is summarised by the following diagram.

$$\begin{tikzcd}
p \arrow[r, no head, squiggly] \arrow[d] & p' \arrow[d, dashed] \\
q \arrow[r, no head, squiggly, dashed] & \exists q'
\end{tikzcd}$$

Definition 21.3.4 (Bisimulation) *A relation $\sim$ on A is a bisimulation for $\twoheadrightarrow$ when $\sim$ and $\sim^{-1}$ are both simulations for $\twoheadrightarrow$.*

A bisimulation does not quite fit the bill yet though. The simulation property does look like (ii) but it would conclude `leA y' x` for some `eA y y'` instead of plain `leA y x`. Also, there is no reason for a bisimulation to be an equivalence relation and even less for the existence of a well-behaved coercion function for the second order.

If we go back to the simple—*i. e.* not dependent—case for a bit, a bisimulation that is also an equivalence relation is sufficient.

```
Inductive lexmod {A B}
    (leA : A -> A -> Prop)
    (eA : A -> A -> Prop)
    (leB : B -> B -> Prop)
  : A * B -> A * B -> Prop :=

| left_lexmod :
    forall x x' y y',
      leA x x' ->
      lexmod leA eA leB (x,y) (x',y')

| right_lexmod :
    forall x x' y y',
      eA x x' ->
      leB y y' ->
      lexmod leA eA leB (x,y) (x',y').

Notation "e \ R1 ⊕ R2" :=
  (lexmod R1 e R2)
  (at level 20, right associativity).

Lemma lexmod_Acc :
  forall A B (leA : A -> A -> Prop) (eA : A -> A -> Prop)
    (leB : B -> B -> Prop),
    well_founded leB ->
    (forall x x' y,
      eA x x' ->
      leA y x' ->
      exists y',
        leA y' x *
        eA y' y
    ) ->
    (forall a, eA a a) ->
    (forall a b c, eA a c -> eA b c -> eA a b) ->
    forall x y,
      Acc leA x ->
      Acc (eA \ leA ⊕ leB) (x, y).
```

Here it might not seem obvious that I am indeed requiring `eA` to be an equivalence relation but if you read carefully the expression `eA a c -> eA b c -> eA a b` does not state transitivity but a mix between transitivity and symmetry (assuming reflexivity).

Regarding the requirement of the simulation to also be an equivalence relation, we can actually use the notion of bisimilarity, a special kind of bisimulation.

Definition 21.3.5 (Bisimilarity) *Given a relation $\twoheadrightarrow$ on type A, two terms p and q in A are called* bisimilar *when there exist a bisimulation $\sim$ for $\twoheadrightarrow$ such that $p \sim q$.*

The bisimilarity relation is itself an equivalence relation meaning we can use it to quotient simple lexicographic orders.

Again, in the dependent case this is not enough but I still kept the name because it is roughly a generalisation of the simple case.

In the simple case, the coercion function is always the indentity, so all its properties follow trivially.

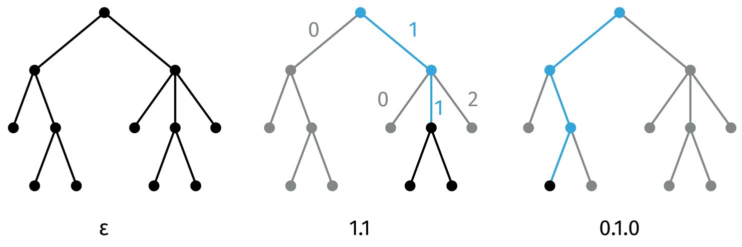
Term positions and stacks 22

As we will see in Chapter 23 (Reduction) and Chapter 24 (Conversion), when manipulating terms we sometimes have to go deep withing subterms. Positions point you to a specific subterm of a term while stacks operate as some sort of terms with a hole or equivalently some evaluation environments. These notions are studied in [88–90] for instance.

[88]: Danvy et al. (2008), 'On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion'
[89]: Biernacka et al. (2009), 'Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines'
[90]: Danvy (2009), 'Towards compatible and interderivable semantic specifications for the Scheme programming language, Part I: Denotational semantics, natural semantics, and abstract machines'

22.1 Positions

In a general setting, positions in trees are given by sequences of choices or directions. The empty sequence corresponds to the root of the tree, and at each branching you have to say which branch you want to take.



Sequences such as $0.1.0$ are read from left to right, and correspond to directions starting from the root. In black is the subtree at the given position.

Now, terms are a special kind of tree so we can do something similar. There are many ways to represent positions: for instance in the example above the position 0.3 does not correspond to anything, so is it still considered a position, only an *invalid* one? Or should all expressible positions be valid?

My approach is a bit in between, I constrain the syntax of choices a bit more than this, but they are not necessarily valid. Basically choices are defined inductively, with several constructors for each of the constructs of the syntax: for instance, applications will have two corresponding choices, one for the applicant, one for the argument.

```
Inductive choice :=
| app_l
| app_r
| case_p
| case_c
| case_brs (n : nat)
| proj_c
| fix_mfix_ty (n : nat)
| fix_mfix_bd (n : nat)
| lam_ty
| lam_tm
| prod_l
| prod_r
| let_bd
| let_ty
| let_in.
```

`app_l` corresponds to going left under an application while `app_r` corresponds to going right.

A position is just a list of choices.

```
Definition position := list choice.
```

Now as I already said, these are not necessarily valid positions, for this we define a function which verifies if a given position is valid in a given term.

```
Fixpoint validpos t (p : position) {struct p} :=
  match p with
  | [] => true
  | c :: p =>
    match c, t with
    | app_l, tApp u v => validpos u p
    | app_r, tApp u v => validpos v p
    | case_p, tCase indn pr c brs => validpos pr p
    | case_c, tCase indn pr c brs => validpos c p
    | case_brs n, tCase indn pr c brs =>
      match nth_error brs n with
      | Some (_, br) => validpos br p
      | None => false
      end
    | proj_c, tProj pr c => validpos c p
    | fix_mfix_ty n, tFix mfix idx =>
      match nth_error mfix n with
      | Some d => validpos d.(dtype) p
      | None => false
      end
    | fix_mfix_bd n, tFix mfix idx =>
      match nth_error mfix n with
      | Some d => validpos d.(dbody) p
      | None => false
      end
    | lam_ty, tLambda na A t => validpos A p
    | lam_tm, tLambda na A t => validpos t p
    | prod_l, tProd na A B => validpos A p
    | prod_r, tProd na A B => validpos B p
    | let_bd, tLetIn na b B t => validpos b p
    | let_ty, tLetIn na b B t => validpos B p
    | let_in, tLetIn na b B t => validpos t p
    | _, _ => false
    end
  end.
```

It should not feel too surprising, the empty position is always valid, and otherwise, the head choice should match the structure of the term. There are some trickier cases for pattern-matching and fixed-points because they involve lists of terms but it is still pretty natural.

This function might serve as a specification for the positions.

Finally we can define a type of valid positions in a term using a subset type.

```
Definition pos (t : term) :=
  { p : position | validpos t p = true }.
```

For instance `[ app_l ; let_in ]` is valid position in term

```
tApp (tLetIn na b B t) u
```

which represents the term `(let na := b : B in t) u` and points to subterm `t`.

We can also define a function to access the subterm at a given position.

```
Fixpoint atpos t (p : position) {struct p} : term :=
  match p with
  | [] => t
  | c :: p =>
    match c, t with
    | app_l, tApp u v => atpos u p
```

```
   | app_r, tApp u v => atpos v p
   | case_p, tCase indn pr c brs => atpos pr p
   | case_c, tCase indn pr c brs => atpos c p
   | case_brs n, tCase indn pr c brs =>
     match nth_error brs n with
     | Some (_, br) => atpos br p
     | None => tRel 0
     end
   | proj_c, tProj pr c => atpos c p
   | fix_mfix_ty n, tFix mfix idx =>
     match nth_error mfix n with
     | Some d => atpos d.(dtype) p
     | None => tRel 0
     end
   | fix_mfix_bd n, tFix mfix idx =>
     match nth_error mfix n with
     | Some d => atpos d.(dbody) p
     | None => tRel 0
     end
   | lam_ty, tLambda na A t => atpos A p
   | lam_tm, tLambda na A t => atpos t p
   | prod_l, tProd na A B => atpos A p
   | prod_r, tProd na A B => atpos B p
   | let_bd, tLetIn na b B t => atpos b p
   | let_ty, tLetIn na b B t => atpos B p
   | let_in, tLetIn na b B t => atpos t p
   | _, _ => tRel 0
   end
 end.
```

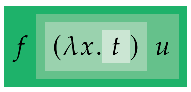
The `tRel 0` case is in an impossible branch when the position is valid, but for simplicity, the function is defined for any position. Otherwise we would have to carry the proof that it is valid everywhere.

Positions let you go deep inside a term, forgetting about its surrounding; surrounding which can be recorded using stacks.

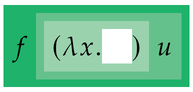
22.2 Stacks

My use of the term *stack* might be an abuse as it is probably a generalisation of it and might be better called an evaluation environment or context; I will stick to *stack* anyway. The main reason behind the name is that it is not presented as a term with a hole but rather as a succession of terms with a hole that stack on top of each other.

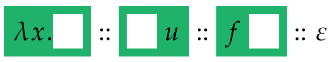
If you take the following example,



you are considering term t against the stack



It can be decomposed into

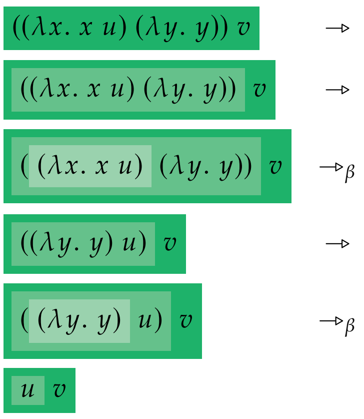


ε represents the empty stack.

meaning that the term will first be put under an abstraction, the result of this applied to u and the whole given as an argument to f.

This notion will prove particularly useful when considering the reduction machine in Chapter 23 (Reduction). Indeed the stack is a way to remember the surrounding term when focusing on a subterm. Once it has reached a normal form, we can use the stack as a *continuation*.

You can see the process step by step in the following example.



I use $\rightarrow_\beta$ to denote the cases where a *real* reduction happens and the focused term (a λ-abstraction) consumes its argument; the other cases are *focusing*, *i. e.* pushing some term with hole on the stack.

The interesting bit is that the focused term still interacts with the stack. To see clearer we can use the notation $\langle t \mid \pi \rangle$ representing term t *against* stack π. From this we can write the following reduction rules that together correspond to β-reduction.

In the formalism above, $\langle t \mid \boxed{\square\ u} :: \varepsilon \rangle$ corresponds to $\boxed{t\ u}$. I am simply now making explicit the separation between the focused term and the stack.

$$\begin{aligned} \langle u\ v \mid \pi \rangle &\rightarrow \langle u \mid \boxed{\square\ v} :: \pi \rangle \\ \langle \lambda x.\, t \mid \boxed{\square\ u} :: \pi \rangle &\rightarrow \langle t[x \leftarrow u] \mid \pi \rangle \end{aligned}$$

β-reduction now happens in two steps: $\langle (\lambda x.\, t)\, u \mid \pi \rangle \rightarrow \langle \lambda x.\, t \mid \boxed{\square\ u} :: \pi \rangle \rightarrow \langle t[x \leftarrow u] \mid \pi \rangle$

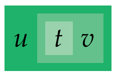
In Coq I define stacks as follows.

```
Inductive stack : Type :=
| Empty
| App (t : term) (π : stack)
| Fix (f : mfixpoint term) (n : nat) (args : list term)
     (π : stack)
| Fix_mfix_ty (na : name) (bo : term) (ra : nat)
             (mfix1 mfix2 : mfixpoint term) (id : nat)
             (π : stack)
| Fix_mfix_bd (na : name) (ty : term) (ra : nat)
             (mfix1 mfix2 : mfixpoint term) (id : nat)
             (π : stack)
| CoFix (f : mfixpoint term) (n : nat) (args : list term)
       (π : stack)
| Case_p (indn : inductive * nat) (c : term)
        (brs : list (nat * term)) (π : stack)
| Case (indn : inductive * nat) (p : term)
      (brs : list (nat * term)) (π : stack)
| Case_brs (indn : inductive * nat) (p c : term) (m : nat)
          (brs1 brs2 : list (nat * term)) (π : stack)
| Proj (p : projection) (π : stack)
| Prod_l (na : name) (B : term) (π : stack)
| Prod_r (na : name) (A : term) (π : stack)
| Lambda_ty (na : name) (b : term) (π : stack)
| Lambda_tm (na : name) (A : term) (π : stack)
| LetIn_bd (na : name) (B t : term) (π : stack)
| LetIn_ty (na : name) (b t : term) (π : stack)
| LetIn_in (na : name) (b B : term) (π : stack)
| coApp (t : term) (π : stack).
```

```
Notation "'ε'" := (Empty).
```

22.3 Positions induced by stacks

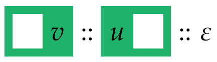
The reason I present stacks and positions together is because they are related. If you consider the term t against some stack π as one term, there exists a position in that term that points to t. For instance if you consider



then the position that first chooses the right-hand term of the top-level application and then the left-hand term will indeed point to t. In Coq syntax as earlier this would be

```
[ app_r ; app_l ]
```

In fact the position can be computed from the stack itself, regardless of the term we want to plug in. In the example above, the stack is



We read it from right to left[1] to reconstruct the position and simply give the position of the hole.

1: This is because the leftmost term with hole corresponds to the innermost element in the resulting term.

```
Fixpoint stack_position π : position :=
  match π with
  | ε => []
  | App u ρ => stack_position ρ ++ [ app_l ]
  | Fix f n args ρ => stack_position ρ ++ [ app_r ]
  | Fix_mfix_ty na bo ra mfix1 mfix2 idx ρ =>
    stack_position ρ ++ [ fix_mfix_ty #|mfix1| ]
  | Fix_mfix_bd na ty ra mfix1 mfix2 idx ρ =>
    stack_position ρ ++ [ fix_mfix_bd #|mfix1| ]
  | CoFix f n args ρ => stack_position ρ ++ [ app_r ]
  | Case_p indn c brs ρ => stack_position ρ ++ [ case_p ]
  | Case indn pred brs ρ => stack_position ρ ++ [ case_c ]
  | Case_brs indn pred c m brs1 brs2 ρ =>
    stack_position ρ ++ [ case_brs #|brs1| ]
  | Proj pr ρ => stack_position ρ ++ [ proj_c ]
  | Prod_l na B ρ => stack_position ρ ++ [ prod_l ]
  | Prod_r na A ρ => stack_position ρ ++ [ prod_r ]
  | Lambda_ty na u ρ => stack_position ρ ++ [ lam_ty ]
  | Lambda_tm na A ρ => stack_position ρ ++ [ lam_tm ]
  | LetIn_bd na B u ρ => stack_position ρ ++ [ let_bd ]
  | LetIn_ty na b u ρ => stack_position ρ ++ [ let_ty ]
  | LetIn_in na b B ρ => stack_position ρ ++ [ let_in ]
  | coApp u ρ => stack_position ρ ++ [ app_r ]
  end.
```

As I said multiple times, we will use stacks for reduction machines, to show the termination of such machines it is nice to have an order on stacks. This order will be achieved by going from stacks to positions.

22.4 Ordering positions

If you consider the valid positions in a given term (*i. e.* expressions of type `pos t` for some `t`) there exists a well-founded order on them. There are actually several due to some degree of liberty as we shall see.

Since positions are defined as lists, the natural order on them would the structural one which says that any (strict) prefix of a position is smaller. This order is not really of interest to us because it would correspond to *unfocusing*. Conversely, by extending a position with a new choice you are going deeper into the term, something which you cannot do indefinitely, you are bound to reach a leaf at some (finite) point.

If you think about it dually, I am merely explaining that the structural order on terms is well-founded, so why bother? The interesting part is that the order on positions can be more flexible than the subterm relation in that it allows us to compare 'going left' with 'going right'. Say you have the application $t := u\ v$, both u and v are subterms of t but there is no way to relate the two of them reliably and have v *always* smaller than u because u and v can be arbitrary. If instead we were to consider positions in t we would have the empty one corresponding to t itself, `[ app_l ]` corresponding to u and `[ app_r ]` for v. This means we abstract over the content of both u and v. The positions can thus be ordered as follows:

```
[ app_r ] < [ app_l ] < []
```

I am going to explain why this is desirable in Chapter 23 (Reduction).

In Coq, the definition of the order is done in two steps: first an order on raw positions, from which we then derive an order on valid positions.

```
Inductive positionR : position -> position -> Prop :=
| positionR_app_lr p q :
    positionR (app_r :: p) (app_l :: q)

| positionR_deep c p q :
    positionR p q ->
    positionR (c :: p) (c :: q)

| positionR_root c p :
    positionR (c :: p) [].

Definition posR {t} (p q : pos t) : Prop :=
  positionR (` p) (` q).
```

The `(` p)` notation corresponds to the first projection of a subset type.

While the first order is not well-founded, the second one is, as crystallised by the following lemma.

```
Lemma posR_Acc :
  forall t p, Acc (@posR t) p.
```

Reduction 23

A key element to defining a type-checker is to have a conversion checker. Conversion in turn relies on reduction. The reduction I describe in Chapter 19 (A specification of Coq) is just a relation and is non-deterministic. In this chapter I will present an algorithmic implementation of reduction. This implementation relies on the axiom of strong normalisation, but even then it requires some more work to show the process is terminating.

23.1 Weak head normalisation

Since reduction is non deterministic, we have to pick a strategy to implement. When checking terms for conversion, one thing we want to do is verify that they have the same head constructor: if two terms are λ-abstractions $\lambda(x : A).t$ and $\lambda(x : A').t'$ we need only compare A and A', as well as t and t' for conversion. Weak head reduction is a strategy that allows us to access the head of a term rather efficiently.

Characterisation

The idea is to only deal with redexes (*i.e.* left-hand side of reduction rules) that appear at the top-level and might be hiding the head constructor. As such when considering $\lambda(x : A).t$, neither A nor t will be reduced because we already know the head, or the shape, of the term: it is a λ. However, if we have an application $t\ u$, we will first weak head reduce t to see if it reaches a λ: if it does we substitute u and repeat the process, if it does not, then we have reached a weak head normal form.

Weak head normal forms are by definition the terms that cannot be reduced further by weak head reduction.

Definition 23.1.1 (Normal form) *A term t is a normal form for a reduction $\rightarrowtriangle$ if it does not reduce for $\rightarrowtriangle$.*

$$t \not\rightarrowtriangle$$

It is however possible to characterise weak head normal forms syntactically. For this we need to talk about *neutral* forms. The intuition is that a neutral term is a *stuck* term.

Definition 23.1.2 (Neutral forms) *A term t is neutral for a reduction $\rightarrowtriangle$ if substituting it inside another term does not introduce redexes. Equivalently substituting a neutral term t in a normal form u will* *yield another normal form.*

$$u[x \leftarrow t] \not\rhd$$

A neutral form is in particular a normal form.

Focusing on a small fragment of PCUIC without inductive types and so on, we can characterise weak head neutral (whne) and normal forms whnf with the following mutual judgements:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x \ \mathsf{whne}} \qquad \frac{\Gamma \vdash u \ \mathsf{whne}}{\Gamma \vdash u\ v \ \mathsf{whne}} \qquad \frac{\Gamma \vdash u \ \mathsf{whne}}{\Gamma \vdash u \ \mathsf{whnf}} \qquad \frac{}{\Gamma \vdash \mathsf{Type} \ \mathsf{whnf}}$$

$$\frac{}{\Gamma \vdash \Pi(x.A).B \ \mathsf{whnf}} \qquad \frac{}{\Gamma \vdash \lambda(x.A).t \ \mathsf{whnf}}$$

The first rule is there to make sure that the variable does not correspond to a local definition because local definitions reduce by unfolding.

As you can see, terms like $(\lambda(x.A).t)\ u$ or let $x := u$ in t are not weak head normal forms, and a fortiori not weak head neutral forms. Indeed they both reduce to $t[x \leftarrow u]$ for weak head reduction.

Before I show how we deal with rest of the constructions in the formalisation I need to talk about how we parametrise the weak head reduction.

Reduction flags

In order to be able to fine-tune our reduction strategy we introduce the notion of reduction flags. Those mimic the flags that are used by Coq internally.

```
Module RedFlags.

  Record t := mk {
    beta   : bool ;
    iota   : bool ;
    zeta   : bool ;
    delta  : bool ;
    fix_   : bool ;
    cofix_ : bool
  }.

  Definition default := mk true true true true true true.

  Definition nodelta := mk true true true false true true.

End RedFlags.
```

The names refer to the reduction rules that they activate or not. Setting `beta` to `true` means that we will reduce β-redexes. `iota` is for the reduction of pattern-matching, `zeta` for the reduction of let-bindings and unfolding of local definitions, `delta` is for the unfolding of global definitions, `fix_` is for the unfolding of fixed-points and `cofix_` for that of cofixed-points.

The default is to reduce all those, but later we will also use a version which does not do δ-reductions to speed up conversion.

Coq specification

The reduction flags parameterise the weak head reduction and as such they affect the definition of weak head normal and neutral forms.

The global context and reduction flags are assumed globally in the definition using the commands `Context (flags : RedFlags.t).` and `Context (Σ : global_env).`

```
Inductive whnf Γ : term -> Prop :=
| whnf_ne t             : whne Γ t -> whnf Γ t
| whnf_sort s           : whnf Γ (tSort s)
| whnf_prod na A B      : whnf Γ (tProd na A B)
| whnf_lam na A B       : whnf Γ (tLambda na A B)
| whnf_cstrapp i n u v  : whnf Γ (mkApps (tConstruct i n u) v)
| whnf_indapp i u v     : whnf Γ (mkApps (tInd i u) v)
| whnf_cofix mfix idx   : whnf Γ (tCoFix mfix idx)
| whnf_fix_short mfix idx narg body args :
    unfold_fix mfix idx = Some (narg, body) ->
    nth_error args narg = None ->
    whnf Γ (mkApps (tFix mfix idx) args)

with whne Γ : term -> Prop :=
| whne_rel i :
    option_map decl_body (nth_error Γ i) = Some None ->
    whne Γ (tRel i)

| whne_rel_nozeta i :
    RedFlags.zeta flags = false ->
    whne Γ (tRel i)

| whne_letin_nozeta na B b t :
    RedFlags.zeta flags = false ->
    whne Γ (tLetIn na B b t)

| whne_const c u decl :
    lookup_env Σ c = Some (ConstantDecl decl) ->
    decl.(cst_body) = None ->
    whne Γ (tConst c u)

| whne_const_nodelta c u :
    RedFlags.delta flags = false ->
    whne Γ (tConst c u)

| whne_fix mfix idx narg body args a :
    unfold_fix mfix idx = Some (narg, body) ->
    nth_error args narg = Some a ->
    whne Γ a ->
    whne Γ (mkApps (tFix mfix idx) args)

| whne_fix_nofix_ mfix idx args :
    RedFlags.fix_ flags = false ->
    whne Γ (mkApps (tFix mfix idx) args)

| whne_cofix_nocofix_ mfix idx args :
    RedFlags.cofix_ flags = false ->
    whne Γ (mkApps (tCoFix mfix idx) args)

| whne_app f v :
    whne Γ f ->
    whne Γ (tApp f v)

| whne_case i p c brs :
    whne Γ c ->
    whne Γ (tCase i p c brs)

| whne_case_noiota i p c brs :
    RedFlags.iota flags = false ->
    whne Γ (tCase i p c brs)
```

```
| whne_proj p c :
    whne Γ c ->
    whne Γ (tProj p c)

| whne_proj_noiota p c :
    RedFlags.iota flags = false ->
    whne Γ (tProj p c).
```

Applied constructors and inductive types are normal, same as cofixed-points. A fixed-point can be normal even if applied to arguments, only the recursive argument must not be provided (because when the recursive argument is a constructor, the fixed-point can be unfolded). If the recursive argument is provided and is neutral then the fixed-point is neutral. It is also considered neutral when the corresponding flag `fix_` is set to `false`.

Variables are neutral if they refer to local definitions when ζ is deactivated, or if they refer to assumptions. Similarly, constants are neutral if they refer to global definitions when δ is off, or if they refer to an axiom. let-bindings are only neutral when ζ is off. A pattern-matching expression is neutral when the scrutinee is neutral or if the ι flag is off. The same goes for projections.

This definition can serve as a specification for the weak head reduction procedure we implement.

23.2 The reduction machine

We now have a clearer idea of what we wish to implement. As I already said, this is still not as straightforward as one would hope. As it is a bit more complex than an evaluation function taking a term and returning its weak head normal form, we talk about an *abstract machine*. This idea is of course not novel, even in the case of Coq which featured its own abstract machine called the constructive engine [91]. Abstract machines have been extensively studied since [92, 93].

[91]: Huet (1989), 'The Constructive Engine.'
[92]: Accattoli et al. (2017), 'The negligible and yet subtle cost of pattern matching'
[93]: Accattoli et al. (2017), 'Environments and the complexity of abstract machines'

Idea behind the machine

The idea behind the machine is to *focus and reduce*: whenever there is a redex at the top-level, we reduce it, and when we reach a term potentially stuck (say an application), we focus on a subterm and reduce it to see if we reach another redex.

This is often achieved by taking a list of arguments, *i.e.* a *stack*, besides the term. When considering an application, the argument is *pushed* on top of the stack and we *focus* on the left-hand term. I will write $\langle t \mid \pi \rangle$ for term t *against* stack π. We want to define weak head evaluation $\rightarrowtriangle_{\mathsf{W}}$ as something like this:

$$\frac{\Gamma \vdash \langle t \mid u :: \pi \rangle \rightarrowtriangle_{\mathsf{W}} a}{\Gamma \vdash \langle t\ u \mid \pi \rangle \rightarrowtriangle_{\mathsf{W}} a} \qquad \frac{\Gamma \vdash \langle t[x \leftarrow u] \mid \pi \rangle \rightarrowtriangle_{\mathsf{W}} a}{\Gamma \vdash \langle \lambda(x : A).t \mid u :: \pi \rangle \rightarrowtriangle_{\mathsf{W}} a}$$

$$\frac{\Gamma \vdash \mathsf{zip}\ \langle t \mid \pi \rangle\ \mathsf{whnf}}{\Gamma \vdash \langle t \mid \pi \rangle \rightarrowtriangle_{\mathsf{W}} \langle t \mid \pi \rangle}$$

Here $\langle t \mid \pi \rangle$ is the data of a term t versus a list of arguments π. zip takes the term and apply it to the stack.

The things above the line can be seen as recursive calls of the machine. This is rather informal, but it does not correspond to the final definition anyway.

Regarding termination, we see two cases: in the first we focus on a subterm, whereas in the second we do a recursive call on a *reduct*. In other words we need a well-founded order that satisfies

$$\begin{array}{rcl} \langle t \mid u :: \pi \rangle & \prec & \langle t\ u \mid \pi \rangle \\ \langle t[x \leftarrow u] \mid \pi \rangle & \prec & \langle \lambda(x : A).t \mid u :: \pi \rangle \end{array}$$

As we have seen, assuming strong normalisation is tantamount to having a well-founded order on coreduction, and it should not be surprising that the subterm relation is also well-founded. Taking a lexicographical order on both these orders seems like a good option.

Remember that lexicographical orders operate on pairs and that the first components must be *equal* when using the order on the second components. Thus, we must produce a pair from the data $\langle t \mid \pi \rangle$, coreduction is to be taken on the whole term (t applied to π), while the subterm relation only looks at t. This forces us to consider coreduction before the subterm relation, indeed if we were to consider the pair $(t, \mathsf{zip}\ \langle t \mid \pi \rangle)$ we would satisfy the first inequality as t is a subterm of $t\ u$, but in the second inequality the first components would not be equal. We will thus consider the following pair:

$$(\mathsf{zip}\ \langle t \mid \pi \rangle,\ t)$$

with the corresponding lexicographical order of coreduction and subterm relation. This one works because $\langle t \mid u :: \pi \rangle$ and $\langle t\ u \mid \pi \rangle$ represent the same term so that the first components are indeed equal when considering the subterm order.

$$\begin{array}{rl} (\mathsf{zip}\ \langle t \mid u :: \pi \rangle,\ t) & \prec \quad (\mathsf{zip}\ \langle t\ u \mid \pi \rangle,\ t\ u) \\ \text{since} & \mathsf{zip}\ \langle t \mid u :: \pi \rangle = \mathsf{zip}\ \langle t\ u \mid \pi \rangle \\ \text{and} & t \sqsubset t\ u \end{array}$$

I write $x \sqsubset y$ to mean that x is a subterm of y.

$$\begin{array}{rl} (\mathsf{zip}\ \langle t[x \leftarrow u] \mid \pi \rangle,\ t[x \leftarrow u]) & \prec \quad (\mathsf{zip}\ \langle \lambda x.t \mid u :: \pi \rangle,\ \lambda x.t) \\ \text{since} & \mathsf{zip}\ \langle \lambda x.t \mid u :: \pi \rangle \rightarrowtriangle \mathsf{zip}\ \langle t[x \leftarrow u] \mid \pi \rangle \end{array}$$

Problems arise however when considering other syntactic constructions of PCUIC, particularly pattern-matching and fixed-points. I will identify two such problems before finally moving on to the next section where I introduce the order we will actually use.

To understand where I am coming from here: stacks in the actual kernel are indeed just lists of arguments. It introduces an asymmetry with respect to the other computation rules. This behaviour is justified by my trying to stay as close as possible to the implementation (in particular when considering the extracted program) but can also be interpreted as a legacy design choice and

Problem 1: List of arguments as stacks will prove to be a problem when focusing on something else than a function. This is the case when reducing pattern-matching and projections. I will use projections on pairs as an example.

$$\frac{\Gamma \vdash \langle p \mid \bullet \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} \langle (u, v) \mid \bullet \rangle \qquad \Gamma \vdash \langle u \mid \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a}{\Gamma \vdash \langle p.1 \mid \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a}$$

The idea is that to reduce $p.1$ (potentially applied to arguments) we want to reduce p (this time it is not applied) to see if it is a constructor of the record, *i.e.* a pair. We now need

$$(\mathsf{zip}\ \langle p \mid \bullet \rangle,\ p) \prec (\mathsf{zip}\ \langle p.1 \mid \pi \rangle,\ p.1)$$

We also need

$$(\mathsf{zip}\ \langle u \mid \pi \rangle,\ u) \prec (\mathsf{zip}\ \langle p.1 \mid \pi \rangle,\ p.1)$$

It holds because u is a reduct of $p.1$. To know this we need to know that the reduction machine is correct while proving it terminates. I will say more about this later.

However p is not a reduct of $\mathsf{zip}\ \langle p.1 \mid \pi \rangle$, and is not equal to it so we cannot use the fact that p is a subterm of $p.1$. The solution I propose to this is to actually remember the full term we were considering before focusing. This is what lists of arguments were achieving when considering only λ-abstraction and application. For this we can use the stacks I introduced in Chapter 22 (Term positions and stacks).

$$\frac{\Gamma \vdash \langle t \mid \boxed{\square\ u} :: \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a}{\Gamma \vdash \langle t\ u \mid \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a} \qquad \frac{\Gamma \vdash \langle t[x \leftarrow u] \mid \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a}{\Gamma \vdash \langle \lambda (x : A).t \mid \boxed{\square\ u} :: \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a}$$

$$\frac{\Gamma \vdash \mathsf{zip}\ \langle t \mid \pi \rangle\ \mathsf{whnf}}{\Gamma \vdash \langle t \mid \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} \langle t \mid \pi \rangle}$$

Now $\mathsf{zip}\ \langle t \mid \pi \rangle$ shall refer to plugging t in stack π.

$$\frac{\Gamma \vdash \langle p \mid \boxed{\square.1} :: \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} \langle (u, v) \mid \boxed{\square.1} :: \pi \rangle \qquad \Gamma \vdash \langle u \mid \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a}{\Gamma \vdash \langle p.1 \mid \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a}$$

This time, as the stack is preserved, the first components are indeed the same and we have the inequality.

$$(\mathsf{zip}\ \langle p \mid \boxed{\square.1} :: \pi \rangle,\ p) \prec (\mathsf{zip}\ \langle p.1 \mid \pi \rangle,\ p.1)$$

Problem 2: Using stacks solves our first problem but there remains another when considering fixed-points this time. Indeed, fixed-points can only be unfolded when their recursive argument is a constructor. This means that when reducing a fixed-point we have to focus on the recursive argument which is *on* the stack:

$$\frac{\Gamma \vdash \langle u_n \mid \rho \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} \langle \mathsf{C} \mid \rho \rangle \qquad \Gamma \vdash \langle t[f \leftarrow \mathsf{fix}_n\, f.\ t] \mid \boxed{\square\ u_1} :: \cdots :: \boxed{\square\ u_{n-1}} :: \boxed{\square\ \mathsf{C}} :: \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a}{\Gamma \vdash \langle \mathsf{fix}_n\, f.\ t \mid \boxed{\square\ u_1} :: \cdots :: \boxed{\square\ u_{n-1}} :: \boxed{\square\ u_n} :: \pi \rangle \mathrel{-\!\triangleright}_{\mathsf{W}} a}$$

I here use non-mutual fixed-points to make things simpler. Remember here that n is the index of the recursive argument of the fixed-point.

where

$$\rho = \boxed{(\mathsf{fix}_n\, f.\ t)\ u_1\ \ldots\ u_{n-1}\ \square} :: \pi$$

The idea is that we fetch the n-th argument on the stack (assuming it exists) and we reduce it (while remembering that it was indeed the argument of a fixed-point). If this argument reduces to a constructor[1] then the fixed-point can safely be unfolded and reduced. For the recursive call to be valid we need:

1: In full generality, it would have to reduce to an *applied* constructor.

$$(\mathsf{zip}\ \langle u_n \mid \rho\rangle,\ u_n) \prec (\mathsf{zip}\ \langle \mathsf{fix}_n f.\ t \mid \theta\rangle,\ \mathsf{fix}_n f.\ t)$$

where

$$\theta = \boxed{\square\ u_1} :: \cdots :: \boxed{\square\ u_{n-1}} :: \boxed{\square\ u_n} :: \pi$$

Once again, the order fails us because, even though we manage to have the same term in both left components, we do not have that u_n is a subterm of $\mathsf{fix}_n f.\ t$. The subterm order is in fact ill-suited to deal with this problem and we need to replace it with a more flexible order. For this we will use *positions*.

The order

The order we define is on pairs of terms versus stacks $\langle t \mid \pi\rangle$. We map them to dependent pairs consisting of the full term that $\langle t \mid \pi\rangle$ represents (*i.e.* $\mathsf{zip}\ \langle t \mid \pi\rangle$), and the position corresponding to the stack π.

This was presented in Chapter 22 (Term positions and stacks).

I pointed out earlier that the well-founded order on positions corresponds more or less to the subterm order, but with a *twist*: 'going right', as I called it, is also decreasing for the order. As such, the position of an argument on the stack is smaller than the position of the term on the left, meaning that it makes the recursive call to the recursive argument *legal*.

In Coq we do it in two steps, defining first the order on pairs, then on commands (term versus stack).

```
Definition R_aux Γ :=
  dlexprod (cored Σ Γ) @posR.

Definition R Γ u v :=
  R_aux Γ (zip u ; stack_pos (fst u) (snd u))
          (zip v ; stack_pos (fst v) (snd v)).
```

`stack_pos` is like `stack_position` except it lands in `pos t` for some `t` and not in `position`.

Since the two involved orders are well-founded, their dependent lexicographical order is also well-founded.

See Chapter 21 (Well-founded induction and well-orders) for the definitions and properties of dependent lexicographical order.

```
Lemma R_Acc_aux :
  forall Γ t p,
    wf Σ ->
    wellformed Σ Γ t ->
    Acc (R_aux Γ) (t ; p).

Corollary R_Acc :
  forall Γ t,
    wf Σ ->
    wellformed Σ Γ (zip t) ->
    Acc (R Γ) t.
```

We also define the reflexive closure of R that we will use for correctness.

```
Definition Req Γ t t' :=
  t = t' \/ R Γ t t'.
```

Everything is ready for the definition of the reduction machine.

The implementation

Now that we have an order and a clear idea of what we want, we can write down the machine as a Coq function. I briefly pointed at one last issue however: in some cases, the argument to a recursive call is only smaller assuming that the result of another recursive call is indeed a reduct. Let us focus again on the recursive calls for projections:

$$\frac{\Gamma \vdash \langle p \mid \square.1 :: \pi \rangle \rightarrowtriangle_{\mathrm{W}} \langle (u,v) \mid \square.1 :: \pi \rangle \qquad \Gamma \vdash \langle u \mid \pi \rangle \rightarrowtriangle_{\mathrm{W}} a}{\Gamma \vdash \langle p.1 \mid \pi \rangle \rightarrowtriangle_{\mathrm{W}} a}$$

The second recursive call, made on u, is only valid because u is a reduct of $p.1$, an information that we get from the correction of the first recursive call.

The solution to this is to define the function and prove its correctness all at once.

When I say correctness, I do not include the fact that the machine indeed yields weak head normal forms. This is left as future work, as it will be important to the proof of completeness of the conversion algorithm.

```
reduce_stack :
  forall (Γ : context) (t : term) (π : stack),
    wellformed Σ Γ (zip (t,π)) ->
    { t' : term * stack | Req Σ Γ t' (t,π) }.
```

In other words, the function takes a term, a stack and returns a term that is a reduct (once zipped) in potentially zero steps. We actually need more than this but we have here the main ingredients.

The definition relies heavily on Equations to generate and fill obligations regarding correctness and the order. All in all, this definition is rather lengthy and I invite the interested reader to have a look at the details in the formalisation.

This reduction machine will prove useful in defining the conversion algorithm, but the way we built it, in particular the order, will prove useful as well.

24 Conversion

We want to implement a correct conversion checker that is close enough to the actual implementation that is rather efficient, instead of the naive solution of reducing both terms to normal forms and then comparing them syntactically. We also have to take care of universes and cumulativity which complicates matters a little bit.

24.1 High level description

I will present our approach at a high level.

(1) First we weak head reduce the two terms without δ-reduction (*i.e.* without unfolding definitions);
(2) then we compare their heads, if they match we begin again at (1);
(3) if they do not match, we check if some computation (pattern-matching or fixed-point)—or even the whole term—is blocked by a definition that could unfold to a value, and if so we unfold the definition and start again.

Following reduction, we actually use terms versus stacks to deal with focusing. Conversion is in fact split in four main phases that I codify as follows

- `Reduction` which corresponds to putting the two terms in weak head normal forms;
- `Term` which corresponds to comparing the heads of the focused terms;
- `Args` which corresponds to comparing the arguments on the stacks;
- `Fallback` which happens when comparing different heads and we attempt to reduce both sides a bit more.

These roughly correspond each to one of the mutual definitions that yield conversion. I will write these functions as judgements:

Here I forget about the typing requirements, but conversion, like reduction, can only operate on well-typed terms.

$$\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A$$

where A is the *answer* which is basically 'yes' or an error message. A is the *output* while everything else is the *input*. I will use the subscripts R, T, A and F to differentiate the four phases. I will present them in more depth so that we can see what are the requirements for termination.

`Reduction`

The first phase takes two pairs of a term and a stack and reduce both using the weak head reduction machine of Chapter 23 (Reduction).

In order to avoid unfolding definitions unnecessarily, we disable the δ-reduction. This means that we pass the `RedFlags.nodelta` flags to the machine.

I will once again describe these functions and their recursive calls using inference rules for simplicity.

$$\frac{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \twoheadrightarrow_{\mathsf{W}\not\delta} \langle u_1 \mid \rho_1 \rangle \qquad \Gamma \vdash \langle t_2 \mid \pi_2 \rangle \twoheadrightarrow_{\mathsf{W}\not\delta} \langle u_2 \mid \rho_2 \rangle \qquad \Gamma \vdash \langle u_1 \mid \rho_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle u_2 \mid \rho_2 \rangle \rightrightarrows A}{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A}$$

Here we simply do a recursive call on the weak head normal forms of the two inputs.

Term

For this phase, we take two inputs that are in weak head normal form (without δ-reduction) and look at the heads of the terms. The idea behind disabling δ is that when comparing two constants we hope not to have to unfold them if they already are the same.

$$\frac{u_1 \cong u_2 \qquad \Gamma \vdash \langle \mathsf{const}\ k\ u_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{A}} \langle \mathsf{const}\ k\ u_2 \mid \pi_2 \rangle \rightrightarrows \mathsf{yes}}{\Gamma \vdash \langle \mathsf{const}\ k\ u_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle \mathsf{const}\ k\ u_2 \mid \pi_2 \rangle \rightrightarrows \mathsf{yes}}$$

where const k u refers to the constant of name k and universe instance u. Here we first compare the universe instances to check that they are equal (which is a bit more general that syntactic equality), and if so we compare the stacks, or more specifically the arguments on those stacks. I will explain this part when presenting the corresponding function. If the constant names are not the same, or if the universe instances do not match or if the argument comparison fails, we then unfold one of the constants and compare them again.

One or the two of them might refer to axioms and in that case, we only unfold those that can be, and if we compare two axioms, we know they are distinct.

$$\frac{(k_2 := t_2) \in \Sigma \qquad \Gamma \vdash \langle \mathsf{const}\ k_1\ u_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle t_2[u_2] \mid \pi_2 \rangle \rightrightarrows A}{\Gamma \vdash \langle \mathsf{const}\ k_1\ u_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle \mathsf{const}\ k_2\ u_2 \mid \pi_2 \rangle \rightrightarrows A}$$

I informally write $t_2[u_2]$ to mean t_2 instantiated by u_2.

Another interesting case is the comparison of λ-abstractions.

$$\frac{\begin{array}{c}\Gamma \vdash \langle A_1 \mid \boxed{\lambda(x : \square).\ t_1} :: \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle A_2 \mid \boxed{\lambda(x : \square).\ t_2} :: \pi_2 \rangle \rightrightarrows \mathsf{yes} \\ \Gamma \vdash \langle t_1 \mid \boxed{\lambda(x : A_1).\ \square} :: \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle t_2 \mid \boxed{\lambda(x : A_2).\ \square} :: \pi_2 \rangle \rightrightarrows A\end{array}}{\Gamma \vdash \langle \lambda(x : A_1).\ t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle \lambda(x : A_2).\ t_2 \mid \pi_2 \rangle \rightrightarrows A}$$

Once again I show the rule when there are successes in each recursive calls, all the others yield errors.

First we compare the two domains and then we compare the two bodies. This may seem like a simple case of recursing on the subterms but there is a subtlety: the context when comparing the bodies is still Γ and not $\Gamma, x : A_1$ or $\Gamma, x : A_2$. This is actually a good thing that we do not have to pick one of them so that we can retain a symmetric presentation. We are saved by the stacks once more. Indeed, in $\boxed{\lambda(x : A_1).\ \square}$ we have the information corresponding to $x : A_1$. More

A *good* thing that helps us complete proofs.

generally, stacks do not only yield positions but also contexts. In the expression

$$\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A$$

t_1 lives in $\Gamma, \mathsf{stack\_context}\ \pi_1$, and t_2 in $\Gamma, \mathsf{stack\_context}\ \pi_2$.

Other than those, we also deal with Π-types, pattern-matching, projections, fixed- and cofixed-points. The case of two applications is impossible because the arguments would have been pushed on the stacks. If the terms do not fall in this diagonal, we instead call the fallback function recursively.

$$\frac{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{F}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A}{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A}$$

Args

When the two terms are convertible, it is time to move on the comparison of the stacks, or rather the comparison of the leading arguments on said stacks.

A stack π can be decomposed into a list of arguments $u_1, \dots, u_n$ and a remaining stack ρ that does not start with some $\boxed{\square\ u}$, such that

$$\pi = \boxed{\square\ u_1} :: \cdots :: \boxed{\square\ u_n} :: \rho$$

With two such decomposed stacks, the meaning of the `Args` function is the following

Notice how we have n arguments on both sides, if there is a mismatch, the function will return an error.

$$\frac{\begin{array}{c}\Gamma \vdash \langle u_1 \mid \boxed{t_1\ \square\ u_2\ \dots u_n} :: \rho_1 \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle v_1 \mid \boxed{t_2\ \square\ v_2\ \dots v_n} :: \rho_2 \rangle \rightrightarrows \text{yes} \\ \vdots \\ \Gamma \vdash \langle u_n \mid \boxed{t_1\ u_1\ \dots u_{n-1}\ \square} :: \rho_1 \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle v_n \mid \boxed{t_2\ v_1\ \dots v_{n-1}\ \square} :: \rho_2 \rangle \rightrightarrows \text{yes}\end{array}}{\Gamma \vdash \langle t_1 \mid \boxed{\square\ u_1} :: \cdots :: \boxed{\square\ u_n} :: \rho_1 \rangle \overset{?}{\equiv}_{\mathsf{A}} \langle t_2 \mid \boxed{\square\ v_1} :: \cdots :: \boxed{\square\ v_n} :: \rho_2 \rangle \rightrightarrows A}$$

Assuming that $\Gamma \vdash u_1 \equiv u_2$, it saying 'yes' means that

$$\Gamma \vdash t_1\ u_1\ \dots\ u_n \equiv t_2\ v_1\ \dots\ v_n$$

(and also the stronger property that each u_i is convertible to v_i but the property above is sufficient for our needs).

Fallback

The fallback deals with terms outside of the diagonal that are potentially stuck because of definitions that are not unfolded. It tries to make progress on each side before resorting to plain and simple syntactic equality.

$$\frac{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \rightharpoonup_{\mathsf{W}\neq} \langle u \mid \rho \rangle \qquad \Gamma \vdash \langle u \mid \rho \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A}{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{F}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A}$$

I use $\rightharpoonup_{\mathsf{W}\neq}$ to mean that the reduction produced a different term. In the formalisation it is a different process that checks that the term is indeed stuck because of δ.

$$\frac{\Gamma \vdash \langle t_2 \mid \pi_2 \rangle \rightharpoonup_{\mathsf{W}\neq} \langle u \mid \rho \rangle \qquad \Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle u \mid \rho \rangle \rightrightarrows A}{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{F}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A}$$

$$\frac{t_1 = t_2 \qquad \Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{A}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A}{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{F}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A}$$

Here $t_1 = t_2$ means that t_1 and t_2 have been checked to be syntactically equal, up to universes.

About cumulativity and universes

In the (partial) definitions above I sometimes refer to syntactic equality of terms or equality of universes. When I write $t_1 = t_2$ I mean syntactic equality of terms up to universes. This implicitly refers to universe constraints, which is the reason I do not simply unify them the way I unified k_1 and k_2 into k when considering constants.

Another important point is that we actually define conversion and cumulativity at the same time, using an extra argument (essentially a boolean) telling us which of those we are checking for. I did everything in the setting of conversion to have lighter notations, but everything works just as well with cumulativity.

I will not explain how we specify and check equality or cumulativity of universes as it was not my own contribution. The interested reader should refer either to [4] or the formalisation.

[4]: Sozeau et al. (2020), 'Coq Coq correct! verification of type checking and erasure for Coq, in Coq'

24.2 Termination and correctness

Specification

You will perhaps find my presentation surprising in that I did not even specify what was expected of the algorithm I was showing. I wanted to focus on the intuition before going into these details. It is time to correct this slight.

All the 'judgements' I presented earlier, that is

$$\begin{gathered}
\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A \\
\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A \\
\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{A}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A \\
\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{F}} \langle t_2 \mid \pi_2 \rangle \rightrightarrows A
\end{gathered}$$

are correct if, when they return yes, we have

$$\Gamma \vdash t_1\ u_1\ \dots\ u_n \equiv t_2\ v_1\ \dots\ v_n$$

where π_1 and π_2 are decomposed as

$$\begin{aligned} \pi_1 &= \boxed{\square \; u_1} :: \cdots :: \boxed{\square \; u_n} :: \rho_1 \\ \pi_2 &= \boxed{\square \; v_1} :: \cdots :: \boxed{\square \; v_n} :: \rho_2 \end{aligned}$$

This might come as surprising that we only conclude on applications at the head of the stacks and not about the whole stacks themselves. If we were asking for

$$\Gamma \vdash \mathsf{zip}\ \langle t_1 \mid \pi_1 \rangle \equiv \mathsf{zip}\ \langle t_2 \mid \pi_2 \rangle$$

instead, it would be too strong a requirement. Indeed, in the case of λ-abstractions, we consider the recursive call on the domains

$$\Gamma \vdash \langle A_1 \mid \boxed{\lambda(x : \square).\ t_1} :: \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle A_2 \mid \boxed{\lambda(x : \square).\ t_2} :: \pi_2 \rangle \rightrightarrows \mathsf{yes}$$

Note that zip $\langle A \mid \boxed{\lambda(x : \square).t} :: \pi \rangle$ is equal to zip $\langle \lambda(x : A).t \mid \pi \rangle$.

which we want to conclude on A_1 and A_2, not on the full term, otherwise the recursive call on the bodies becomes superfluous. This, I think, is a sign, that this approach—concluding on the whole term, zipped with its stack—is doomed to fail.

Going back to the bigger picture, it is also not a problem that we conclude only on a portion of the stack since the conversion algorithm will be called with empty stacks at the start.

$$\Gamma \vdash \langle t_1 \mid \varepsilon \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle t_2 \mid \varepsilon \rangle \rightrightarrows A$$

As you can see we also call the `Reduction` phase first as well.

We can now move on to the main difficulty regarding conversion: its termination.

Termination

You can see that the definition of the conversion algorithm is reminiscent of that of weak head reduction, and one might expect the termination orders to be relatively close. This is indeed the case, but things are slightly more complicated here, mainly for two reasons: we sometimes consider terms up to universes; and we have mutually defined functions.

I will go over the different kind of recursive calls to motivate the order, before introducing it.

The first rule I presented involves putting both terms in weak head normal form before recursing on them. As we know from the correctness of weak head reduction, the terms it yields are reducts of the original terms, and some focusing potentially happened. As such, the order used by reduction seems like a good option, save for one problem: in some cases—when the term is already in weak head normal form—the weak reduction returns the original term. Because of this, the phases must be part of the order. In particular we want to have

$$\mathsf{T} \prec \mathsf{R}$$

Similarly, when comparing constants in the `Term` phase, we have a recursive call to the `Args` phase with the same arguments so we also want

$$\mathsf{A} \prec \mathsf{T}$$

When we unfold one of the constants however, we go back from the `Term` phase to the `Reduction` phase so we really have to account for the fact that the unfolded definition is a reduct of the constant, and it has to come before in the lexicographical order.

The case of λ-abstractions tells us that similarly, the subterm relation should come before the phase relation.

The `Fallback` phase also happens with the same arguments as that of the `Term` phase so

$$\mathsf{F} \prec \mathsf{T}$$

The `Args` phase makes recursive calls inside the stack meaning that, like weak head reduction, we must in fact use positions rather than the subterm relation.

Finally, the `Fallback` phase can make recursive calls on the `Args` phase with the same arguments, meaning

$$\mathsf{A} \prec \mathsf{F}$$

The order. Let me now introduce the order that we use to prove conversion terminating. First we need to define the notion of phases (that I call states in the formalisation):

```
Inductive state :=
| Reduction
| Term
| Args
| Fallback.
```

as well as a well-founded order on them

```
Inductive stateR : state -> state -> Prop :=
| stateR_Term_Reduction : stateR Term     Reduction
| stateR_Args_Term      : stateR Args     Term
| stateR_Fallback_Term  : stateR Fallback Term
| stateR_Args_Fallback  : stateR Args     Fallback.
```

It should not prove too hard to see that it is indeed well-founded since we essentially have

$$\mathsf{A} \prec \mathsf{F} \prec \mathsf{T} \prec \mathsf{R}$$

I already said we needed to consider equality of terms up to universes, as such we define an altered version of coreduction that is coreduction up to universes.

```
Notation eqt u v :=
  (‖ eq_term (global_ext_constraints Σ) u v ‖).

Definition cored' Γ u v :=
  exists u' v', cored Σ Γ u' v' /\ eqt u u' /\ eqt v v'.
```

The notation ‖ A ‖ is for the *squashed* version of A, *i.e.* its embedding into **Prop**. This is to make sure it goes away after extraction.

Because coreduction is only well-founded for well-formed terms, I also define the type of well-formed terms and coreduction on them.

```
Definition wterm Γ :=
  { t : term | wellformed Σ Γ t }.

Definition wcored Γ (u v : wterm Γ) :=
  cored' Σ Γ (` u) (` v).
```

` w is the first projection of w, *i.e.* a term, forgetting about its well-formedness proof.

```
Definition weqt {Γ} (u v : wterm Γ) :=
  eqt (` u) (` v).
```

We can finally move on to the definition of the (auxiliary) order:

```
Equations R_aux (Γ : context) :
  (Σ t : term, pos t × (Σ w : wterm Γ, pos (` w) × state)) ->
  (Σ t : term, pos t × (Σ w : wterm Γ, pos (` w) × state)) ->
  Prop :=
  R_aux Γ :=
    t ⊨ eqt \ cored' Σ Γ by _ ⊗
    @posR t ⊗
    w ⊨ weqt \ wcored Γ by _ ⊗
    @posR (` w) ⊗
    stateR.
```

Here I use **Equations** which allows me to leave the coercions blank and provide them using tactics.

So what we have here are nested dependent lexicographical orders modulo syntactical equality up to universes. The order operates on a term, a position in it, another term and position, and finally a state, using coreduction for terms.

The real order simply repacks this information, computing the positions from the stacks.

The order is well-founded and sufficient to prove termination of the conversion algorithm which is also correct by construction. In the end we get a conversion checker that operates on well-formed terms and is proven sound:

```
Theorem isconv_term_sound :
  forall Γ leq t1 h1 t2 h2,
    isconv_term Γ leq t1 h1 t2 h2 = Success I ->
    conv_cum leq Σ Γ t1 t2.
```

where `leq` is the flag saying whether we compare for cumulativity or conversion, and `conv_cum leq` is either cumulativity or conversion.

24.3 Future work: η-conversion

In the Coq implementation, η-expansion is handled even though the conversion is untyped. This method is described in [94, 95]. The way this is achieved is by only expanding a term (which is not a λ-abstraction) when it is compared to a λ-abstraction.

[94]: Coquand (1991), 'An algorithm for testing conversion in type theory'
[95]: Goguen (2005), 'Justifying algorithms for $\beta\eta$-conversion'

$$\frac{\Gamma \vdash \langle \lambda(x : A).\ t_1\ x \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle \lambda(x : A).\ u \mid \pi_2 \rangle \rightrightarrows A}{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle \lambda(x : A).\ u \mid \pi_2 \rangle \rightrightarrows A}$$

The recursive call would thus be followed by the congruence rule for λ-abstractions. For termination purposes this step is actually skipped and we directly do

$$\frac{\Gamma \vdash \langle t_1\ x \mid \boxed{\lambda(x : A).\ \square} :: \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{R}} \langle u \mid \boxed{\lambda(x : A).\ \square} :: \pi_2 \rangle \rightrightarrows A}{\Gamma \vdash \langle t_1 \mid \pi_1 \rangle \overset{?}{\equiv}_{\mathsf{T}} \langle \lambda(x : A).\ u \mid \pi_2 \rangle \rightrightarrows A}$$

We also take advantage of this to skip the comparison of the domains since they are both necessarily the same.

In our setting however, we would need to have the order not only modulo universes but also modulo η, and I have not yet managed to find a way to account for η in a way that preserves well-foundedness. As unfortunate as it is—it gets in the way of actually using our type-checker on real-life examples—it will have to be future work for us.

Type inference and checking 25

Now that we have our conversion algorithm, we can move on to type-checking. This work is not mine but that of Simon Boulier so I will go over it briefly. It needs to be mentioned as it is the goal of both the weak head reduction and the conversion algorithms.

Inference versus checking. In Coq, without existential variables, type inference is decidable. That is, given a term of PCUIC in a well-formed environment Γ, there is an algorithm that decides whether there exists a type A such that

$$\Gamma \vdash t : A$$

Checking that t has type B is then implemented by inferring the type of A and verifying that it is a subtype of B.

This implies that the inference algorithm should not return any type, but the principal type of t.

The good news is that this time, termination is structural so it does not require extra work like we had for reduction and conversion. Inference looks like this

```
Program Fixpoint infer
  (Γ : context) (HΓ : ∥ wf_local Σ Γ ∥) (t : term) {struct t}
  : typing_result ({ A : term & ∥ Σ ;;; Γ |- t : A ∥ }) :=
  match t with
  | tRel n =>
    match nth_error Γ n with
    | Some c => ret ((lift0 (S n)) (decl_type c) ; _)
    | None   => raise (UnboundRel n)
    end

  | tSort u =>
    match Universe.get_is_level u with
    | Some l =>
      check_eq_true
        (LevelSet.mem l (global_ext_levels Σ))
        (Msg ("undeclared level " ++ string_of_level l)) ;;
      ret (tSort (Universe.super l) ; _)
    | None  =>
      raise (Msg (string_of_sort u ++ " is not a level"))
    end

  | tProd na A B =>
    s1 <- infer_type infer Γ HΓ A ;;
    s2 <- infer_type infer (Γ ,, vass na A) _ B ;;
    ret (tSort (Universe.sort_of_product s1.π1 s2.π1) ; _)

  | tLambda na A t =>
    s1 <- infer_type infer Γ HΓ A ;;
    B <- infer (Γ ,, vass na A) _ t ;;
    ret (tProd na A B.π1 ; _)

  | tLetIn n b b_ty b' =>
    infer_type infer Γ HΓ b_ty ;;
    infer_cumul infer Γ HΓ b b_ty _ ;;
    b'_ty <- infer (Γ ,, vdef n b b_ty) _ b' ;;
    ret (tLetIn n b b_ty b'_ty.π1 ; _)

  | tApp t u =>
```

```
    ty <- infer Γ HΓ t ;;
    pi <- reduce_to_prod Γ ty.π1 _ ;;
    infer_cumul infer Γ HΓ u pi.π2.π1 _ ;;
    ret (subst10 u pi.π2.π2.π1 ; _)

  | tConst cst u =>
    match lookup_env (fst Σ) cst with
    | Some (ConstantDecl d) =>
      check_consistent_instance d.(cst_universes) u ;;
      let ty := subst_instance_constr u d.(cst_type) in
      ret (ty ; _)
    |  _ => raise (UndeclaredConstant cst)
    end

  (* ... *)

  end.
```

We use Program this time to allow us to solve some parts using tactics as, once again, the function is defined to be correct by construction. The function is written in monadic style, using the following monad

```
Inductive typing_result (A : Type) :=
| Checked (a : A)
| TypeError (t : type_error).
```

which corresponds to either returning a type or an error message.

We can then decide checking

```
Program Definition infer_cumul Γ HΓ t A (hA : wellformed Σ Γ A)
  : typing_result (‖ Σ ;;; Γ |- t : A ‖) :=
  A' <- infer Γ HΓ t ;;
  X <- convert_leq Γ A'.π1 A _ _ ;;
  ret _.
```

I do not expect you to read the definition in detail. For a variable it checks if it is bound, and if so returns the type it is assigned in the environment, otherwise it raises an error. For a sort, it checks if it is algebraic, if not and it is a declared level, it returns the successor level. For a Π-type it infers the types of the domain and codomain, checking that they are indeed sorts, and returns the sort of the product (*i.e.* the maximum of the two sorts if the second one is not Prop, and Prop otherwise). In the case of application we use `infer_cumul` to *check* rather than infer the type of the argument. The checking function is defined afterwards so this is another version of it (as you can see it takes `infer` as argument).

All this is as one should expect, and concludes our quest of a type-checker defined in Coq and proven correct while relying only on a few axioms stating that the theory is well behaved.

Conclusions

Elimination of reflection

In this document (and in the orginal paper [3]) we provide the first effective translation from ETT to ITT with UIP and funext. The translation has been formalised in Coq. This translation is also effective in the sense that we can produce in the end a Coq term using the MetaCoq denotation machinery. With ongoing work to extend the translation to the inductive fragment of Coq, we are paving the way to an extensional version of the Coq proof assistant which could be translated back to its intensional version, allowing the user to navigate between the two modes, and in the end produce a proof term checkable in the intensional fragment.

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

This thesis also introduces a translation from ETT to WTT which thus yields a translation from ITT to WTT. This shows that computation is not *essential* to the logical power of a type theory.

Perspectives

The translation from ETT is effective, but not particularly efficient. This is mainly due to the fact that derivations in ETT are really big, due to a lot of redundant information. I think this calls for an intermediary between the term and its typing derivation on which to perform the translation.

On another note, I believe we can improve ETT so that it is more practical, by reconciling it with computation. I believe (at least hope) that this can be achieved by using a weaker notion of irrelevant reflection that would only allow reflection to happen in specific places like the indices of inductive types, provided said indices are irrelevant for computation (one should not be able to get the length of a vector by simply matching the head but needs to recurse in the tail, same as for lists). It would be interesting how it would affect the translation I presented.

A verified type-checker for **Coq**, in **Coq**

We have formalised an almost feature-complete type-checker for Coq in Coq and proven it correct. Thanks to the extraction mechanism of Coq, this can actually be turned into an independent type-checker program. For instance it can be run within Coq in the manner of a plugin:

```
MetaCoq SafeCheck nat.
```

Because we do not deal with template polymorphism, the module system and η-expansion, we are not yet able to type-check the standard library or the formalisation itself (as it relies on the standard library). We were however able to test it on reasonable proof terms coming from the HoTT library [46]. For instance, we have been able to type-check the proof that an isomorphism can be turned into an equivalence. It currently is about one order of magnitude slower than the Coq implementation (0.015s vs 0.002s averaged over 10 runs for each checking). This can in particular be attributed to our very inefficient representation of global environments as association lists indexed by character lists, where Coq uses efficient hash-maps on strings.

[46]: Bauer et al. (2017), 'The HoTT library: a formalization of homotopy type theory in Coq'

Examples of the checker in action, including the HoTT theorems can be found in the file test-suite/safechecker_test.v.

Another point I would like to address is the lack of completeness proof. Our tests suggest that we are close to completeness if not there yet for the fragment we are considering. It would be however really interesting to have a proof that conversion and inference are complete with respect to their specifications. At the moment, dealing with η-conversion seems like a much more pressing matter as it would make the checker usable in practice at least as a standalone independent checker.

There is also the long term dream of having a Coq kernel fully verified and running in Coq.

Bibliography

Here are the references in citation order.

[1] Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. 'The Next 700 Syntactical Models of Type Theory'. In: *Certified Programs and Proofs – CPP 2017*. Jan. 2017, pp. 182–194 (cited on pages iv, 89–91, 99).

[2] The Coq development team. *The Coq proof assistant reference manual*. Version 8.11. LogiCal Project. 2020 (cited on pages v, xi, 1, 7, 16).

[3] Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. 'Eliminating Reflection from Type Theory'. In: *CPP 2019 - 8th ACM SIGPLAN International Conference on Certified Programs and Proofs*. Lisbonne, Portugal: ACM, Jan. 2019, pp. 91–103. DOI: 10.1145/3293880.3294095 (cited on pages v, 1, 96, 179).

[4] Matthieu Sozeau et al. 'Coq Coq correct! verification of type checking and erasure for Coq, in Coq'. In: *Proceedings of the ACM on Programming Languages* 4.POPL (2020), pp. 1–28 (cited on pages v, 1, 125, 128, 130, 143, 145, 172).

[5] Matthieu Sozeau et al. 'The MetaCoq Project'. In: *Journal of Automated Reasoning* (Feb. 2020). DOI: 10.1007/s10817-019-09540-0 (cited on pages v, 1).

[6] Andrej Bauer, Philipp G. Haselwarter, and Théo Winterhalter. *Formalising Type Theory in a modular way for translations between type theories*. 2016. URL: https://github.com/TheoWinterhalter/formal-type-theory (cited on pages v, 1, 77, 84).

[7] Ulf Norell. *Towards a practical programming language based on dependent type theory*. Vol. 32. Citeseer, 2007 (cited on pages xi, 7, 16, 45).

[8] NG De Bruijn. 'A plea for weaker frameworks'. In: *Logical frameworks* (1991), pp. 40–67 (cited on pages 12, 47).

[9] Gerhard Gentzen. 'Untersuchungen über das logische Schließen. I'. In: *Mathematische zeitschrift* 39.1 (1935), pp. 176–210 (cited on page 14).

[10] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002 (cited on page 16).

[11] Kurt Gödel. 'Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I'. In: *Monatshefte für mathematik und physik* 38.1 (1931), pp. 173–198 (cited on page 17).

[12] Raphael M Robinson. 'An essentially undecidable axiom system'. In: *Proceedings of the international Congress of Mathematics*. Vol. 1. American Mathematical Society 80 Waterman Street, Providence, RI. 1950, pp. 729–730 (cited on page 17).

[13] A Bezboruah and John C. Shepherdson. 'Gödel's second incompleteness theorem for Q'. In: *The Journal of Symbolic Logic* 41.2 (1976), pp. 503–512 (cited on page 17).

[14] Felice Cardone and J Roger Hindley. 'History of lambda-calculus and combinatory logic'. In: *Handbook of the History of Logic* 5 (2006), pp. 723–817 (cited on page 19).

[15] Robin Milner. 'A theory of type polymorphism in programming'. In: *Journal of computer and system sciences* 17.3 (1978), pp. 348–375 (cited on page 21).

[16] M Zakrzewski. 'Definable functions in the simply typed lambda-calculus'. In: *TLCA*. cs. LO/0701022. cs/0701022. 2007 (cited on page 23).

[17] András Kovács and Ambrus Kaposi. 'Signatures and Induction Principles for Higher Inductive-Inductive Types'. In: *Logical Methods in Computer Science* 16 (2020) (cited on page 34).

[18] Eduardo Giménez. 'Structural recursive definitions in type theory'. In: *International Colloquium on Automata, Languages, and Programming*. Springer. 1998, pp. 397–408 (cited on page 34).

[19] Eduarde Giménez. 'Codifying guarded definitions with recursive schemes'. In: *International Workshop on Types for Proofs and Programs*. Springer. 1994, pp. 39–59 (cited on page 34).

[20] Matthieu Sozeau. 'Equations: A Dependent Pattern-Matching Compiler'. In: *ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*. Ed. by Matt Kaufmann and Lawrence C. Paulson. Vol. 6172. Lecture Notes in Computer Science. Springer, 2010, pp. 419–434 (cited on pages 38, 118).

[21] Matthieu Sozeau and Cyprien Mangin. 'Equations reloaded: high-level dependently-typed functional programming and proving in Coq'. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–29 (cited on pages 38, 118).

[22] Fredrik Nordvall Forsberg and Anton Setzer. 'Inductive-inductive definitions'. In: *International Workshop on Computer Science Logic*. Springer. 2010, pp. 454–468 (cited on page 39).

[23] Peter Dybjer. 'A general formulation of simultaneous inductive-recursive definitions in type theory'. In: *The Journal of Symbolic Logic* 65.2 (2000), pp. 525–549 (cited on page 39).

[24] Andreas Abel et al. 'Copatterns: programming infinite structures by observations'. In: *ACM SIGPLAN Notices* 48.1 (2013), pp. 27–38 (cited on page 41).

[25] Jesper Cockx, Dominique Devriese, and Frank Piessens. 'Eliminating dependent pattern matching without K'. In: *Journal of functional programming* 26 (2016) (cited on page 43).

[26] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. 'Observational equality, now!' In: *Proceedings of the 2007 workshop on Programming languages meets program verification*. ACM. 2007, pp. 57–68 (cited on pages 44, 51, 125).

[27] Thorsten Altenkirch et al. 'Setoid Type Theory—A Syntactic Translation'. In: *International Conference on Mathematics of Program Construction*. Springer. 2019, pp. 155–196 (cited on pages 44, 51).

[28] Marc Bezem, Thierry Coquand, and Simon Huber. 'A Model of Type Theory in Cubical Sets'. In: (Dec. 2013) (cited on pages 44, 52, 99).

[29] Cyril Cohen et al. 'Cubical type theory: a constructive interpretation of the univalence axiom'. In: *arXiv preprint arXiv:1611.02108* (2016) (cited on pages 44, 52).

[30] Conor McBride. 'Dependently typed functional programs and their proofs'. PhD thesis. University of Edinburgh, 2000 (cited on page 44).

[31] Philipp G. Haselwarter. 'Effective Metatheory for Type Theory'. PhD thesis. University of Ljubljana, 2020 (cited on page 45).

[32] Henk Barendregt. 'Introduction to generalized type systems'. In: *Journal of functional programming* 1.2 (1991), pp. 125–154 (cited on page 45).

[33] Floris van Doorn, Herman Geuvers, and Freek Wiedijk. 'Explicit convertibility proofs in pure type systems'. In: *Proceedings of the Eighth ACM SIGPLAN international workshop on Logical frameworks & meta-languages: theory & practice*. ACM. 2013, pp. 25–36 (cited on pages 46, 86, 126).

[34] Andrej Bauer et al. *The 'Andromeda' prover*. 2016. URL: `http://www.andromeda-prover.org/` (cited on pages 47, 123).

[35] Robert L. Constable and Joseph L. Bates. *The NuPrl system, PRL project*. 2014. URL: `http://www.nuprl.org/` (cited on page 47).

[36] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. 2004 (cited on page 48).

[37] Jean-Yves Girard. 'Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur'. PhD thesis. Éditeur inconnu, 1972 (cited on page 48).

[38] Antonius JC Hurkens. 'A simplification of Girard's paradox'. In: *International Conference on Typed Lambda Calculi and Applications*. Springer. 1995, pp. 266–278 (cited on page 48).

[39] Robert Harper and Robert Pollack. 'Type checking with universes'. In: *Theoretical computer science* 89.1 (1991), pp. 107–136 (cited on page 48).

[40] Gaëtan Gilbert et al. ‘Definitional Proof-Irrelevance without K’. In: *Proceedings of the ACM on Programming Languages*. POPL’19 (Jan. 2019), pp. 1–28. DOI: 10.1145/329031610.1145/3290316 (cited on page 48).

[41] Matthieu Sozeau and Nicolas Tabareau. ‘Universe polymorphism in Coq’. In: *International Conference on Interactive Theorem Proving*. Springer. 2014, pp. 499–514 (cited on page 49).

[42] Amin Timany and Matthieu Sozeau. ‘Cumulative inductive types in Coq’. In: *LIPIcs: Leibniz International Proceedings in Informatics* (2018) (cited on page 49).

[43] Thorsten Altenkirch. ‘Extensional equality in intensional type theory’. In: *Proceedings. 14th Symposium on Logic in Computer Science (Cat. No. PR00158)*. IEEE. 1999, pp. 412–420 (cited on page 51).

[44] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013 (cited on page 51).

[45] Nicolas Tabareau, Éric Tanter, and Matthieu Sozeau. ‘Equivalences for free: univalent parametricity for effective transport’. In: *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), pp. 1–29 (cited on page 51).

[46] Andrej Bauer et al. ‘The HoTT library: a formalization of homotopy type theory in Coq’. In: *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*. 2017, pp. 164–172 (cited on pages 51, 180).

[47] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. ‘Cubical agda: a dependently typed programming language with univalence and higher inductive types’. In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), pp. 1–29 (cited on page 52).

[48] Vladimir Voevodsky. *A simple type system with two identity types*. 2013. URL: https://ncatlab.org/homotopytypetheory/files/HTS.pdf (cited on page 52).

[49] Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. ‘Extending Homotopy Type Theory with Strict Equality’. In: (2016) (cited on page 52).

[50] Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. ‘Two-Level Type Theory and Applications’. In: *CoRR* abs/1705.03307 (2017) (cited on page 52).

[51] Thorsten Altenkirch and Ambrus Kaposi. ‘Type theory in type theory using quotient inductive types’. In: *ACM SIGPLAN Notices* 51.1 (2016), pp. 18–29 (cited on pages 52, 86).

[52] Pierre-Marie Pédrot et al. ‘A Reasonably Exceptional Type Theory’. In: *Proceedings of the ACM on Programming Languages*. Issue ICFP 3 (Aug. 2019), pp. 1–29. DOI: 10.1145/3341712 (cited on page 58).

[53] Pierre-Marie Pédrot and Nicolas Tabareau. ‘The Fire Triangle’. In: *Proceedings of the ACM on Programming Languages* (Jan. 2020), pp. 1–28. DOI: 10.1145/3371126 (cited on page 58).

[54] Alexandre Miquel. ‘Le Calcul des Constructions implicite: syntaxe et sémantique’. In: *These de doctorat, Université Paris 7* (2001) (cited on page 61).

[55] Peter Aczel. ‘On relating type theories and set theories’. In: *International Workshop on Types for Proofs and Programs*. Springer. 1998, pp. 1–18 (cited on pages 61, 62).

[56] Hongwei Xi. ‘Imperative programming with dependent types’. In: *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No. 99CB36332)*. IEEE. 2000, pp. 375–387 (cited on page 61).

[57] Benjamin Werner. ‘Sets in types, types in sets’. In: *International Symposium on Theoretical Aspects of Computer Software*. Springer. 1997, pp. 530–546 (cited on page 61).

[58] Thierry Coquand. ‘Metamathematical investigations of a calculus of constructions’. In: (1989) (cited on page 61).

[59] P Dybjer. ‘Inductive sets and families in Martin-L of type theory and their set-theoretic semantics’. In: *Proceedings of the First Workshop on Logical Frameworks*, pp. 280–306 (cited on page 61).

[60] Douglas J Howe. ‘On computational open-endedness in Martin-Lof’s type theory’. In: *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE. 1991, pp. 162–172 (cited on page 61).

[61] Amin Timany and Matthieu Sozeau. *Consistency of the Predicative Calculus of Cumulative Inductive Constructions (pCuIC)*. Research Report RR-9105. Version 2 fixes some typos from version 1.Version 3 fixes a typo in a typing rule from version 2. KU Leuven, Belgium ; Inria Paris, Oct. 2017, p. 30 (cited on page 62).

[62] John Cartmell. ‘Generalised algebraic theories and contextual categories’. In: *Annals of pure and applied logic* 32 (1986), pp. 209–243 (cited on page 64).

[63] Martin Hofmann. ‘On the interpretation of type theory in locally cartesian closed categories’. In: *International Workshop on Computer Science Logic*. Springer. 1994, pp. 427–441 (cited on page 64).

[64] Vladimir Voevodsky. ‘Subsystems and regular quotients of C-systems’. In: *Conference on Mathematics and its Applications,(Kuwait City, 2014)*. 658. 2016, pp. 127–137 (cited on page 64).

[65] Peter Dybjer. ‘Internal type theory’. In: *International Workshop on Types for Proofs and Programs*. Springer. 1995, pp. 120–134 (cited on page 64).

[66] Martin Hofmann. ‘Syntax and semantics of dependent types’. In: *Extensional Constructs in Intensional Type Theory*. Springer, 1997, pp. 13–54 (cited on page 64).

[67] Simon Pierre Boulier. ‘Extending type theory with syntactic models’. PhD thesis. 2018 (cited on pages 74, 89–91).

[68] Andrew M Pitts. ‘Nominal logic: A first order theory of names and binding’. In: *International Symposium on Theoretical Aspects of Computer Software*. Springer. 2001, pp. 219–242 (cited on page 78).

[69] Frank Pfenning and Conal Elliott. ‘Higher-order abstract syntax’. In: *ACM sigplan notices* 23.7 (1988), pp. 199–208 (cited on page 78).

[70] NG De Bruijn. ‘Lambda calculus with namefree formulas involving symbols that represent reference transforming mappings’. In: *Indagationes Mathematicae (Proceedings)*. Vol. 81. 1. North-Holland. 1978, pp. 348–356 (cited on page 78).

[71] Martin Abadi et al. ‘Explicit substitutions’. In: *Journal of functional programming* 1.4 (1991), pp. 375–416 (cited on page 79).

[72] Jean-philippe Bernardy, Patrik Jansson, and Ross Paterson. ‘Proofs for free: Parametricity for dependent types’. In: *Journal of Functional Programming* 22.2 (2012), pp. 107–152 (cited on pages 92, 99, 111).

[73] Guilhem Jaber et al. ‘The definitional side of the forcing’. In: *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*. 2016, pp. 367–376 (cited on page 92).

[74] Martin Hofmann and Thomas Streicher. ‘The Groupoid Interpretation of Type Theory’. In: *Twenty-five years of constructive type theory (Venice, 1995)*. Vol. 36. New York: Oxford Univ. Press, 1998, pp. 83–111 (cited on page 99).

[75] Chris Kapulkin and Peter LeFanu Lumsdaine. ‘The simplicial model of univalent foundations’. In: *arXiv preprint arXiv:1211.2851* (2012) (cited on page 99).

[76] Nicolas Oury. ‘Extensionality in the calculus of constructions’. In: *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2005, pp. 278–293 (cited on pages 99, 100, 102, 126).

[77] Abhishek Anand et al. ‘Towards Certified Meta-Programming with Typed Template-Coq’. In: *ITP 2018, Oxford, UK, July 9-12, 2018, Proceedings*. Ed. by Jeremy Avigad and Assia Mahboubi. Vol. 10895. Lecture Notes in Computer Science. Springer, 2018, pp. 20–39 (cited on page 99).

[78] Martin Hofmann. ‘Conservativity of equality reflection over intensional type theory’. In: *International Workshop on Types for Proofs and Programs*. Springer. 1995, pp. 153–164 (cited on pages 102, 109, 125, 126).

[79] Jean-Pierre Jouannaud and Pierre-Yves Strub. ‘Coq without Type Casts: A Complete Proof of Coq Modulo Theory’. In: *LPAR-21, Maun, Botswana, May 7-12, 2017*. Ed. by Thomas Eiter and David Sands. Vol. 46. EPiC Series in Computing. EasyChair, 2017, pp. 474–489 (cited on page 124).

[80] T. Altenkirch. ‘Extensional equality in intensional type theory’. In: *Proceedings of LICS*. 1999, pp. 412–420 (cited on page 125).

[81] Conor McBride. ‘Epigram: Practical programming with dependent types’. In: *International School on Advanced Functional Programming*. Springer. 2004, pp. 130–170 (cited on page 125).

[82] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. ‘Decidability of Conversion for Type Theory in Type Theory’. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 23:1–23:29. DOI: 10.1145/3158111 (cited on page 125).

[83] Thomas Streicher. *Investigations into intensional type theory*. 1993 (cited on page 126).

[84] Martin Hofmann. *Extensional constructs in intensional type theory*. CPHC/BCS distinguished dissertations. Springer, 1997 (cited on page 126).

[85] Nicolas Oury. ‘Egalité et filtrage avec types dépendants dans le calcul des constructions inductives’. PhD thesis. 2006, 1 vol., 188 p. (Cited on page 126).

[86] Matthieu Sozeau. ‘Program-ing Finger Trees in Coq’. In: *ICFP’07*. Freiburg, Germany: ACM Press, 2007, pp. 13–24 (cited on page 126).

[87] Pierre Letouzey. ‘Coq Extraction, an Overview’. In: *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*. Ed. by A. Beckmann, C. Dimitracopoulos, and B. Löwe. Vol. 5028. Lecture Notes in Computer Science. Springer-Verlag, 2008 (cited on page 129).

[88] Olivier Danvy and Kevin Millikin. ‘On the equivalence between small-step and big-step abstract machines: a simple application of lightweight fusion’. In: *Information Processing Letters* 106.3 (2008), pp. 100–109 (cited on page 155).

[89] Małgorzata Biernacka and Olivier Danvy. ‘Towards compatible and interderivable semantic specifications for the Scheme programming language, Part II: Reduction semantics and abstract machines’. In: *Semantics and algebraic specification*. Springer, 2009, pp. 186–206 (cited on page 155).

[90] Olivier Danvy. ‘Towards compatible and interderivable semantic specifications for the Scheme programming language, Part I: Denotational semantics, natural semantics, and abstract machines’. In: *Semantics and algebraic specification*. Springer, 2009, pp. 162–185 (cited on page 155).

[91] Gérard P Huet. ‘The Constructive Engine.’ In: *A Perspective in Theoretical Computer Science* 16 (1989), pp. 38–69 (cited on page 164).

[92] Beniamino Accattoli and Bruno Barras. ‘The negligible and yet subtle cost of pattern matching’. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2017, pp. 426–447 (cited on page 164).

[93] Beniamino Accattoli and Bruno Barras. ‘Environments and the complexity of abstract machines’. In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*. 2017, pp. 4–16 (cited on page 164).

[94] Thierry Coquand. ‘An algorithm for testing conversion in type theory’. In: *Logical frameworks* 1 (1991), pp. 255–279 (cited on page 175).

[95] Healfdene Goguen. ‘Justifying algorithms for $\beta\eta$-conversion’. In: *International Conference on Foundations of Software Science and Computation Structures*. Springer. 2005, pp. 410–424 (cited on page 175).

Notation

The next list describes several symbols that will be later used within the body of the document.

$\Gamma \vdash \mathcal{J}$	Judgment in some type theory (often ITT or Coq)
$\Gamma \vdash_{\mathsf{x}} \mathcal{J}$	Judgment in ETT

Special Terms

STL Simply Typed λ-calculus. 22, 24, 133
STT Setoid Type Theory. 44, 51, 52

T
TCB Trusted Code Base. 128
TTB Trusted Theory Base. 128, 129

U
UIP Uniqueness of Identity Proofs. 43, 44, 51, 52, 98, 99, 103, 111, 117, 125, 126, 179

W
WTT Weak Type Theory. xiii, xvii, 8, 47, 54, 59, 94, 96–99, 101–104, 106, 109–119, 125, 126, 179

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE NANTES

Titre : Formalisation et Méta-Théorie de la Théorie des Types

Mot clés : théorie des types, formalisation, traduction de programme, vérification de type

Résumé : Dans cette thèse, je parle de la méta-théorie de la théorie des types et de la façon de la formaliser dans un assistant de preuve.

Je me concentre d'abord sur une traduction *conservative* de la théorie des types extensionnelle vers la théorie des types intensionnelle ou faible, entièrement écrite en Coq. La première traduction consiste en une suppression de la règle de *reflection* de l'égalité, tandis que la deuxième traduction produit quelque chose de plus fort : la théorie des types faibles est une théorie des types sans notion de conversion. Le résultat de conservativité implique que la conversion n'augmente pas la puissance logique de la théorie des types.

Ensuite, je montre ma contribution au projet MetaCoq de formalisation et de spécification de Coq au sein de Coq. En particulier, j'ai travaillé sur l'implantation d'un vérificateur de type pour Coq, en Coq. Ce vérificateur de type est prouvé correct vis à vis de la spécification et peut être extrait en code OCaml et exécuté indépendamment du vérificateur de type du noyau de Coq. Pour que cela fonctionne, nous devons nous appuyer sur la méta-théorie de Coq que nous développons, en partie, dans le projet MetaCoq. Cependant, en raison des théorèmes d'incomplétude de Gödel, nous ne pouvons pas prouver la cohérence de Coq dans Coq, ce qui signifie que certaines propriétés — principalement la forte normalisation — doivent être supposées, c'est-à-dire prises comme axiomes.

Title: Formalisation and Meta-Theory of Type Theory

Keywords: type theory, extensionality, program translation, type checking

Abstract: In this thesis, I talk about the meta-theory of type theory and about how to formalise it in a proof assistant.

I first focus on a conservative translation between extensional type theory and either intensional or weak type theory, entierely written in Coq. The first translation consists in a removal of the reflection of equality rule, whereas the second translation produces something stronger: weak type theory is a type theory with no notion of conversion. The conservativity result implies that conversion doesn't increase the logical power of type theories.

Then, I show my work for the MetaCoq project of formalising and specifying Coq within Coq. In particular I worked on writing a type-checker for Coq, in Coq. This type checker is proven sound with respect to the specification and can be extracted to OCaml code and run independently of Coq's kernel type-checker. For this to work we have to rely on the meta-theory of Coq which we develop, in part, in the MetaCoq project. However, because of Gödel's incompleteness theorems, we cannot prove consistency of Coq within Coq, and this means that some properties—mainly strong normalisation—have to be assumed, *i.e.* taken as axioms.